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

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.

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 \([x = a \text{ and } y = b]\); or \([x = c \text{ and } y = d]\)), or we may choose to ignore such dependencies (leading to information of the form \([x = a \text{ or } x = c]\); and \([y = b \text{ or } y = d]\)). Jones and Muchnick refer to the former kind of analyses as relational-attribute analyses, and the latter kind as independent-attribute analyses \[5\]. The tradeoff between these methods is that independent-attribute analyses are usually more efficient but less precise than relational-attribute analyses.

When addressing a program analysis problem, it is useful to consider the computational complexity of obtaining a precise (up to 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 \[7, 10\] 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 \[4\]. 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 \[7, 10\] but not for def-use chains \[4\]. With very little conceptual and notational eff-

\(^1\) 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” \[4\], is commonly used to sidestep the problem of undecidability, and “precision” of program analyses is typically defined with respect to this assumption.
fort, a number of complexity results given in the literature [7, 10, 11, 12, 14] 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 [11, 12, 14]. For example, we show that the following analyses are EXPTIME-complete: interprocedural pointer alias analysis in the presence of two-level pointers (previous best result: PSPACE-hardness [7]), interprocedural reaching definitions in the presence of single-level pointers (previous best result: NP-hardness [14]), and interprocedural liveness analysis and available expressions in the presence of reference parameters (previous best result: NP-hardness [12]). In the process, our work exposes conceptual commonalities underlying a variety of program analyses.

To maintain continuity, the proofs of most of the results are given in the Appendix.

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 program point, e.g., in the context of constant propagation; 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 $x = \text{c}$ 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 $x_1 = c_1 \land x_2 = c_2 \land \cdots \land x_k = c_k$ 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-attribute analysis; we will show, however, that while an independent-attribute analysis is often adequate for a single value problem, there are some situations where it is necessary to resort to relational-attribute 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. 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, representing it by 1 instead. 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):

\[
\begin{align*}
\text{Prog} & ::= \text{Stmt} \\
\text{Stmt} & ::= \text{Var} = \text{Expr} \\
& \mid \text{if } (\ -\ ) \text{Stmt} ; \ldots \\
& \mid \text{else if } (\ -\ ) \text{Stmt} ; \ldots \\
& \mid \text{else Stmt} \\
& \mid \text{Label}: \text{Stmt} \\
& \mid \text{goto} \text{Label}; \\
\text{Expr} & ::= \text{Const} \mid \text{Var} \\
\text{Const} & ::= 0 \mid 1
\end{align*}
\]

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.

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 it is of some historical interest because its proof, given in Appendix 6, is essentially isomorphic to similar NP-hardness results for acyclic programs given by a number of authors [5, 7, 10, 11, 12, 14]. 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., [5, 15]). Theorem 3.4 and Corollary 4.2 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.

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 arbitrary 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 $\Sigma$ is the input alphabet; $\Gamma = \{0, 1, \ldots, ns\}$ is the tape alphabet, with 0 being

---

*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.*
the blank symbol; \( \delta \in Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\} \) is the transition function; \( q_0 \in Q \) is the initial state; and \( F = \{q_f\} \) 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. \( q_0, \ldots, q_{nq}, \) where \( nq = |Q| - 1 \): These variables represent the current state of \( M \): intuitively, \( q_i \) is 1 denotes that \( M \) is in state \( q_i \).

2. \( T_{0,0}, \ldots, T_{t_n,t_n} \), where \( t_n = |x|^k - 1 \): These variables represent the contents of \( M \)’s tape: intuitively, \( T_{i,j} = 1 \) denotes that cell \( i \) of \( M \)’s tape contains symbol \( j \).

3. \( x_0, \ldots, x_{n_x} \): these variables are temporary for copying the tape contents while we “rotate” the tape.

A configuration where \( M \) is in state \( q_i \), the tape contents are \( s_0 s_1 \ldots s_{n_t} \), and where \( M \)’s tape head is scanning the \( m+1 \)th tape square, is described by the following variable settings:

\[ \begin{align*}
Q_i &= \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} \\
X_i &= 0, \text{ for all } i \\
T_{i,j} &= \begin{cases} 1 & \text{if } s_i \equiv j \mod (n_t+1) \\ 0 & \text{otherwise} \end{cases}
\end{align*} \]

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:

\[
\begin{array}{c|c|c}
\delta(q_i, s_j) &= (q_k, s_m, L) & \delta(q_i, s_j) &= (q_k, s_m, R) \\
\hline
Q_i = Q_k; & Q_i = Q_k; & \hline
Q_k = 1; & Q_k = 1; & \hline
T_{0,j} = T_{0,m}; & T_{0,j} = T_{0,m}; & \hline
T_{m,j} = 1; & T_{m,j} = 1; & \hline
\text{goto copy_left;} & \text{goto copy_right;} & \end{array}
\]

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 \( T_{i,j} \) 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 \( q_i \) variables, or multiple \( T_{i,j} \) 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:

\[
\begin{align*}
\text{UnambiguousFinalState} &\equiv (q_0 = 0 \land q_1 = 1 \land q_2 = 0 \land \cdots \land q_{nq} = 0) \\
\text{TempClear} &\equiv (x_0 = 0 \land \cdots \land x_{n_x} = 0) \\
\text{TapeClear} &\equiv \\
& \quad (T_{0,0} = 1 \land T_{1,0} = 1 \land \cdots \land T_{t_n,0} = 1) \land \\
& \quad (T_{0,1} = 0 \land T_{1,1} = 0 \land \cdots \land T_{t_n,1} = 0) \land \cdots \land \\
& \quad (T_{0,n_x} = 0 \land T_{1,n_x} = 0 \land \cdots \land T_{t_n,n_x} = 0));
\end{align*}
\]

Intuitively, \( \text{UnambiguousFinalState} \) is true if and only if the only state variable that is 1 is \( q_1 \), corresponding to the final state of \( M \); \( \text{TempClear} \) is true if and only if the variables \( x_i \) are all 0; and \( \text{TapeClear} \) is true if and only if the contents of the variables \( T_{i,j} \) correspond to all the tape cells of \( M \) containing a blank. The proofs of the following Lemmas are given in Appendix 6.

