On the Sequential Nature of Interprocedural Program-Analysis Problems

Thomas Reps
University of Wisconsin[1]

Abstract In this paper, we study two interprocedural program-analysis problems—interprocedural slicing and interprocedural dataflow analysis—and present the following results:

- Interprocedural slicing is log-space complete for $\mathcal{P}$.
- The problem of obtaining "meet-over-all-valid-paths" solutions to interprocedural versions of distributive dataflow-analysis problems is $\mathcal{P}$-hard.
- Obtaining "meet-over-all-valid-paths" solutions to interprocedural versions of distributive dataflow-analysis problems that involve finite sets of dataflow facts (such as the classical "gen/kill" problems) is log-space complete for $\mathcal{P}$.

These results provide evidence that there do not exist fast ($\mathcal{NC}$-class) parallel algorithms for interprocedural slicing and precise interprocedural dataflow analysis (unless $\mathcal{P} = \mathcal{NC}$). That is, it is unlikely that there are algorithms for interprocedural slicing and precise interprocedural dataflow analysis for which the number of processors is bounded by a polynomial in the size of the input, and whose running time is bounded by a polynomial in the logarithm of the size of the input. This suggests that there are limitations on the ability to use parallelism to overcome compiler bottlenecks due to expensive interprocedural-analysis computations.

1. Introduction

Interprocedural analysis concerns the static examination of a program that consists of multiple procedures. Its purpose is to determine certain kinds of summary information associated with the elements of a program (such as reaching definitions, available expressions, live variables, *etc.*). Interprocedural program analysis is a fundamental part of the process of compiling programs to run efficiently. Information gathered via interprocedural analysis is crucial for determining how optimization, vectorization, and parallelization transformations can applied most effectively. This paper concerns the computational complexity of interprocedural program-analysis problems.

In our work, we make the assumption that interprocedural analyses are to take into account only "valid paths" in the program (*i.e.*, paths in which each procedure-return edge must have a corresponding procedure-call edge). The valid-paths assumption relates to the precision of the analysis: when an analysis algorithm reports only "effects" transmitted along valid paths, it filters out a certain class of infeasible execution paths and thereby is able to report more precise (static) estimates of what can happen at runtime. The concept of valid paths arises in both the interprocedural-slicing problem [9,10,27] and in "flow-sensitive" interprocedural dataflow-analysis problems [32,23,2,17,15,18,19,28,29,31]. For example, Sharir and Pnueli generalized Kildall's concept of the "meet-over-all-paths" solution of an *intra*procedural dataflow-analysis problem [14] to the "meet-over-all-valid-paths" solution of an *inter*procedural dataflow-analysis problem [32].

Interprocedural analysis is generally expensive, and can be the bottleneck in compilers that employ it. This raises the question of whether it might be possible to devise fast parallel algorithms for interprocedural analysis problems. The results presented in this paper show that this is not likely to be the case.

Our results can be summarized as follows:

- Interprocedural slicing is log-space complete for $\mathcal{P}$.
- The problem of obtaining "meet-over-all-valid-paths" solutions to interprocedural versions of distributive dataflow-analysis problems is $\mathcal{P}$-hard.

---

[1]Work performed at the Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant CCR-9100424, and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

Author's address: Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706.
Electronic mail: reps@cs.wisc.edu.

- Obtaining "meet-over-all-valid-paths" solutions to interprocedural versions of distributive dataflow-analysis problems that involve finite sets of dataflow facts is log-space complete for $\mathcal{P}$.

Note that the class of problems referred to in the third bullet point includes all of the classical "gen/kill" (or "bit-vector") problems.

The practical implications of these results are that there do not exist fast ($\mathcal{NC}$-class) parallel algorithms for interprocedural slicing and precise interprocedural dataflow analysis (unless $\mathcal{P} = \mathcal{NC}$). That is, it is unlikely that there are algorithms for interprocedural slicing and precise interprocedural dataflow analysis for which the number of processors is bounded by a polynomial in the size of the input, and whose running time is bounded by a polynomial in the logarithm of the size of the input.

The remainder of the paper is organized into four sections. Section 2 introduces some terminology that is used in the body of the paper. Section 3 concerns interprocedural slicing. Section 4 concerns interprocedural dataflow analysis. Section 5 discusses related work.

2. Terminology and Assumptions

One accepted notion of how to characterize problems with "fast parallel algorithms" is by the complexity class $\mathcal{NC}$, which is the class of all languages decidable by a PRAM that operates in polylogarithmic time while using a polynomially bounded number of processors.

The class $\mathcal{NC}$ has an intimate connection with the class of languages that are log-space complete for $\mathcal{P}$ (or "$\mathcal{P}$-complete under log-space reductions"). A language that is log-space complete for $\mathcal{P}$ has the property that if it is recognizable in space $\log^k(\cdot)$, then every language in $\mathcal{P}$ (a.k.a. $\mathcal{PTIME}$) is also recognizable in space $\log^k(\cdot)$ [11]. If any language that is log-space complete for $\mathcal{P}$ is decidable by an $\mathcal{NC}$ parallel algorithm, then every language in $\mathcal{P}$ has an $\mathcal{NC}$ parallel algorithm. (See, for example, [25, pp. 377].) That is, the problems that are log-space complete for $\mathcal{P}$ do not have $\mathcal{NC}$ parallel algorithms unless $\mathcal{P} = \mathcal{NC}$. At the present time it is unknown whether $\mathcal{P} = \mathcal{NC}$, but it is considered to be unlikely.

The interprocedural-analysis problems examined in the paper are deliberately presented in a very trimmed-down form. In particular, we assume only a language with assignment statements, conditional statements, and non-recursive procedure calls (with value-result parameter passing). The reason we work with such an impoverished language is purely for expository purposes—neither a looping construct nor recursion is necessary in order to obtain our results about the inherent difficulties of interprocedural-analysis problems. If the language is extended with while-loops and recursive procedures, our $\mathcal{P}$-completeness results for interprocedural slicing and interprocedural dataflow analysis continue to hold. For even richer languages the interprocedural-slicing and interprocedural dataflow-analysis problems are $\mathcal{P}$-hard, but they may not be $\mathcal{P}$-complete (because for richer languages these problems may not be solvable in polynomial time).

Remark. Discussing the problems in their trimmed-down form allows one to gain greater insight into exactly what aspects of an interprocedural-analysis problem introduce what computational limitations on algorithms for these problems. What this paper demonstrates is that, *by itself, the aspect of considering only* ***valid paths*** *in an interprocedural-analysis problem imposes some inherent computational limitations*, namely that the problem is unlikely to have a fast ($\mathcal{NC}$-class) parallel algorithm.[2] For example, for interprocedural dataflow analysis, if we drop the condition that only valid paths are to be considered, less precise (but safe) solutions can be obtained by treating an *inter*procedural dataflow-analysis problem as one large *intra*procedural problem. For such problems there *does* exist a fast ($\mathcal{NC}$-class) parallel algorithm. (Compare, for example, Theorem 4.3 and Corollary 4.4 against Theorem 4.5.) □

[2]See also the discussion in Section 5 relating the work reported in this paper with Landi and Ryder's work on other aspects of interprocedural analysis and inherent computational limitations [17].

