On the Sequential Nature of Interprocedural Program-Analysis Problems

Thomas Reps
University of Copenhagen[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 dataflow analysis problems is $\mathcal{P}$-hard.
- Obtaining "meet-over-all-valid-paths" solutions to interprocedural gen/kill dataflow analysis 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 interprocedural dataflow analysis (unless $\mathcal{P} \subseteq \mathcal{NC}$). That is, it is unlikely that there are algorithms for interprocedural slicing and 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 most 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] and in "flow-sensitive" interprocedural dataflow analysis problems [22,19,2,16,15]. In particular, Sharir and Pnueli have 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 [22].

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 dataflow analysis problems is $\mathcal{P}$-hard.
- Obtaining "meet-over-all-valid-paths" solutions to interprocedural gen/kill dataflow analysis problems is log-space complete for $\mathcal{P}$.

The practical implications of these results are that there do not exist fast ($\mathcal{NC}$-class) parallel algorithms for interprocedural slicing and interprocedural dataflow analysis (unless $\mathcal{P} \subseteq \mathcal{NC}$). That is, it is unlikely that

---

[1]On sabbatical leave from the University of Wisconsin−Madison.

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: Datalogisk Institut, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark.
Electronic mail: reps@diku.dk.

there are algorithms for interprocedural slicing and 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

A problem that is log-space complete for $\mathcal{P}$ (or "$\mathcal{P}$-complete under log-space reductions") 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]. Furthermore, if any problem that is log-space complete for $\mathcal{P}$ has an $\mathcal{NC}$ parallel algorithm (*i.e.* a parallel algorithm in 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) then every language in $\mathcal{P}$ has an $\mathcal{NC}$ parallel algorithm. That is, the problems that are log-space complete for $\mathcal{P}$ do not have $\mathcal{NC}$ parallel algorithms unless $\mathcal{P} \subseteq \mathcal{NC}$. This 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. 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 gen/kill dataflow analysis problems 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).

Following Sharir and Pnueli [22] and Horwitz, Reps, and Binkley [9], we assume that the programs being analyzed do not contain aliasing and that they do not use procedure-valued variables. We also make some simplifying assumptions about global variables. In Section 3, we follow Horwitz, Reps, and Binkley and assume that there are no global variables but that procedures have value-result parameters. In Section 4, we make the opposite assumptions: we follow Sharir and Pnueli and assume that all variables are global variables and that procedures are parameterless.

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]

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 [24], 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 [17], automatically parallelizing programs [23,1], 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. This is a more general kind of slice than is often needed: Rather than a slice taken with respect to program-point p and an *arbitrary* variable, one is often interested in a slice taken with respect to a variable x that is *defined* or *used* at p. 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.

[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 the inherent computational limitations they introduce [16].

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 this kind of slicing [20]. Once a program is represented by its program dependence graph, the slicing problem is simply a vertex-reachability problem, and thus 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 reversal of the dependence graph).

This section concerns the problem of *inter*procedural slicing—generating a slice of an entire program, where the slice crosses the boundaries of procedure calls.

3.1. The Calling-Context Problem and Valid Paths

One algorithm for interprocedural slicing was presented by Weiser [24]. However, as pointed out independently by Horwitz, Reps, and Binkley [7,9] 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 *Main* calls P twice—say at c_1 and c_2—one invalid path would start at the entry point of *Main*, travel through *Main* to c_1, enter P, travel through P to the return point, and return to *Main* at c_2 (rather than c_1). Such paths fail to account correctly for the calling context (*e.g.*, c_1 in *Main*) of a called procedure (*e.g.*, P).

Weiser describes his method for interprocedural slicing as follows [24]:

> For each criterion C for a procedure P, there is a set of criteria $\mathrm{UP}_0(C)$ which are those needed to slice callers of P, and a set of criteria $\mathrm{DOWN}_0(C)$ which are those needed to slice procedures called by P. . . . $\mathrm{UP}_0(C)$ and $\mathrm{DOWN}_0(C)$ can be extended to functions UP and DOWN which map sets of criteria into sets of criteria. Let CC be any set of criteria. Then
>
> $$\mathrm{UP}(CC) = \bigcup_{C \in CC} \mathrm{UP}_0(C)$$
>
> $$\mathrm{DOWN}(CC) = \bigcup_{C \in CC} \mathrm{DOWN}_0(C)$$
>
> The union and transitive closure of UP and DOWN are defined in the usual way for relations. $(\mathrm{UP} \cup \mathrm{DOWN})^*$ will map any set of criteria into all those criteria necessary to complete the corresponding slices through all calling and called routines. The complete interprocedural slice for a criterion C is then just the union of the intraprocedural slices for each criterion in $(\mathrm{UP} \cup \mathrm{DOWN})^*(C)$.