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

\[
\begin{align*}
\text{AcceptingConfig} &\equiv \text{UnambiguousFinalState} \land \text{TempClear} \land \text{TapeClear}.
\end{align*}
\]

It is easy to show that the program \( P_{M,x} \) can be generated using \( O(\log(|M| + |x|)) \) space (Lemma 1).

**Theorem 3.4** The (binary) simultaneous value problem for programs in \( \text{BASE} \) is PSPACE-complete.

**Proof:** (sketch) PSPACE-hardness follows directly from Lemmas 3.3 and 1. 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 \( M \) to solve the problem. Details are given in Appendix 6.

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-attribute analyses [5], 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 \( \text{BASE} \) with procedures where parameters are passed by value: let the resulting language be \( \text{BASE+PROC} \). For non-recursive programs in this language, the complexity of simultaneous value problems does not change:

**Theorem 3.5** Inter-procedural simultaneous values problems for non-recursive programs in \( \text{BASE+PROC} \) is PSPACE-complete.
/* Program $P_{M,x}$ to emulate a given polynomial space-bounded Turing Machine $M$ 
on input $x$ */
/* int $Q_0$, ..., $Q_{n_q}$; 
 int $T_{0,0}$, ..., $T_{n_t,n_x}$; 
 int $X_0$, ..., $X_{n_s}$; */
{
  $T_{0,0} = \cdots$; $T_{n_t,n_s} = \cdots$;  /* initialize $T_{i,j}$ based on input string $x$ */
  $Q_0 = 1$; $Q_1 = 0$; $\cdots$ $Q_{n_q} = 0$;  /* initial state */

  Start:  /* emulation loop */
  $X_0 = 0$; $\cdots$ $X_{n_s} = 0$;  /* 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 (- )  { /* $T_{0,l} = 1$ */ MOV$\_0,l$; }
          ...
        else if (- )  { /* $T_{0,n_s} = 1$ */ MOV$\_0,n_s$; }
          }
      else if (- ) goto Done;  /* $Q_l = 1$: $q_l$ = final state */
    else if (- )
      { /* $Q_{n_q} = 1$ */
        if (- )  { /* $T_{0,0} = 1$ */ MOV$\_n_q,0$; }
          ...
        else if (- )  { /* $T_{0,l} = 1$ */ MOV$\_n_q,l$; }
          ...
        else if (- )  { /* $T_{0,n_s} = 1$ */ MOV$\_n_q,n_s$; }
          }
    }

  /* copy tape left or right */
  if (- )
    copy_right:
      $X_0 = T_{0,0}$; $\cdots$ $X_{n_s} = T_{0,n_s}$;
      $T_{0,0} = T_{1,0}$; $\cdots$ $T_{0,n_s} = T_{1,n_s}$;
      $\cdots$
      $T_{n_t,0} = X_0$; $\cdots$ $T_{n_t,n_s} = X_{n_s}$;
      goto Start;
  copy_left:
    $X_0 = T_{n_t,0}$; $\cdots$ $X_{n_s} = T_{n_t,n_s}$;
    $T_{n_t,0} = T_{n_t-1,0}$; $\cdots$ $T_{n_t,n_s} = T_{n_t-1,n_s}$;
    $\cdots$
    $T_{0,0} = X_0$; $\cdots$ $T_{0,n_s} = X_{n_s}$;
    goto Start;
  Done:
    $X_0 = 0$; $\cdots$ $X_{n_s} = 0$;
  End:
}

Figure 1: The program $P_{M,x}$ to emulate Turing machine $M$ on input $x$
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+IPTR. 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 \( \ast p \).

It is not hard to see that the single-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 3.4. By contrast to the language BASE, where the single value problem is in \( P \), the complexity of the single value problem for BASE+IPTR 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:

\[
\begin{align*}
\text{a} & = 0; \\
\text{if} \ ( - ) & \\
\{ & \text{p} = \&\text{a}; \ x = 0; \} \\
\text{else} & \\
\{ & \text{p} = \&\text{b}; \ x = 1; \} \\
\ast\text{p} & = x;
\end{align*}
\]

Suppose we are interested in the single-value problem of whether \( \text{a} = 1 \) may hold immediately after the assignment \( \ast\text{p} = x \). An independent-attribute analysis would infer that immediately after the conditional, \( \text{p} \) can point to either \( \text{a} \) or \( \text{b} \), and therefore that the assignment \( \ast\text{p} = x \) the value of \( \text{a} \) or may not be 1. A relational-attribute analysis, on the other hand, would be able to infer that the value of \( \text{a} \) 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 3.6 The single-value problem for pointer variables in BASE+IPTR can be solved in polynomial time. The single-value problem for base variables in BASE+IPTR is PSPACE-complete.