3. Interprocedural Slicing

The *slice* of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p. This concept, originally discussed by Mark Weiser in [34], allows one to isolate individual computation threads within a program. Slicing can be used for such diverse activities as helping a programmer understand complicated code, aiding debugging [21], automatically parallelizing programs [33,1], and automatically combining program variants [8]. In Weiser's terminology, a *slicing criterion* is a pair $<p, V>$, where p is a program point and V is a subset of the program's variables. In his work, a slice consists of all statements and predicates of the program that might affect the values of variables in V at point p. In many situations, this is a more general kind of slice than is needed: Rather than a slice taken with respect to program point p and an *arbitrary* set of variables, one is often interested in a slice taken with respect to variables that are *defined* or *used* at p. We will use the latter notion of slicing to establish that interprocedural slicing is log-space complete for $\mathcal{P}$. (The result holds for the more general kind of slice, as well.)

The value of a variable x defined at p is directly affected by the values of the variables used at p and by the loops and conditionals that enclose p. The value of a variable y *used* at p is directly affected by assignments to y that reach p and by the loops and conditionals that enclose p. In *intra*procedural slicing—the problem of slicing a program that consists of just a single monolithic procedure—a slice can be determined from the closure of the directly-affects relation.

Ottenstein and Ottenstein pointed out how well-suited *program dependence graphs* are for intraprocedural slicing [24]. Once a program is represented by its program dependence graph, the slicing problem is simply a reachability problem. Consequently, slices can be computed in time linear in the size of the dependence graph (for instance, by starting from the vertex that corresponds to the program point p of interest and performing a depth-first traversal over the dependence graph with all edges reversed in direction).

The problem of *inter*procedural slicing concerns how to generate a slice of a multi-procedure program, where the slice can cross the boundaries of procedure calls. One algorithm for interprocedural slicing was presented by Weiser [34]. However, as pointed out independently by Horwitz, Reps, and Binkley [9] and Hwang, Du, and Chou [10], Weiser's algorithm is *imprecise* in the sense that it can report "effects" that are transmitted (only) through paths in a graph representation of the program that do not represent feasible ("valid") execution paths. In general, the question of whether a given path is a possible execution path is undecidable, but certain paths can be identified as being invalid because they would correspond to execution paths with infeasible call/return linkages. For example, if procedure P calls Q twice—say at c_1 and c_2—one invalid path would start at the entry point of P, travel through P to c_1, enter Q, travel through Q to the return point, and return to P at c_2 (rather than c_1). Such paths fail to account correctly for the calling context (*e.g.*, c_1 in P) of a called procedure (*e.g.*, Q).

More recent algorithms for interprocedural slicing improve on Weiser's algorithm by only considering effects transmitted along interprocedurally valid paths [9,10,27].[3] In the remainder of the paper, when we use the term "interprocedural slicing" we will mean the problem of finding slices that are precise up to valid paths.

3.1. Valid Paths and System Dependence Graphs

The notion of a "valid path" is most easily understood in terms of the concept of a "system dependence graph", a graph used to represent multi-procedure programs ("systems") [9]. The system dependence graph is similar to other dependence-graph representations of programs (*e.g.*, [16,5]), but represents collections of procedures rather than just monolithic programs. An example program and its system dependence graph

[3]It has been shown that running time of the Horwitz-Reps-Binkley algorithm is polynomial in the size of the program [9]. An improved version of the algorithm (one with an asymptotically better running time) is given in [27]. The Hwang-Du-Chou algorithm, on the other hand, takes exponential time in the worst case [27]. That is, there is a family of examples on which the Hwang-Du-Chou algorithm uses time exponential in the size of the program being sliced.

are shown in Figure 1.

A program's *system dependence graph* (SDG) is a collection of procedure dependence graphs (PDGs), one for each procedure. Due to space limitations we will not give a detailed definition here; however, the important ideas should be clear from Figure 1 and the summary provided below. (For a detailed description of SDGs, see [9].)

The vertices of a procedure's PDG represent the individual statements and predicates of the procedure. A call statement is represented by several vertices: a call vertex and a collection of actual-in and actual-out vertices. There is an actual-in vertex for each actual parameter; there is an actual-out vertex for each actual parameter that might be modified during the call. Similarly, procedure entry is represented by an entry vertex and a collection of formal-in and formal-out vertices.

- An actual-in vertex represents the action taken by the calling procedure to pass in the actual parameter.
- A formal-in vertex represents the action taken by the called procedure to receive the parameter's value.
- A formal-out vertex represents the action taken by the called procedure to pass back the parameter's return value.
- An actual-out vertex represents the action taken by the calling procedure to receive the parameter's return value.

```
proc Main
   local a
   a := 1
   call P(a)
return

proc P(x: inout)
   if x > 0 then
      call Q(x)
   fi
   call Q(x)
return

proc Q(y: inout)
   y := y + 1
return
```

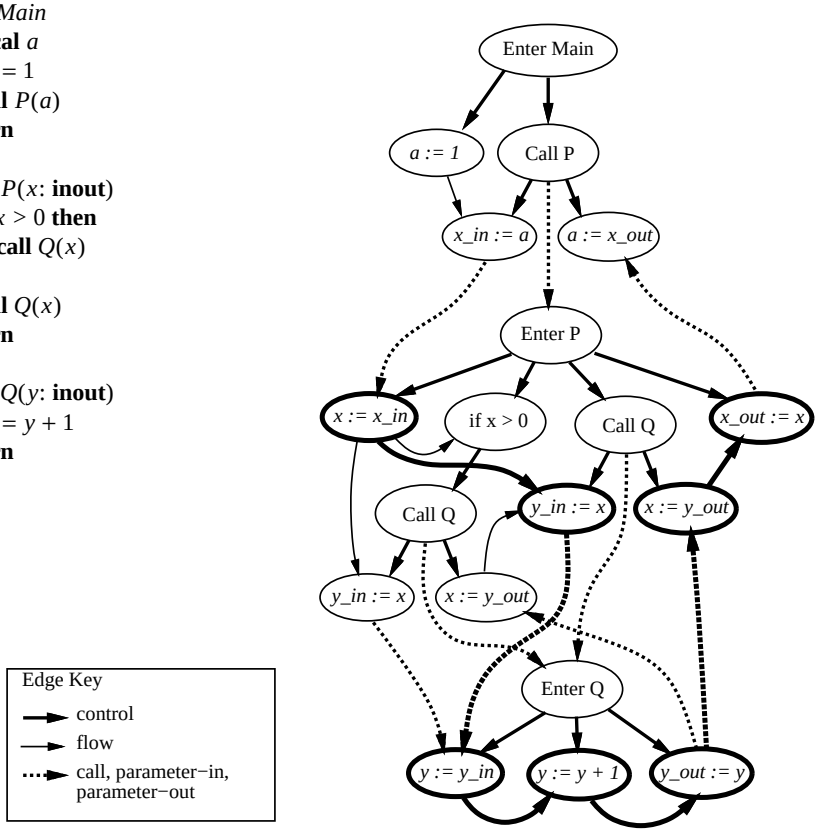


Figure 1. A program and its system dependence graph. The path shown in bold is a (same-level) valid path from the formal-in vertex $x := x\_in$ to the formal-out vertex $x\_out := x$ in procedure P.