The key idea behind the PSPACE-hardness proof is to show how pointers can be used to reduce a simultaneous-value problem in BASE to a single-value problem in BASE+IPTR. Given a program \( P \) in BASE, we generate a program \( P' \) in BASE+IPTR such that each variable \( X \) in \( P \) corresponds to a pair of pointer variables \( \text{x}, \overline{\text{x}} \) in \( P' \). \( P' \) also contains a pair of global variables \( \text{Zero} \) and \( \text{One} \) that are initialized to 0 and 1 respectively: a value of 0 in \( P \) simulated using the pointer value \&\text{Zero} in \( P' \), while a value of 1 is simulated by \&\text{One}. For each variable \( X \) in \( P \), the variable \( \text{x} \) in \( P' \) specifies what the value of \( X \) is, while \( \overline{\text{x}} \) specifies what it is not (recall that a variable in a program in BASE can only take on the values 0 and 1). Then, given a simultaneous value problem \( \text{x}_1 = c_1 \land \ldots \land \text{x}_n = c_n \) for \( P' \), we generate an additional series of indirect assignments \( \ast\text{x}_i = 0 \) appended to the end of \( P' \), where \( \text{x}_i \equiv \text{x} \) if \( c_i \equiv 0 \), and \( \text{x}_i \equiv \overline{\text{x}} \) if \( c_i \equiv 1 \). The original simultaneous value problem in \( P \) is true if and only if all of these indirect assignments in \( P' \) write to the variable \text{Zero}. Thus, by checking whether the single-value problem \( \forall\text{a} = 1 \) is satisfied in \( P' \) we can solve the original simultaneous-value problem in \( P \). The details of the proof are given in Appendix 6.

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

Corollary 3.7 Precise intra-procedural constant propagation in BASE+IPTR is PSPACE-complete.

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+IPTR. 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 3.6, therefore, we have the following result:

Theorem 3.8 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 [7]; his proof can be adapted to require only two levels of indirection [9]. 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-attribute analysis, while the presence of even two levels of indirection requires a relational-attribute analysis.

A similar line of reasoning can be used to derive a recent result by Chatterjee et al. [3], 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+IPTR, i.e., in the presence of single-level pointers. The following example illustrates that an independent-attribute analysis is not enough for a precise solution to this problem, and that a relational-attribute analysis is necessary:

\[
\begin{align*}
\text{int} \ & \text{a, b, *p, *q;} \\
\text{D:} \ & \text{a = 0;} \\
\text{if} \ ( - ) \ & \{ \text{p} = \&\text{a}; \ q = &\text{b}; \} \text{ else } \{ \ q = &\text{a}; \ p = &\text{b}; \} \\
\ast\text{p} & = 1; \\
\ast\text{q} & = 1;
\end{align*}
\]

L:
We want to know whether the definition labelled $\mathfrak{D}$ can reach the program point labelled $L$. An independent-attribute 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 $\mathfrak{D}$. A similar reasoning would apply to $q$ and the indirect assignment $q = 1$. Such an analysis would therefore conclude that definition $\mathfrak{D}$ could reach $L$. A relational-attribute 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 $\mathfrak{D}$—i.e., definition $\mathfrak{D}$ does not reach $L$. Thus, the independent-attribute analysis is not precise, and a relational-attribute analysis is necessary. The following theorem discusses the complexity of precise analyses; its proof uses a reduction very similar to that for Theorem 3.6.

Theorem 3.9 The determination of precise solutions for the following intra-procedural analysis problems for base variables in programs in $\text{BASE+IPTR}$ is $\text{PSPACE}$-complete: (a) reaching definitions; (b) live variables; and (c) available expressions.

Theorem 3.9 improves on a result due to Pandey, Landi, and Ryder, who showed that the problem of computing inter-procedural def-use chains in the presence of single-level pointers is $\text{NP}$-hard [14].

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 $\text{BASE+PROC}$ (i.e., the base language together with procedures). Each program now has a distinguished global variable $\text{NoErr}$ whose value is initially $1$. We add a statement $\text{Error-if-Zero}(\cdots)$ that behaves as follows: when $\text{Error-if-Zero}(x)$ is executed, $\text{NoErr}$ 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 $\text{malloc}(\cdots)$ 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 $\text{BASE+PROC}$ as $\text{BASE+PROC+Err}$.

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

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; $\Sigma$ is the input alphabet; $\Gamma$ is the tape alphabet; $\delta: Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$ is the transition function; $q_0 \in Q$ is the initial state; and $\tau: Q \rightarrow \{\text{accept}, \text{reject}, \forall, \exists\}$ is a labelling function on states.\(^3\)

\(^3\)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. [2]), so for simplicity we restrict ourselves to alternating Turing machines without negating states.

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 $q\ell$ where $\ell$ is a state and $x$ the tape contents.

The notion of acceptance for alternating Turing machines is a generalization of that for ordinary nondeterministic 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. [6]. 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 $\neg u'$ that such $q u' + q u''$; if $p$ is an internal node of a tree with label $qu$ and $q$ is a universal state with successors $q'$ and $q''$, 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 $q_0 x$.

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 $n_0 = p(|x|) - 1$ and $n_s = |\Gamma| - 1$. We generate a program $P_{M,x}$ in $\text{BASE+PROC+Err}$ that simulates the behavior of $M$ on input $x$. There is a function $n_2(x)$ for each state $q$ of $M$. Each such function has a tuple of parameters $\text{T}_{0,0}, \ldots, \text{T}_{n,ns}$ 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 variables 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 $\text{if-then}$ construct, while moves to the successors of a universal state are simulated by a sequence of function calls. The crucial point in the construction is that the $\text{Error-if-Zero}(\cdots)$ construct is used to keep track of whether anything "goes wrong" along an execution path: it sets the global variable $\text{NoErr}$, 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 $\text{NoErr}$ 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 $\text{NoErr}$ can be used to determine whether that path corresponds to a valid accepting computation of $M$.

The code necessary to simulate $M$'s actions when it makes a transition from state $q_i$ to state $q_j$ upon scanning a tape cell containing symbol $s_j$ is represented by $\text{TRANSITION}(q_i, s_j, q_k)$ and is defined as follows:
The three first lines of this code, which assign to $T_{i,j}$, restore the contents of the tape; the four line uses Error-if-Zero() to verify that the symbol on the scanned tape cell is 1; the next two lines update the tape. After this the tape is rotated to simulate the movement of the tape head, with COPY_LEFT and COPY_RIGHT corresponding to the code fragments labelled copy_left and copy_right respectively in Figure 1. Finally, a function call is used to simulate the computation from the state to which control is transferred.

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

1. $q_i$ is an accepting state. The function $f_{q_i}$ is defined as

\[
f_{q_i}(T_{0,0}, \ldots, T_{n,t,nx}) \{ /\ast \; do \; nothing \;/\ast \}
\]

2. $q_i$ is a rejecting state. The function $f_{q_i}$ is defined as

\[
f_{q_i}(T_{0,0}, \ldots, T_{n,t,nx}) \{ \text{Error-if-Zero(0);} \}
\]

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 $f_{q_i}$ is defined as

\[
f_{q_i}(T_{0,0}, \ldots, T_{n,t,nx})
\]

\[
\{ \text{local} \; x_{0,0} = T_{0,0}, \ldots, x_{n,t,nx} = T_{n,t,nx};
\]

\[
\text{if} \; (-) \; \{ /* \; moves \; on \; s_j */ \text{TRANSITION}(q_i,s_j,q_{j'}); \text{TRANSITION}(q_i,s_j,q_{j''}); \}
\]

\[
\ldots \text{else} \; \{ /* \; moves \; on \; s_k */ \text{TRANSITION}(q_i,s_k,q_{k'}); \text{TRANSITION}(q_i,s_k,q_{k''}); \}
\]

\[
\}
\]

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 $f_{q_i}$ is defined as

\[
f_{q_i}(T_{0,0}, \ldots, T_{n,t,nx})
\]

\[
\{ \text{local} \; x_{0,0} = T_{0,0}, \ldots, x_{n,t,nx} = T_{n,t,nx};
\]

\[
\text{if} \; (-) \; \{ /* \; moves \; on \; s_j */ \text{TRANSITION}(q_i,s_j,q_{j'});
\]

\[
\text{if} \; (-) \; \{ /* \; moves \; on \; s_j */ \}
\]

\[
\text{TRANSITION}(q_i,s_j,q_{j''});
\]

\[
\text{else}
\]

\[
\text{TRANSITION}(q_i,s_k,q_{k'});
\]

\[
\text{else}
\]

\[
\text{TRANSITION}(q_i,s_k,q_{k''});
\]

\[
\}
\]

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

\[
\text{main()}
\]

\[
\{ \text{START}
\}
\]

\[
\text{local} \; T_{0,0}, \ldots, T_{n,t,nx}:
\]

\[
\text{INIT_TAPE; /* initialize } T_{i,j} \text{ based on } x */
\]

\[
\text{f}_{M}(T_{0,0}, \ldots, T_{n,t,nx});
\]

\[
\text{END}
\]

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 $(f, \bar{a})$ in such a tree has children $(f_1, \bar{a}_1), \ldots, (f_k, \bar{a}_k)$ if there is an execution path in $P_{M,x}$, starting with the call $f(\bar{a})$ with the value of NoErr = 1, that executes the procedure calls $f_1(\bar{a}_1), \ldots, f_k(\bar{a}_k)$ 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, $T_{i,j} \approx u$ denotes that the values of the tuple of variables $(T_{0,0}, \ldots, T_{n,t,nx})$ in $P_{M,x}$ correctly reflect the tape contents $u$ in $M$. The proofs are given in Appendix 6.

**Theorem 4.1** $P_{M,x}$ has a valid call tree with root $(f_q,T_{i,j})$ if and only if $M$ has an accepting computation tree with root $q_u$, where $T_{i,j} \approx u$.

From this, it is straightforward to show that $M$ accepts $x$ if and only if $P_{M,x}$ has an execution path at the end of which we have NoErr = 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 4.2** The inter-procedural single-value problem for Base+Proc+Err is EXPTIME-hard.

It is interesting and instructive to compare this result with Theorem 3.4. For the intra-procedural case considered in Theorem 3.4, 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 4.2 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, 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 3.8).
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 $\text{BASE}+\text{PROC}+\text{1PTR}$. The proof, which is given in Appendix 6, relies on 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 4.3** The inter-procedural single-value problem for base variables in $\text{BASE}+\text{PROC}+\text{1PTR}$ is EXPTIME-complete.

**Corollary 4.4** The complexity of precise inter-procedural pointer alias analysis in the presence of 2-level pointers is EXPTIME-complete.

**Corollary 4.5** The determination of precise solutions for the following inter-procedural analysis problems for base variables in $\text{BASE}+\text{PROC}+\text{1PTR}$ 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 $\text{BASE}$ 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-attribute analysis is inadequate for solving the single value problem in this case. To see this, consider the following program:

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

We want to know whether or not $a = 1$ can hold immediately after the conditional in $\text{main}()$. We need a relational-attribute 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-attribute analysis is inadequate for this single value problem.