Actual-in, actual-out, formal-in, and formal-out vertices will be referred to collectively as "parameter vertices". (Global variables are treated as "extra" parameters, and thus give rise to additional parameter vertices.) The edges of a PDG represent the control and flow dependences among the procedure's statements and predicates.[4]

The PDGs are connected together to form the SDG by *call* edges (which represent procedure calls, and run from a call vertex to an entry vertex) and by *parameter-in* and *parameter-out* edges (which represent parameter passing, and which run from an actual-in vertex to the corresponding formal-in vertex, and from a formal-out vertex to all corresponding actual-out vertices, respectively). These are shown as dashed lines in Figure 1.

A key fact about SDGs is that the SDG for a program can be constructed in time polynomial in the size of the program [9].

The notion of a valid path captures the idea that not every path through a system dependence graph represents a potentially valid execution path:

Definition 3.1. Let each call site in program P be given a unique index in the range [1 . . CallSites], where CallSites is the total number of call sites in the program. For each call site i, label the call-site's parameter-in edges and parameter-out edges by the symbols "$(_i$" and "$)_i$", respectively; label the call edge from call site i to the entry vertex for the called procedure with "$(_i$".

A ***same-level valid path*** in the system dependence graph for P is a path in the graph such that the sequence of symbols labeling the parameter-in and parameter-out edges in the path is a string in $L(matched)$, where $L(matched)$ is the language of balanced parentheses generated from nonterminal *matched* by the following context-free grammar:

$$\begin{array}{lll} matched & \rightarrow & matched \;\; matched \\ & | & (_i \;\; matched \;\;)_i \qquad\qquad \text{for } 1 \le i \le \text{CallSites} \\ & | & \varepsilon \end{array}$$

A ***valid path*** is a path in the system dependence graph for P such that the sequence of symbols labeling the parameter-in and parameter-out edges is a string in $L(valid)$, where $L(valid)$ is the language generated from nonterminal *valid* in the following grammar (where *matched* is as defined above):

$$\begin{array}{lll} unbalanced\_right & \rightarrow & unbalanced\_right \;\;)_i \;\; matched \qquad \text{for } 1 \le i \le \text{CallSites} \\ & | & matched \\ unbalanced\_left & \rightarrow & matched \;\; (_i \;\; unbalanced\_left \qquad \text{for } 1 \le i \le \text{CallSites} \\ & | & matched \\ valid & \rightarrow & unbalanced\_right \;\; unbalanced\_left \end{array}$$

□

A same-level valid path from v to w represents the transmission of an effect from v to w, where v and w are in the same procedure, via a sequence of execution steps during which the call stack can temporarily grow deeper—because of calls—but never shallower than its original depth before eventually returning to its original depth. An example of a same-level valid path is shown in bold in Figure 1. The "unbalanced_right" part of a valid path represents an execution sequence that may leave the call stack shallower than it was originally; the "unbalanced_left" part represents an execution sequence that may leave the call stack deeper than it was originally.

Definition 3.2. The ***(backward) slicing problem*** is that of finding, given a program P and a program point w in P, all program points v such that there is a valid path in P's SDG from the vertex that corresponds to v to the vertex that corresponds to w. □

---

[4]As defined in [9], procedure dependence graphs include four kinds of dependence edges: control, loop-independent flow, loop-carried flow, and def-order. However, for slicing the distinction between loop-independent and loop-carried flow edges is irrelevant, and def-order edges are not used. Therefore, in this paper we assume that PDGs include only control-dependence edges and a single kind of flow-dependence edge.

3.2. Interprocedural Slicing is Log-Space Complete for $\mathcal{P}$

We now show that the interprocedural-slicing problem is log-space complete for $\mathcal{P}$. More precisely, we show that when the interprocedural-slicing problem is recast as a decision problem, the problem is log-space complete for $\mathcal{P}$. The decision-problem version answers questions of the form: "Is program point v in the slice of program P with respect to program point w?" (or, equivalently, "Is there a valid path in P's SDG from v to w?")

Focusing on the decision-problem version of the interprocedural-slicing problem entails no significant loss of generality. The main reason for being interested in whether interprocedural slicing is log-space complete for $\mathcal{P}$ is to understand whether it might be possible to devise an $\mathcal{NC}$ parallel algorithm for the problem—that is, whether there is an algorithm for the problem that has polylogarithmic running time using a polynomial number of processors. Because for a given program there are only a polynomial number of questions of the form "Is vertex v in the slice of the program with respect to program point p?" the existence of an $\mathcal{NC}$ algorithm for the decision problem would mean that it would be possible to have a fast parallel algorithm to find *all* the elements of the slice (*i.e.*, in polylogarithmic time and with a polynomial number of processors).

The proof that interprocedural slicing is log-space complete for $\mathcal{P}$ is by a log-space reduction from a known $\mathcal{P}$-complete problem, the monotone circuit value problem [6], which is defined as follows:

Definition 3.3. A ***monotone circuit*** is a directed acyclic graph with five different kinds of nodes (called ***gates***):

- ***input gates*** have no in-edges and one out-edge.
- ***and gates*** have two in-edges and one out-edge.
- ***or gates*** have two in-edges and one out-edge.
- ***fan-out gates*** have one in-edge and two out-edges.
- there is a single ***output gate***, which has one in-edge and no out-edges.

In addition, all gates must be reachable from an input gate, and the output gate must be reachable from all gates.

Every assignment a of truth values to the input gates of the circuit can be extended uniquely to a truth-value assignment $\bar{a}$ on all gates as follows:

- If n is an input gate, then $\bar{a}(n) = a(n)$.
- If n is an and gate with predecessors n' and n'', then $\bar{a}(n) = \bar{a}(n') \wedge \bar{a}(n'')$.
- If n is an or gate with predecessors n' and n'', then $\bar{a}(n) = \bar{a}(n') \vee \bar{a}(n'')$.
- If n is a fan-out gate with predecessor n', then $\bar{a}(n) = \bar{a}(n')$.
- If n is the output gate with predecessor n', then $\bar{a}(n) = \bar{a}(n')$.

The ***monotone circuit value problem*** is that of deciding, given a monotone circuit and a truth-value assignment a for the circuit's input gates, the value of $\bar{a}(n)$ for the circuit's output gate n. □

Theorem 3.4. *The problem of interprocedural slicing is log-space complete for $\mathcal{P}$.*

Proof. It is known that interprocedural slicing using SDGs can be performed in polynomial time [9,27].[5]

The proof that interprocedural slicing is $\mathcal{P}$-hard is by reduction from the monotone circuit value problem via a log-space Turing machine program.[6] Given a circuit C and a truth-value assignment a on the input tape, the construction described below writes out a program P_C that has the following property: P_C has two

---

[5] In particular, it was shown that the running time of a certain algorithm for finding slices of an SDG is bounded by $O((\mathrm{P} \times \mathrm{E} \times (\mathrm{Globals} + \mathrm{Params})) + (\mathrm{TotalSites} \times (\mathrm{Globals} + \mathrm{Params})^3))$, where P is the number of procedures in the program, E is the maximum number of control and flow edges in any procedure's PDG, TotalSites is the total number of call sites in the program, Params is the largest number of formal parameters in any procedure, and Globals is the number of global variables in the program [27]. This result holds even when the program being sliced uses recursive—or mutually recursive—procedures.