We have the following results, whose proofs are given in Appendix 6:

**Theorem 4.6** The single value problem for $\text{BASE}$ extended with procedures with reference parameters is $\text{PSPACE}$-complete for non-recursive programs and EXPTIME-complete for arbitrary programs.

**Corollary 4.7** Precise inter-procedural liveness analysis and available expressions analysis for $\text{BASE}$ extended with procedures with reference parameters are both $\text{PSPACE}$-complete for non-recursive programs, and EXPTIME-complete for arbitrary programs.

This result corrects a minor flaw in Myers' original proof of the difficulty of such analysis problems [12]. 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 $\text{BASE}$ 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 3.8 and Corollary 4.4, improves on an NP-hardness result by Zhang and Ryder [18]:

**Theorem 4.8** Precise control flow analysis in the presence of function pointers is $\text{PSPACE}$-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 new results: Corollary 3.7, Theorem 3.9(b,e), Corollary 4.4, and Corollary 4.5.

2. **Improvements to Existing Results**: Theorem 3.9 and Corollary 4.5 improve on a result by Pande et al. [14]. Corollary 4.4 improves on a result by Landi [7, 10]. Theorem 4.6 and Corollary 4.7 improve on a result by Myers [12].

3. **Explanation of Previous Results**: Theorems 3.6 and 3.8 explain the underlying reasons for Landi's complexity results for pointer alias analysis [7, 10]. Theorems 3.8 and 3.9(a) together explain why single-level pointers are hard to deal with when constructing intra-procedural def-use chains but not when considering inter-procedural pointer analyses. Theorem 4.8 explains the difficulty of inter-procedural control flow analysis in the presence of function pointers.

The distinction between independent-attribute analyses and relational-attribute analyses was first defined by Jones and Muchnick [5], who also examined the complexity of these approaches to program analysis. They showed that independent-attribute analyses over a fixed finite domain has worst case complexity that is polynomial in the size of the program, while relational-attribute 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 [5]. Variations on the basic idea of this proof have been used for NP-hardness results by a number of authors [7, 10, 11, 12, 14], 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 kind of functions considered [13]; this work is aimed at finding the cost of particular formulations of dataflow analysis problems. By contrast, our work focuses on the inherent 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-attribute analyses for such programs are NP-complete (Theorem 3.2).

Many researchers have given complexity results for specific program analysis problems [see, for example, [7, 10, 11, 12, 14, 16, 17]]. With a few exceptions, e.g., [16, 17], 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-attribute 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

Appendix: Proofs of Theorems

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.

Proof: A straightforward independent attribute analysis suffices in this case. Jones and Muchnick ([5], 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.

Theorem 3.2 The simultaneous values 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 $\varphi$ each containing three literals, whether $\varphi$ is satisfiable. This problem is known to be NP-complete [4]. Given a formula $\varphi \equiv (x_{11} \lor \cdots \lor x_{1k}) \land \cdots \land (x_{m1} \lor \cdots \lor x_{mk})$ over a set of variables $x_{ij}$, where each of the literals $u_{ij}$ is either a variable or its negation, we generate a program $P_\varphi$ with variables $x_{1t}, \ldots, x_{nt}, x_{nf}, \ldots, x_{nf}, c_1, \ldots, c_m$, of the following form:

$$
\begin{align*}
  &\text{if } ( - ) \{ \\
  &\quad x_{1t} = 0; x_{nf} = 1; \\
  &\} \\
  &\text{else } \{ \\
  &\quad x_{1t} = 1; x_{nf} = 0; \\
  &\} \\
  &\text{if } ( - ) \{ \\
  &\quad x_{2t} = 0; x_{2f} = 1; \\
  &\} \\
  &\text{else } \{ \\
  &\quad x_{2t} = 1; x_{2f} = 0; \\
  &\} \\
  &\ldots \\
  &\text{if } ( - ) \{ \\
  &\quad x_{nt} = 0; x_{nf} = 1; \\
  &\} \\
  &\text{else } \{ \\
  &\quad x_{nt} = 1; x_{nf} = 0; \\
  &\} \\
  &\text{if } ( - ) \text{ c1} = w_{11}; \\
  &\text{else if } ( - ) \text{ c1} = w_{12}; \\
  &\text{else } \text{ c1} = w_{13}; \\
  &\ldots \\
  &\text{if } ( - ) \text{ cm} = w_{m1}; \\
  &\text{else if } ( - ) \text{ cm} = w_{m2}; \\
  &\text{else cm} = w_{m3}; \\
\end{align*}
$$

Here, $w_{1j}$ are defined as follows: if the literal $u_{ij}$ is a variable $x_i$ for some $k$, then $w_{ij} = x_{ik}$; if the literal $u_{ij}$ is a negated variable $\bar{x}_i$ for some $k$, then $w_{ij} = x_{ik}$. Intuitively, $x_{1t} = 1$ in $P_\varphi$ represents an assignment of a truth value true to $x_i$ in $\varphi$, while $x_{nf} = 1$ represents a truth value false. Each path through the first group of conditionals represents a truth assignment for the variables of $\varphi$. 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 $c_i$. The simultaneous value problem we pose at the program point labeled L is

$$c_1 = 1 \land \ldots \land c_m = 1.$$ 

This is true if and only if there is a path through all of the statements in $P_\varphi$ that assigns 1 to each of the $c_i$, i.e., if and only if there is a truth assignment to the variables of $\varphi$ 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.

Lemma 3.3 A given polynomial-space-bounded Turing machine $M$ accepts its input $x$ if and only if AcceptingConfig may hold at the point in $P_{M,x}$ labelled End, where

$$\text{AcceptingConfig} \equiv \text{UnambiguousFinalState} \land \text{TapeClear} \land \text{TapeClear}.$$ 

Proof: (sketch) Let a configuration $\sigma$ of $M$ correspond to a state $\hat{\sigma}$ of $P_{M,x}$, written $\sigma \sim \hat{\sigma}$ if and only if the following holds: in $\sigma$, $M$ is in state $q_s$, scanning tape cell $m$, with tape contents $s_0s_1\ldots s_u$; and in $\hat{\sigma}$, $P_{M,x}$ has the following values for its variables, with control at the point labelled Dispatch:

$$Q_i = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

$$X_i = 0, \text{for all } i;$$

$$T_{ij} = \begin{cases} 1 & \text{if } s_{i(m+1)} \bmod (m+1) = j \\ 0 & \text{otherwise} \end{cases}$$

We use the following notation: if $M$ can go from configuration $\sigma$ to configuration $\beta$ via a sequence of transitions, we write $\sigma \vdash^* \beta$. 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 $u \vdash^* v$.

We first show that, if, given configurations $\sigma$ and $\beta$ for $M$ and states $\hat{\sigma}$ and $\hat{\beta}$ for $P_{M,x}$, such that $\sigma \sim \hat{\sigma}$ and $\beta \sim \hat{\beta}$, if $\sigma \vdash^* \beta$ then $\hat{\sigma} \vdash^* \hat{\beta}$. Pictorially:

$$\sigma \overset{L_M}{\longrightarrow} \beta$$

$$\sigma \overset{L_P}{\longrightarrow} \beta$$

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 $\sigma$, $\beta$ and $\gamma$ of $M$ and states $\hat{\sigma}$ and $\hat{\gamma}$ of $P_{M,x}$, with $\sigma \sim \hat{\sigma}$ and $\beta \sim \hat{\gamma}$, such that $\sigma \vdash^* \beta$ $M$ goes from state $q_s$, scanning tape symbol $c$, to state $q_b$. In $P_{M,x}$, consider state resulting from $\gamma$ by taking the path from the point labelled Dispatch to that referred to as MOV$_{ij}$. An examination of the definition of the code corresponding to MOV$_{ij}$ shows that the resulting state $\hat{\beta}$ of $P_{M,x}$ corresponds
to the configuration $\beta$ of $M$ after the $n + 1^{st}$ transition. The claim follows.

Since, from the definition of $P_{M,s}$, the initial configuration of $M$ corresponds to the state of $P_{M,s}$ 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 $P_{M,s}$ 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 $P_{M,s}$ 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 $P_{M,s}$, consisting of “good” guesses, at the end of which AcceptingConfig holds at the point End.

Next, consider any path in $P_{M,s}$ that does not correspond to a valid computation of $M$. This must come from a “bad guess” in $P_{M,s}$ of either the state (variables $q_i$) or the tape symbol (variables $T_{j,k}$), 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 $q_i$ set to 1, or more than one of the variables $T_{j,k}$ 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 $P_{M,s}$.

Together, this follows from these that AcceptingConfig will hold at the point labelled End if and only if $M$ accepts $x$.

**Lemma 1** Given a polynomial-space-bounded Turing machine $M$ and input $x$, the program $P_{M,s}$ 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 $P_{M,s}$ 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 $X$ followed by the code for the transitions of $M$; and the code for “rotating” the tape, labelled copyright and copyleft, 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 $\log(\log(|\Gamma| \times p(|x|))) = \log |\Gamma| + \log p(|x|)$ 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 $O(|\Gamma|) = O(1)$ bits. The outer if statement in the emulation loop consists of $|Q|$ cases, where each case (with the exception of that for $q_1 = 1$) 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 $O(Q \times |\Gamma|) = \log |\Gamma| + \log |\Gamma| = O(\log |M|)$. Thus, the total space required for this component is $O(\log |M|)$.

- Each of the copyright and copyleft portions of the program consists of $|\Gamma| \times |p(|x|)| = O(|\Gamma| \times p(|x|))$ 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 3.4** The simultaneous value problem for programs in Base is PSPACE-complete.

**Proof:** (sketch) PSPACE-hardness follows directly from Lemmas 3.3 and 1. 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 if 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 $T_{exc}$ that contains a list of (variable, value) pairs, one for each program variable. $T_{exc}$ 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 $T_{exc}$ 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 $T_{exc}$ 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 $T_{exc}$. 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 $T_{exc}$ are $O(n)$. It follows that $M$ is polynomial-space-bounded.
Theorem 3.5 The class of Inter-procedural simultaneous values problems for non-recursive programs in BASE+PROC is PSPACE-complete.

Proof: PSPACE-hardness follows from Theorem 3.4. 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 non-deterministic Turing machine similar to that used to show membership in PSPACE in the proof of Theorem 3.4, 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.

Theorem 3.6 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-attribute analysis is sufficient. Reasoning as for Theorem 3.1 shows that this is solvable in polynomial time.

To prove PSPACE-hardness of the base variable case, 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 the idea is to generate a program \( P' \) as follows (here, \( X_1, X_2, \ldots \) denote variables in \( P \) while \( x_1, x_2, \ldots \) denote variables in \( P' \)). The program \( P' \) contains two variables, \( \text{Zero} \) and \( \text{One} \), that are initialized to the constants 0 and 1 respectively. For each variable \( X \) in \( P \) we have two variables \( \overline{X} \) and \( \overline{X} \) in \( P' \). Assignments in \( P \) are translated into \( P' \) as follows:

- An assignment \( X = 0 \) in \( P \) is translated to a pair of assignments \( X = \&\text{Zero}; \overline{X} = \&\text{One} \) in \( P' \).
- An assignment \( X = 1 \) in \( P \) is translated to a pair of assignments \( X = \&\text{One}; \overline{X} = \&\text{Zero} \) in \( P' \).
- An assignment \( X = Y \) in \( P \) is translated to a pair of assignments \( X = Y; \overline{X} = \overline{Y} \) in \( P' \).