[6] A log-space Turing machine has a read-only input tape, a read-write work tape with $O(\log n)$ cells, where n is the size of the input, and a write-only output tape.

distinguished statements in procedure *Main*—$x := 0$ and $y := x$—such that $x := 0$ is in the (same-level) slice of P_C with respect to $y := x$ iff the output gate of C has value **true**. That is, there is a (same-level) valid path from $x := 0$ to $y := x$ iff the output gate of C has value **true**.

For each of the five gate types, there is a different kind of "gadget"; each gadget is a schema for a simple procedure. The gadgets used in the construction—and their corresponding dependence graphs—are illustrated in Figure 2. Each instance of a gate in C is translated to a procedure of the appropriate kind. Gate names are used to fill in the subscripts on procedure names as gadget instances (*i.e.*, procedures) are written out to the output tape.

The work tape comes into play in translating input gates. Whenever an input gate is encountered in the circuit, the work tape is used to hold the gate name n together with an index into the input tape that records the position of the gate in the circuit. The input-tape head is then used to locate the truth assignment for n—which is to be found in the portion of the input tape that contains a—and an appropriate procedure is written out for n. Finally, the index recorded on the work tape is used to resume the translation process at the appropriate place in the circuit description on the input tape. Because a gate name and an index position each take no more than $O(\log x)$ bits, where x is the size of the input, the construction meets the log-space restriction.

It can be shown by induction on the height of circuit C that for each gate n in C (with corresponding procedure P_n), there is a (same-level) valid path in the program's system dependence graph from the formal-in vertex $x := x\_in$ in P_n to the formal-out vertex $x := x\_out$ in P_n iff $\bar{a}(n) =$ **true**. Consequently, the output gate of circuit C has value **true** iff the statement $x := 0$ in procedure *Main* is in the (same-level) slice of the program with respect to statement $y := x$ in procedure *Main*. □

In the above construction, it really makes no difference whether we think of the Turing machine as writing out a program P_C (using the textual representation of gadgets shown in column 3 of Figure 2) or the system dependence graph of P_C (using the graph representation of gadgets shown in column 2.)

Example. Figure 3 shows an example circuit, together with the system dependence graph for the program created via the construction described in the proof of Theorem 3.4. The circuit evaluates to **true**, and one of the (same-level) valid paths from $x := 0$ in *Main* to $y := x$ in *Main* is outlined in bold. □

It is interesting to note that the construction used in Theorem 3.4 uses only "interprocedural straight-line code". That is, it uses only straight-line code and procedure calls, but no conditional statements (nor loops, nor recursion).

3.3. $\mathcal{P}$-Complete Attribute-Grammar Analysis Problems

Horwitz, Reps, and Binkley pointed out a correspondence between the call structure of a program and a context-free grammar, and between the *intra*procedural transitive dependences among a PDG's parameter vertices and the dependences among attributes in an attribute grammar [9]. Because of this correspondence, Theorem 3.4 also has consequences for the computation of IO graphs of attribute grammars [13,3,22] and other similar approximations to the characteristic graphs of an attribute grammar's nonterminals that can be computed in polynomial time, such as TDS graphs of ordered attribute grammars [12].

Theorem 3.5. *The problem of deciding whether an edge occurs in the IO graph of a nonterminal of an attribute grammar is log-space complete for* $\mathcal{P}$.

Proof. The IO graphs of an attribute grammar's nonterminals can be computed in polynomial time (by means of a "bottom-up" iterative algorithm) [3, pp. 16].

The proof that the problem is $\mathcal{P}$-hard is again by reduction from the monotone circuit value problem. Given a circuit and a truth-value assignment a on the input tape, a log-space Turing machine can write out on the output tape an attribute grammar that has the following property: An edge $(Start.v, Start.w)$ exists in the start symbol's IO graph iff the circuit has value **true** under the given truth-value assignment.

The circuit is transformed into an attribute grammar using the "gadgets" shown in Figure 4 as schemas for the grammar's productions. Each instance of a gate in the circuit is translated to a production of the appropriate kind. Gate names are used to fill in the subscripts on nonterminal names as gadget instances (*i.e.*, productions) are written to the output tape. (As in Theorem 3.4, the work tape comes into play to find