The intuition is that \( X \) tells us what the value of the original variable \( X \) is, while \( \overline{X} \) 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 the form \( X_1 = c_1 \land \cdots \land X_n = c_n \) at a point \( p \) in the original program \( P \), where \( c_i \in \{0, 1\} \). Consider the conjunction \( X_1 = c_1 \); if \( c_1 \equiv 0 \) then, in the generated program program, we want to test whether \( X_1 \) points to \( \text{Zero} \). If \( c_1 \equiv 1 \), we want to test whether \( X_1 \) points to \( \text{One} \); or equivalently, whether \( X_1 \) does not point to \( \text{Zero} \) (since the variables \( \text{One} \) and \( \text{Zero} \) are the only base variables in the program, and hence the only things that \( X_1 \) could point to); or equivalently, whether \( X_1 \) points to \( \text{Zero} \). 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 \( X_1 \rightsquigarrow \text{Zero} \land \cdots \land x_n \rightsquigarrow \text{Zero} \), where \( x_i \) is \( X_i \) if \( c_i \equiv 0 \), and \( \overline{X}_i \) if \( c_i \equiv 1 \). We do this by inserting the following code fragment at the point \( p' \) (where \( x_i \) is either \( X_i \) or \( \overline{X}_i \), depending on whether \( c_i \) is 0 or not, as just described).

\[
\text{if ( - ) } \{
\begin{align*}
*_{\text{Zero}} &= 0; \\
\ldots
\end{align*}
\}
L: \text{goto End}; /* go to end of pgm, halt */
\]

If, for some execution path leading to \( p' \) in the program \( P \), \( x_j \rightsquigarrow \text{Zero} \) for each \( x_j \), then all of the assignments \( *_{\text{Zero}} = 0 \) will write to the variable \( \text{Zero} \). This means that the initial assignment of 1 to the variable \( \text{One} \) will not be overwritten (since there are no other assignments to either \( \text{Zero} \) or \( \text{One} \), or any indirect assignments through any of the variables \( X_i \) or \( \overline{X}_i \), elsewhere in the program), so \( \text{One} \) 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 \( x_j \not\rightsquigarrow \text{Zero} \) for some \( j \), it must be the case that \( x_j \rightsquigarrow \text{One} \), which means that the assignment \( *_{\text{One}} = 0 \) will overwrite the initial assignment to \( \text{One} \). Thus, by answering the single-value problem of whether or not \( \text{One} \) 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 3.6.

Theorem 4.1 \( P_{M,x} \) has a valid call tree with root \((f_q, \overline{t}_{i_{ij}})\) if and only if \( M \) has an accepting computation tree with root \( qu \), where \( \overline{t}_{i_{ij}} \approx u \).

Proof: We first show that \( P_{M,x} \) has a valid call tree \( T_F \) with root \((f_q, \overline{t}_{i_{ij}})\) if \( M \) has an accepting computation tree \( T_M \) with root \( qu \), where \( \overline{t}_{i_{ij}} \approx u \). 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 \((f_q, \overline{t}_{i_{ij}})\), where \( \overline{t}_{i_{ij}} \approx u \), is a valid call tree.

For the inductive case, assume that \( P_{M,x} \) has a valid call tree with root \((f_q', \overline{t}')\) whenever \( M \) has an accepting computation tree with root \( q' \) and height \( k \), where \( \overline{t}' \approx u' \), 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 \( \overline{t}_{i_{ij}} \approx u \). We have two possibilities:

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

Suppose that the transition from \( q \) to \( q' \) occurs on tape symbol \( s_i \). From the construction of \( P_{M,x} \), the function \( f_q \) contains an execution path through the code defined by \( \text{TRANSITION}(q, s_i, q') \) that verifies that the tape symbol scanned is \( s_i \), adjusts the variables \( \overline{t}_{i_{ij}} \) as necessary to correspond to the tape contents \( u' \), and calls \( f_{q'} \). It follows from this that a tree with root \((f_{q''}, \overline{t}'')\) that has a single subtree \( T_F \) 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 \( q' \) and \( q'' \), and that the subtrees \( T_{M,q} \) and \( T_{M,q''} \) 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 $T_{p,x}$, with root
labelled $(f_x, \bar{y}_x)$, and $T'_{p,x}$ with root labelled $(f_{x'}, \bar{y'}_x)$, where
$\bar{y'}_x \approx u'$ and $\bar{y'}_x \approx u''$.

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 $f_{s_i}$ contains an execution path

$$
\text{if} \ (\ - ) \ \{ \\
\text{TRANSITION}(q, s_i, q'); \\
\text{TRANSITION}(q, s_i, q'); \\
\}
$$

that simulates each of these transitions by verifying that the
tape symbol scanned is $s_i$, adjusting the vari-
ables $\bar{y}_{i,j}$ as necessary, and calling the appropriate func-
tion in $P_{M,x}$. It follows that a tree with root $(f_{s_i}, \bar{y}_{i,j})$
that has two subtrees $T_{p,x}$ and $T'_{p,x}$ 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 0.2** $M$ accepts $x$ if and only if there is an execu-
tion path $p$ in $P_{M,x}$ from the program point labelled $\text{Start}$
to that labelled $\text{End}$ such $\text{NoErr} = 1$ at the end of $p$.

**Proof:** We observe that by construction of $P_{M,x}$, the code
at the point labelled $\text{Start}$ sets $\text{NoErr}$ to 1 and initializes
the variables $\bar{y}_{i,j}$ 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 The-
orem 4.1 that there is a valid call tree $T_F$ for $P_{M,x}$, with root
$(f_{s_i}, \bar{y}_{i,j})$ where $\bar{y}_{i,j} \approx x$. This means that there is an execu-
tion path in $P_{M,x}$ from $\text{Start}$ to $\text{End}$ such that $\text{NoErr} = 1$
at $\text{End}$.

Suppose that $M$ does not accept $x$, i.e., there is no ac-
cepting computation tree $T_M$ rooted at $q_0x$. From The-
orem 4.1, it follows that there is no valid call tree in $P_{M,x}$, with root
$(f_{s_i}, \bar{y}_{i,j})$ such that $\bar{y}_{i,j} \approx x$. It follows that there is no execu-
tion path from $\text{Start}$ to $\text{End}$ along which the value of
$\text{NoErr}$ remains 1.

**Theorem 4.3** 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+PROC+1PTR (here,
$X_1, X_2, \ldots$ denote variables in $P$ while $x_1, x_2, \ldots$ denote vari-
ables 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
&0One.

2. For each $n$-argument function $f$ in $P$ there is an $n-$
argument function $f$ in $P'$. For each such pair of corre-
sponding 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

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

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

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

Conditionals are translated unchanged.

4. A statement Error-if-Zero ($X$) is translated to 
$'\ast\text{NoErr} = \ast X.'$

5. The single-value problem $'\text{NoErr} = 0'$ in $P$
corresponds to the base-variable single-value problem $'\text{One} = 1'$ 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 to $V$ 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 execu-
tion path $p$ in $P$. If $p$ does not contain any occurrence of
a Error-if-Zero($\sim$) statement, the execution along the cor-
responding 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 state-
ment Error-if-Zero($X$), then the corresponding statement in
$P'$ is $'\ast\text{NoErr} = \ast X.'$ We have the following possibilities:

1. $\text{NoErr}$ points to One, $X$ points to Zero, 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 $\text{NoErr}$ has no effect on the value of any
variable in $P'$. This parallels the behavior of $P$.

2. $\text{NoErr}$ points to One and $X$ points to Zero (correspond-
ing 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. $\text{NoErr}$ points to One, but the value of One is 0 (due to
an assignment 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 $\ast X$ is 0, so the
assignment $'\ast\text{NoErr} = \ast X.$ does not change the value
of any variable in $P'$. In particular this means that
$\ast\text{NoErr}$ remains 0. Again, this parallels the behavior of $P$.
Thus, at the end of the execution of \( P' \), the variable \( \text{one} \) has the value 1 if and only if, at the end of the corresponding execution path in \( P \), the value of \( \text{Nonce} \) is 1. The reduction described above establishes that the inter-procedural single-value problem for base variables in \( \text{BASE+IPTR} \) is \( \text{EXPTIME-hard.} \)

We next show how a program \( P \) in \( \text{BASE+PROC+IPTR} \) 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: \( \text{Globals} \), \( \text{AnticipatedGlobals} \), \( \text{TempGlobals} \), and \( \text{Locals} \). \( \text{Globals} \) contains the current snapshot of the global variables. \( \text{AnticipatedGlobals} \) shows the Globals as we expect them to be upon return from the current subroutine. \( \text{TempGlobals} \) is an auxiliary region big enough to hold \( \text{Globals} \) and \( \text{AnticipatedGlobals} \). \( \text{Locals} \) 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 \( \text{Globals} \) and \( \text{Locals} \) 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 \( \text{Globals} \) with \( \text{AnticipatedGlobals} \) 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 \( \text{AnticipatedGlobals} \) into \( \text{TempGlobals} \) \( M_P \) then guesses the effect of the subroutine call on \( \text{Globals} \) and writes this guess into \( \text{AnticipatedGlobals} \). 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 \( \text{AnticipatedGlobals} \) was guessed correctly. The other successor continues interpreting subroutine \( f \) assuming the call to \( g \) behaves as expected, i.e., it copies \( \text{AnticipatedGlobals} \) to \( \text{Globals} \) and \( \text{TempGlobals} \) back to \( \text{AnticipatedGlobals} \).

The subroutine \( \text{main()} \), where the simulation begins is handled slightly differently. At the beginning of \( \text{main()} \) \( \text{Globals} \) is initialized and upon return from \( \text{main} \) \( M_P \) 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 \( \text{main} \)—we can make \( M_P \) test the condition at the end of \( \text{main} \) and either go into an accepting state if the condition is satisfied or in a rejecting state otherwise.

\[ \text{Theorem 4.6} \quad \text{The single value problem for } \text{BASE} \text{ extended with procedures with reference parameters is } \text{PSPACE-complete for non-recursive programs and } \text{EXPTIME-complete for arbitrary programs.} \]

\[ \text{Proof:} \ (\text{sketch}) \quad \text{The proof is very similar to that for Theorem 4.3, the primary difference being that instead of explicit pointer variables we use reference parameters. Each procedure in the program takes two additional arguments that are references to the global variables } \text{Zero} \text{ and } \text{one}. \text{ Instead of explicit assignments of } \&\text{Zero} \text{ and } \&\text{one}, \text{as in the construction in the proof of Theorem 4.3, we use these reference parameters. The remainder of the proof remains essentially unchanged.} \]