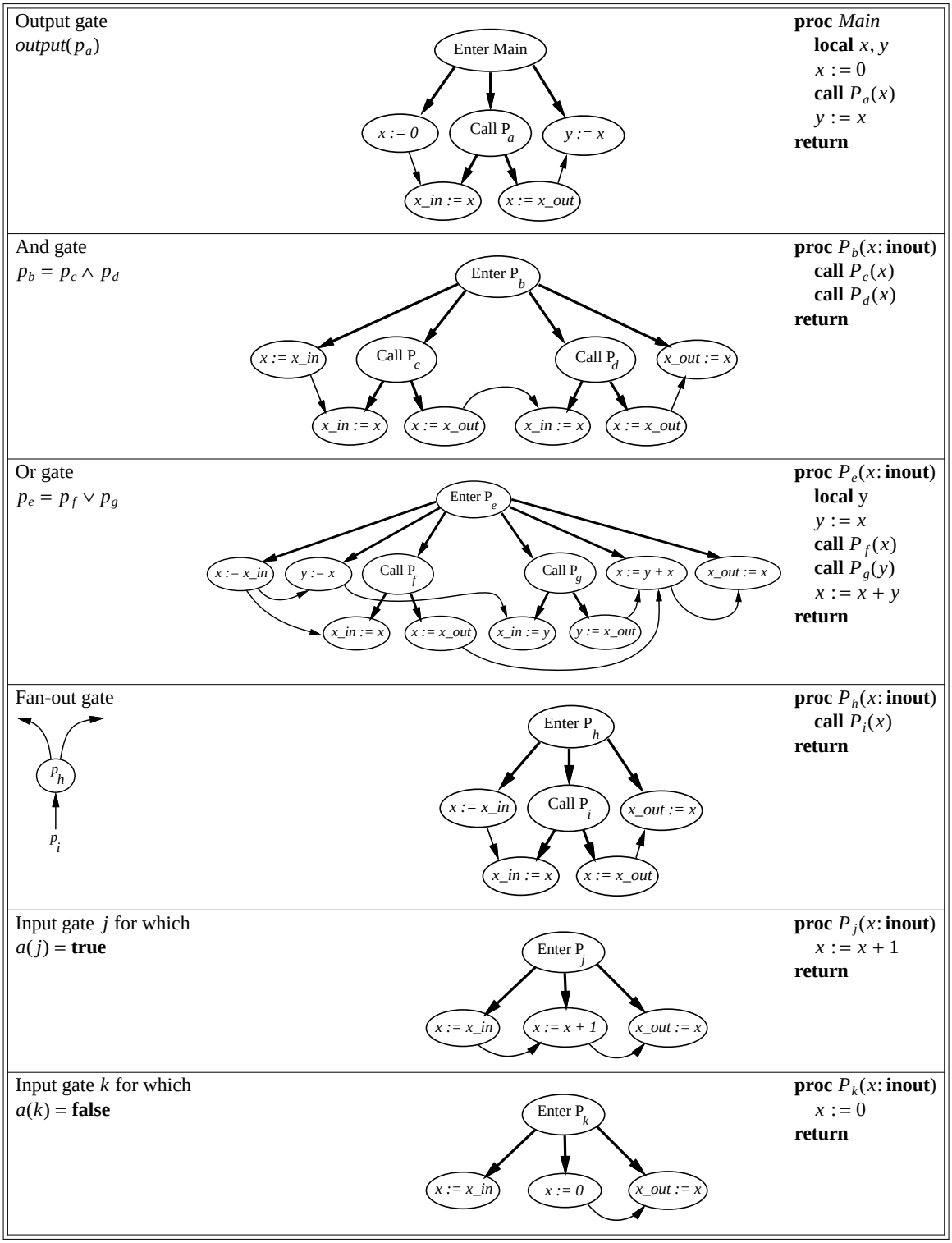


Figure 2. Gadgets used in the construction of a program in which a particular (same-level) slice simulates a given monotone Boolean circuit under a given assignment of truth values to input gates. The output gate of the circuit has value **true** iff the statement $x := 0$ in procedure *Main* is in the (same-level) slice of the program with respect to statement $y := x$ in procedure *Main*.

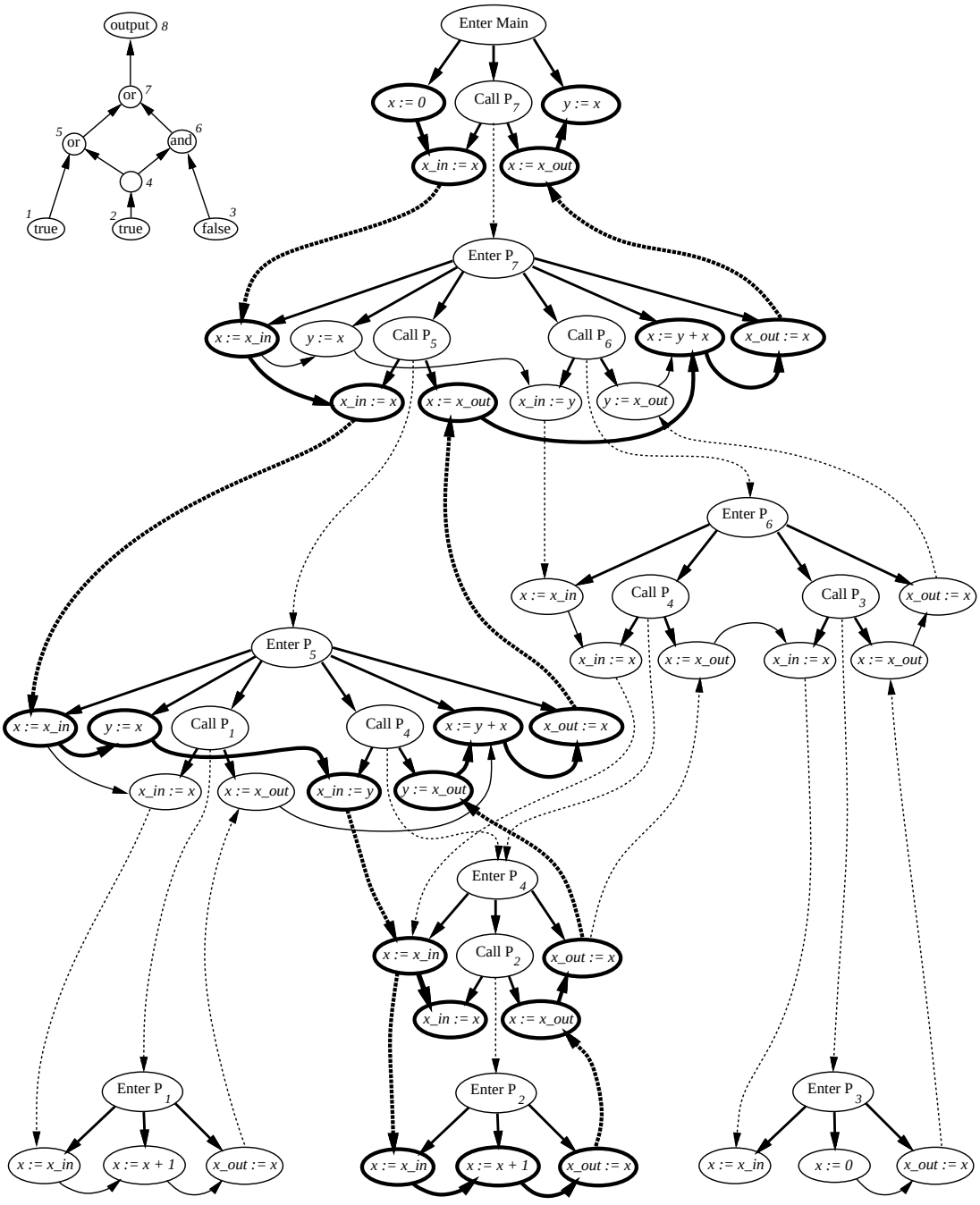


Figure 3. An example circuit (upper-left-hand corner), together with the system dependence graph for the program created via the construction used the proof of Theorem 3.4. The circuit evaluates to **true**, and one of the same-level valid paths from $x := 0$ in *Main* to $y := x$ in *Main* is outlined in bold.

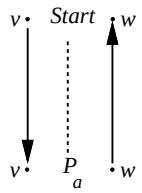
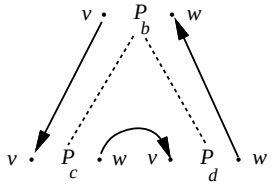
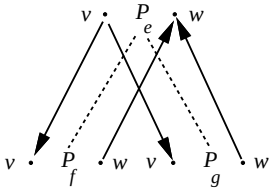
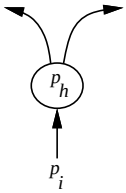
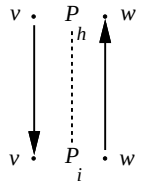
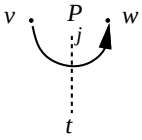
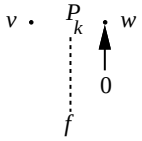
Output gate $output(p_a)$		$Start \rightarrow P_a$ Equations: $P_a.v = Start.v$ $Start.w = P_a.w$
And gate $p_b = p_c \wedge p_d$		$P_b \rightarrow P_c\ P_d$ Equations: $P_c.v = P_b.v$ $P_d.v = P_c.w$ $P_b.w = P_d.w$
Or gate $p_e = p_f \vee p_g$		$P_e \rightarrow P_f\ P_g$ Equations: $P_f.v = P_e.v$ $P_g.v = P_e.v$ $P_e.w = P_f.w + P_g.w$
Fan-out gate 		$P_h \rightarrow P_i$ Equations: $P_i.v = P_h.v$ $P_h.w = P_i.w$
Input gate j for which $a(j) =$ **true**		$P_j \rightarrow t$ Equations: $P_j.w = P_j.v$
Input gate k for which $a(k) =$ **false**		$P_k \rightarrow f$ Equations: $P_k.w = 0$

Figure 4. Gadgets used in the construction of an attribute grammar for which an edge $(Start.v, Start.w)$ exists in the start symbol's IO graph iff a given monotone Boolean circuit has value **true** under a given assignment of truth values to input gates.

a truth assignment in a when an input gate is translated.)

Let nonterminal P_n be the nonterminal of the grammar that corresponds to gate n. It is an easy matter to show by induction on the height of the circuit that there is an edge $(P_n.v, P_n.w)$ in the IO graph for nonterminal P_n iff $\bar{a}(n) =$ **true**. Consequently, an edge $(Start.v, Start.w)$ exists in the start symbol's IO graph iff the circuit has value **true** under the given truth-value assignment. □

4. $\mathcal{P}$-Hardness of Interprocedural Dataflow Analysis

In this section, we show that a large class of interprocedural dataflow-analysis problems are $\mathcal{P}$-hard. The problems considered are those amenable to Sharir and Pnueli's "functional approach" to interprocedural dataflow analysis.[7] This generalizes Kildall's concept of the "meet-over-all-paths" solution of an *intra*procedural dataflow-analysis problem [14] to the "meet-over-all-valid-paths" solution of an *inter*procedural dataflow-analysis problem [32].

This framework for interprocedural dataflow analysis is reviewed below. We then show that the decision-problem versions of the dataflow-analysis problems in this framework are $\mathcal{P}$-hard. (In this form, the problems of the framework answer questions of the form "Does dataflow value l approximate y_n, where $y_n \in L$ is the dataflow value for program point n in the meet-over-all-valid-paths solution?" As in the previous section, focusing on decision problems entails no significant loss of generality.)

Throughout this section, we assume that $(L, \sqcap)$ is a meet semi-lattice of dataflow values with a smallest element $\bot$ and a largest element $\top$. We also assume that dataflow functions are members of a space of distributive functions $F \subseteq L \to L$ and that F contains the identity function.

A program is represented using a directed graph $G^* = (N^*, E^*)$ called a ***supergraph***. G^* consists of a collection of flowgraphs $G_1, G_2, \cdots$ (one for each procedure), one of which, G_{main}, represents the program's main procedure. Each flowgraph G_p has a unique ***start*** node s_p, and a unique ***exit*** node e_p. The other nodes of the flowgraph represent statements and predicates of the program in the usual way, except that a procedure call is represented by two nodes, a ***call*** node and a ***return-site*** node.

In addition to the ordinary intraprocedural edges that connect the nodes of the individual flowgraphs, for each procedure call, represented by call-node c and return-site node r, G^* has three edges:

- An intraprocedural ***call-to-return-site*** edge from c to r;
- An interprocedural ***call-to-start*** edge from c to the start node of the called procedure;
- An interprocedural ***exit-to-return-site*** edge from the exit node of the called procedure to r.

Without loss of generality, we assume that the only edges into start nodes are call-to-start edges. (The call-to-return-site edges are included so that we can handle programs with local variables and parameters; the dataflow functions on call-to-return-site and exit-to-return-site edges permit the information about local variables that holds at the call site to be combined with the information about global variables that holds at the end of the called procedure.)

The notion of interprocedurally valid paths captures the idea that not all paths through G^* represent potentially valid execution paths:

Definition 4.1. For each $n \in N$, IVP(s_{main}, n) denotes the set of all ***interprocedurally valid paths*** in G^* that lead from s_{main} to n. A path $q \in \text{path}_{G^*}(s_{main}, n)$ is in IVP(s_{main}, n) iff the sequence of symbols labeling the parameter-in and parameter-out edges is a string in $L(unbalanced\_left)$, where $L(unbalanced\_left)$ is the language generated from nonterminal *unbalanced_left* in the grammar given in Definition 3.1. □

Note that Definition 4.1 involves the language $L(unbalanced\_left)$ and not $L(valid)$, as used in Definition 3.1. The reason for this is that in the solution to a dataflow-analysis problem, we are only concerned with paths that begin at s_{main}, the start node of the program's main procedure:

Definition 4.2. If q is a path in G^*, let pf_q denote the (path) function obtained by composing the functions associated with q's edges (in the order that they appear in path q). The ***meet-over-all-valid-paths*** solution to the dataflow problem consists of the collection of values y_n defined by the following set of equations:

$$\Phi_n = \bigsqcap_{q \in \text{IVP}(s_{main}, n)} pf_q \qquad \text{for each } n \in N^*$$

$$y_n = \Phi_n(\bot) \qquad \text{for each } n \in N^*$$

□

---

[7] In order to handle programs in which recursive procedures have local variables and parameters, we actually use a slightly generalized version of the Sharir-Pnueli framework that was introduced in [29].

Theorem 4.3. *The interprocedural dataflow-analysis problems are $\mathcal{P}$-hard under log-space reductions.*

Proof. Suppose problem X is any interprocedural dataflow-analysis problem that meets the conditions of the dataflow framework. We make the reasonable assumption that there is a dataflow value that can be generated in the main procedure and "killed" in other procedures of the program. Any instance of the monotone circuit value problem can be transformed (via a log-space transformation) into an instance of X.

The construction is very similar to the one used in Theorem 3.4: the idea is for a distinguished dataflow fact to "flow" along a (same-level) valid path from *Main* to *Main* iff the value of the circuit's output gate is **true**. The gadgets used in the construction for the interprocedural reaching-definitions problem—and their corresponding flow graphs—are illustrated in Figure 5. In these gadgets, the edge functions on call-to-return-site edges are all $\lambda x.\top$, and, with two exceptions, the edge functions on ordinary control-flow-graph edges are all the identity function. The edge functions that are not the identity function are (1) in procedure *Main*, where the distinguished dataflow fact is introduced, and (2) on certain edges in procedures that correspond to input gates of the circuit with value **false**, where the distinguished dataflow fact is killed.

For instance, in Figure 5 the distinguished dataflow fact is "x is defined at statement $x := 0$ of *Main*". This is introduced by the edge function on the edge leading from $x := 0$ in *Main*. The presence of the assignment statement $x := x + 1$ in each procedure that corresponds to an input gate with value **false** causes the definition of x in *Main* to be killed (via the function on the edge leading from $x := x + 1$) on any valid path that passes through the procedure.

It can be shown by induction on the height of circuit C that for each gate n in C (with corresponding procedure P_n), there is a (same-level) valid path along which the distinguished dataflow fact is *not* killed iff $\bar{a}(n) =$ **true**. Consequently, the output gate of circuit C has value **true** iff the distinguished dataflow fact is in the fact set for the Exit vertex of procedure *Main*. □

Instances of the dataflow framework that are solvable in polynomial time, such as interprocedural reaching definitions, interprocedural available expressions, and interprocedural live variables, are consequently log-space complete for $\mathcal{P}$. In fact, this is the case for all of the so-called "interprocedural, finite, distributive, subset problems" (or IFDS problems, for short)—distributive dataflow-analysis problems in which the domain of dataflow values for a program being analyzed consists of all subsets of a finite set D, where the size of D is polynomial in the size of the program [29]. Furthermore, it still holds even if we restrict our attention to just the so-called "gen/kill" problems—problems in which every edge function f_e is of the form $\lambda x.(x - kill_e) \cup gen_e$, where $kill_e$ and gen_e are set-valued constants associated with edge e. (Again, we are assuming that the domain of dataflow values for a program being analyzed consists of all subsets of a finite set D, where the size of D is polynomial in the size of the program.)

Corollary 4.4. *Every gen/kill or IFDS problem for which there is a dataflow value that can be generated in the main procedure and killed in other procedures of the program is log-space complete for $\mathcal{P}$.*

Proof. Reps, Horwitz, and Sagiv have given a polynomial-time algorithm for the IFDS problems [29]. The (finite-subset) gen/kill problems are a subclass of the IFDS problems and hence are also solvable in polynomial time. $\mathcal{P}$-completeness follows from Theorem 4.3. □

Note that the construction used in the proof of Theorem 4.3 does not carry over to the case of *intra*procedural dataflow analysis: the transformation presented in Theorem 4.3 uses multiple calls and (same-level) valid paths to simulate shared subexpressions that occur in the circuit. In fact, for a large class of intraprocedural dataflow-analysis problems, the situation is quite the opposite: they *are* solvable by $\mathcal{NC}$-class parallel algorithms.

Theorem 4.5. *Every intraprocedural, finite, distributive, subset problem is solvable by an $\mathcal{NC}$-class parallel algorithm.*

Proof. Reps, Horwitz, and Sagiv have shown how all intraprocedural finite, distributive, subset problems can be converted to graph-reachability problems (where the size of the graph is polynomial in the size of the program) [29]. Reachability is known to be in $\mathcal{NC}$. (See, for example, [25, pp. 362].) □

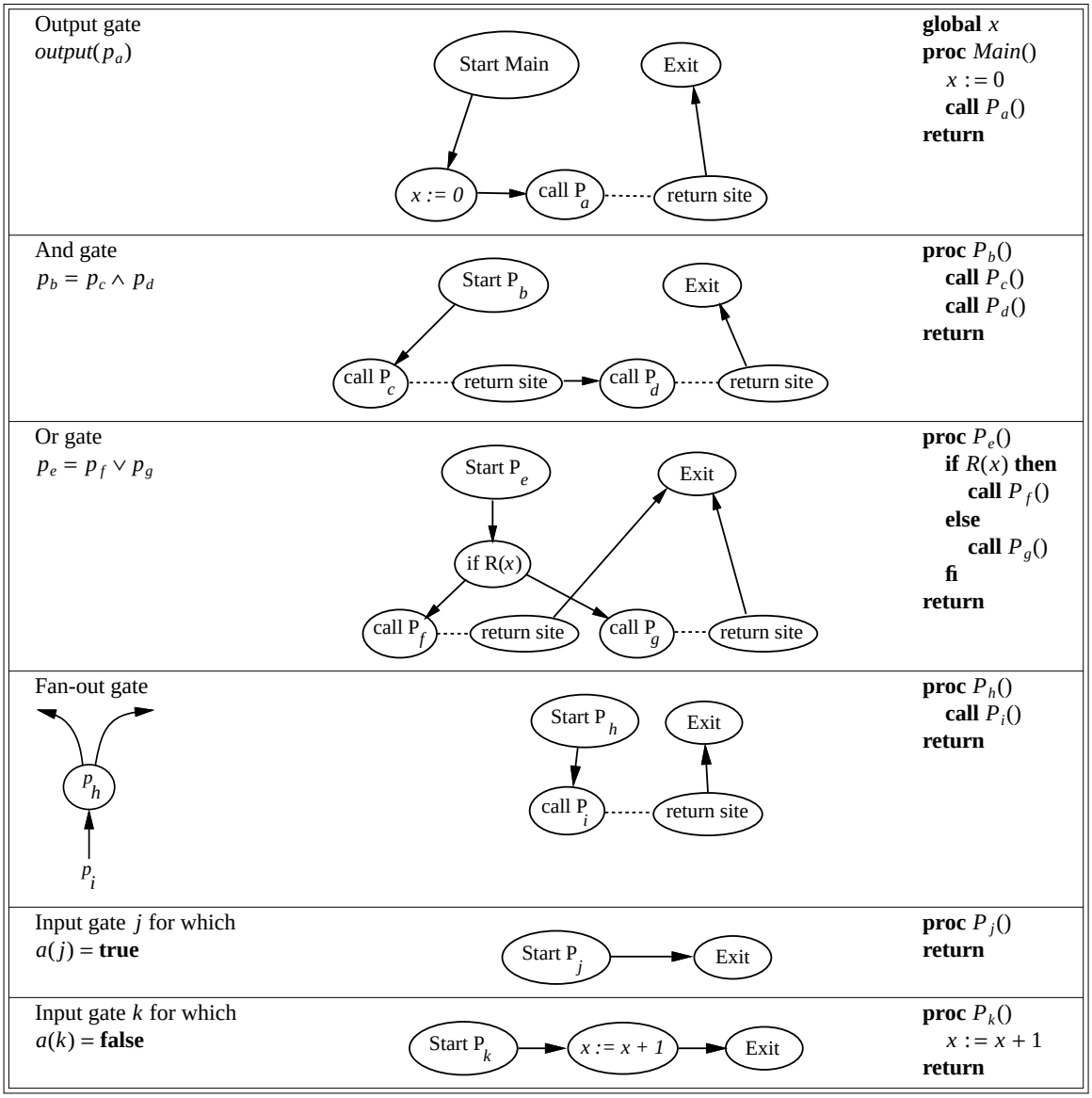


Figure 5. Gadgets used in the construction of a program in which a particular reaching-definitions problem simulates a given monotone Boolean circuit under a given assignment of truth values to input gates. Ordinary control-flow-graph edges are indicated with solid directed arrows. Call-to-return-site edges, which connect call sites with their corresponding return sites, are shown with dashed lines. The output gate of the circuit has value **true** iff the assignment to variable x in procedure *Main* reaches the Exit vertex of procedure *Main*.

5. Relation to Previous Work

This paper has investigated the computational complexity of two interprocedural program-analysis problems under the assumption that only valid paths are to be considered. Our results provide evidence that there do not exist fast ($\mathcal{NC}$-class) parallel algorithms for interprocedural slicing and interprocedural dataflow analysis. In other words, this work suggests that there are limitations on the ability to use parallelism to

overcome compiler bottlenecks due to expensive interprocedural-analysis computations.

In terms of the impact on the programming-languages area, our results can be viewed as being complementary to the results of Dwork, Kanellakis, and Mitchell, who showed that there are limitations on the ability to use parallelism to speed up unification [4]. The Dwork-Kanellakis-Mitchell result demonstrates that there are limitations on the use of parallelism to speed up the *execution* of programs (specifically, programs written in languages such as Prolog that use unification for parameter passing). Our results demonstrate that there are similar limitations on the use of parallelism to speed up the *static analysis* of programs. (However, it may well be possible to use parallelism to achieve useful speedups on the kinds of static-analysis problems that actually arise in practice—*cf.* Robinson's comments on the Dwork-Kanellakis-Mitchell result [30].)

The gadgets used in the constructions in Sections 3 and 4 have some similarities to the ones used in the proofs that the unification problem [4] and the left-linear semi-unification problem [7] are log-space complete for $\mathcal{P}$.

Landi and Ryder have also investigated the computational complexity of interprocedural dataflow analysis under the assumption that only valid paths are to be considered [17]. (Valid paths are called "realizable" paths in [17].) Their work shows that when the program-analysis problem to be solved involves certain kinds of constructs (*e.g.*, single or multiple levels of pointers, reference parameters, *etc.*) one faces certain kinds of computational limitations (*e.g.*, $\mathcal{NP}$-hardness, undecidability, *etc.*).[8]

This paper demonstrates that, *by itself*, the aspect of considering only *valid paths* in an interprocedural-analysis problem imposes some inherent computational limitations, namely that the problem is unlikely to have a fast ($\mathcal{NC}$-class) parallel algorithm. For example, in the case of interprocedural dataflow analysis, Theorem 4.3 shows that the problem of finding solutions that are precise up to valid paths is log-space complete for $\mathcal{P}$. On the other hand, dropping the restriction that only valid paths are to be considered leads to less precise, but safe, solutions. Such solutions can be obtained by treating an *inter*procedural dataflow-analysis problem as one large *intra*procedural problem on supergraph G^*. Consequently, for the imprecise version of interprocedural dataflow analysis there *does* exist a fast ($\mathcal{NC}$-class) parallel algorithm (see Theorem 4.5).

Acknowledgement

Fritz Henglein suggested that the interprocedural-slicing problem was likely to be log-space complete for $\mathcal{P}$.

References

1. Badger, L. and Weiser, M., "Minimizing communication for synchronizing parallel dataflow programs," in *Proceedings of the 1988 International Conference on Parallel Processing*, (St. Charles, IL, Aug. 15-19, 1988), Pennsylvania State University Press, University Park, PA (1988).
2. Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 47-56 (July 1988).
3. Deransart, P., Jourdan, M., and Lorho, B., *Attribute Grammars: Definitions, Systems and Bibliography, Lecture Notes in Computer Science*, Vol. 323, Springer-Verlag, New York, NY (1988).
4. Dwork, C., Kanellakis, P.C., and Mitchell, J.C., "On the sequential nature of unification," *Journal of Logic Programming* **1** pp. 35-50 (1984).
5. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.* **9**(3) pp. 319-349 (July 1987).
6. Goldschlager, L., "The monotone and planar circuit value problems are log-space complete for P," *ACM SIGACT News* **9**(2) pp. 25-29 (1977).
7. Henglein, F., "Fast left-linear semi-unification," pp. 82-91 in *Proceedings of the International Conference on Computing and Information*, (May 1990), *Lecture Notes in Computer Science*, Vol. 468, Springer-Verlag, New York, NY (1990).
8. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

[8] Some program-analysis problems are undecidable even in their *intra*procedural version [20,26].

9. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).
10. Hwang, J.C., Du, M.W., and Chou, C.R., "Finding program slices for recursive procedures," in *Proceedings of IEEE COMPSAC 88*, (Chicago, IL, Oct. 3-7, 1988), IEEE Computer Society, Washington, DC (1988).
11. Jones, N.D. and Laaser, W.T., "Complete problems for deterministic polynomial time," *Theoretical Computer Science* **3** pp. 105-117 (1977).
12. Kastens, U., "Ordered attribute grammars," *Acta Informatica* **13**(3) pp. 229-256 (1980).
13. Kennedy, K. and Warren, S.K., "Automatic generation of efficient evaluators for attribute grammars," pp. 32-49 in *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, (Atlanta, GA, Jan. 19-21, 1976), ACM, New York, NY (1976).
14. Kildall, G., "A unified approach to global program optimization," pp. 194-206 in *Conference Record of the First ACM Symposium on Principles of Programming Languages*, ACM, New York, NY (1973).
15. Knoop, J. and Steffen, B., "The interprocedural coincidence theorem," pp. 125-140 in *Proceedings of the Fourth International Conference on Compiler Construction*, (Paderborn, FRG, October 5-7, 1992), *Lecture Notes in Computer Science*, Vol. 641, ed. U. Kastens and P. Pfahler, Springer-Verlag, New York, NY (1992).
16. Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Dependence graphs and compiler optimizations," pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).
17. Landi, W. and Ryder, B.G., "Pointer-induced aliasing: A problem classification," pp. 93-103 in *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, (Orlando, FL, January 1991), ACM, New York, NY (1991).
18. Landi, W. and Ryder, B.G., "A safe approximate algorithm for interprocedural pointer aliasing," *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, (San Francisco, CA, June 17-19, 1992), *ACM SIGPLAN Notices* **27**(7) pp. 235-248 (July 1992).
19. Landi, W., Ryder, B.G., and Zhang, S., "Interprocedural modification side effect analysis with pointer aliasing," pp. 56-67 in *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation*, (Albuquerque, NM, June 23-25, 1993), ACM, New York, NY (June 1993).
20. Landi, W., "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems* **1**(4) p. December 1993 ().
21. Lyle, J. and Weiser, M., "Experiments on slicing-based debugging tools," in *Proceedings of the First Conference on Empirical Studies of Programming*, (June 1986), Ablex Publishing Co. (1986).
22. Möncke, U. and Wilhelm, R., "Grammar flow analysis," pp. 151-186 in *Attribute Grammars, Applications and Systems*, (International Summer School SAGA, Prague, Czechoslovakia, June 1991), *Lecture Notes in Computer Science*, Vol. 545, ed. H. Alblas and B. Melichar, Springer-Verlag, New York, NY (1991).
23. Myers, E., "A precise inter-procedural data flow algorithm," pp. 219-230 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).
24. Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).
25. Papadimitriou, C.H., *Computational Complexity*, Addison-Wesley, Reading, MA (1994).
26. Ramalingam, G., "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.* **16**(5) pp. 1476-1471 (September 1994).
27. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," *SIGSOFT 94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (New Orleans, LA, December 7-9, 1994), *ACM SIGSOFT Software Engineering Notes* **19**(5) pp. 11-20 (December 1994).
28. Reps, T., "Demand interprocedural program analysis using logic databases," pp. 163-196 in *Applications of Logic Databases*, ed. R. Ramakrishnan, Kluwer Academic Publishers, Boston, MA (1994).
29. Reps, T., Horwitz, S., and Sagiv, M., "Precise interprocedural dataflow analysis via graph reachability," pp. 49-61 in *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).
30. Robinson, J.A., "Logic and logic programming," *Commun. of the ACM* **35**(3) pp. 40-65 (March 1992).
31. Sagiv, M., Reps, T., and Horwitz, S., "Precise interprocedural dataflow analysis with applications to constant propagation," pp. 651-665 in *Proceedings of FASE 95: Colloquium on Formal Approaches in Software Engineering*, (Aarhus, Denmark, May 22-26, 1995), *Lecture Notes in Computer Science*, Vol. 915, ed. P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, Springer-Verlag, New York, NY (1995).
32. Sharir, M. and Pnueli, A., "Two approaches to interprocedural data flow analysis," pp. 189-233 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones, Prentice-Hall, Englewood Cliffs, NJ (1981).
33. Weiser, M., "Reconstructing sequential behavior from parallel behavior projections," *Information Processing Letters* **17** pp. 129-135 (October 1983).
34. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).