The reason this method does not produce as precise a slice as possible is that transitive closure fails to account for the calling context of a called procedure. For example, the relation $(\mathrm{UP} \cup \mathrm{DOWN})^*(\langle p, V \rangle)$ includes the relation $\mathrm{UP}(\mathrm{DOWN}(\langle p, V \rangle))$. $\mathrm{UP}(\mathrm{DOWN}(\langle p, V \rangle))$ includes all call sites that call procedures containing the program points in $\mathrm{DOWN}(\langle p, V \rangle)$, not just the procedure that contains p. This fails to account for the calling context, namely the procedure that contains p.

Example. To illustrate this problem, and the shortcomings of Weiser's algorithm, consider the following example program, which sums the integers from 1 to 10. (Recall that parameters are passed by value-result.)

```
program Main
   sum := 0;
   i := 1;
   while i < 11 do
      call A(sum, i)
   od
   output(sum)
   output(i)
end

procedure A(x, y)
   call Add(x, y);
   call Increment(y)
return

procedure Add(a, b)
   a := a + b
return

procedure Increment(z)
   call Add(z, 1)
return
```

Using Weiser's algorithm to slice this program with respect to variable z and the **return** statement of procedure *Increment*, we would obtain everything from the original program except for the two output statements in procedure *Main*. However, a closer inspection reveals that computations involving the variable *sum* do not contribute to the value of z at the end of procedure *Increment*; in particular, neither the initialization of *sum*, nor the first actual parameter of the call on procedure A in *Main*, nor the call on *Add* in A (which adds the current value of i to *sum*) should be included in the slice. The reason these components are included in the slice computed by Weiser's algorithm is as follows: The initial slicing criterion "<end of procedure *Increment*, z>," is mapped by the DOWN relation to a slicing criterion "<end of procedure *Add*, a>." The latter criterion is then mapped by the UP relation to *two* slicing criteria—corresponding to *all* sites that call *Add*—the criterion "<call on *Add* in *Increment*, z>" and the (irrelevant) criterion "<call on *Add* in A, x>." Weiser's algorithm does not produce as precise a slice as possible because transitive closure fails to account for the calling context (*Increment*) of a called procedure (*Add*), and thus generates a spurious criterion (<call on *Add* in A, x>).

A more precise slice consists of the following elements:

```
program Main
   i := 1;
   while i < 11 do
      call A(i)
   od
end

procedure A(y)
   call Increment(y)
return

procedure Add(a, b)
   a := a + b
return

procedure Increment(z)
   call Add(z, 1)
return
```

This set of program elements is computed by the slicing algorithms of Horwitz, Reps, and Binkley [9], and Hwang, Du, and Chou [10].
End of Example.

The algorithms of Horwitz, Reps, and Binkley and Hwang, Du, and Chou improve on Weiser's algorithm by only considering effects transmitted along interprocedurally valid paths. However, whereas the running time of the Horwitz-Reps-Binkley algorithm has been shown to be polynomial in the size of the program [9], it is possible to show that the Hwang-Du-Chou algorithm is exponential. That is, there is a family of examples on which the algorithm takes time exponential in the size of the program being sliced.

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") and to implement the Horwitz-Reps-Binkley slicing algorithm [9]. The system dependence graph extends previous dependence-graph representations to incorporate collections of procedures (with procedure calls) rather than just monolithic programs. The system dependence graph for the example program discussed above is shown in Figure 1.

To understand the results reported in this paper, it is not really important to understand the details of how program dependence graphs are constructed or even the exact nature of the various kinds of edges that occur in them, and for this reason such material is not included in the paper (see [9]). (However, it is important to know that the dependence graph for a program can be constructed in time polynomial in the size of the program.) For the purposes of this paper, it is really only necessary to understand the manner in which dependence graphs for different procedures are linked together at call sites to reflect value-result parameter passing. This involves five kinds of vertices: A call site is represented by a *call-site vertex*; information transfer is represented by four kinds of *parameter 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.

Edges from actual-in vertices to formal-in vertices are called *parameter-in edges*; edges from formal-out vertices to actual-out vertices are called *parameter-out edges*; edges from call-site vertices to entry vertices are called *call edges*. These are shown as dashed lines in Figure 1.

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-

```
program Main                procedure A(x, y)        procedure Add(a, b)     procedure Increment(z)
   sum := 0;                   call Add(x, y);          a := a + b              call Add(z, 1)
   i := 1;                     call Increment(y)     return                  return
   while i < 11 do          return
       call A(sum, i)
   od
   output(sum)
   output(i)
end
```

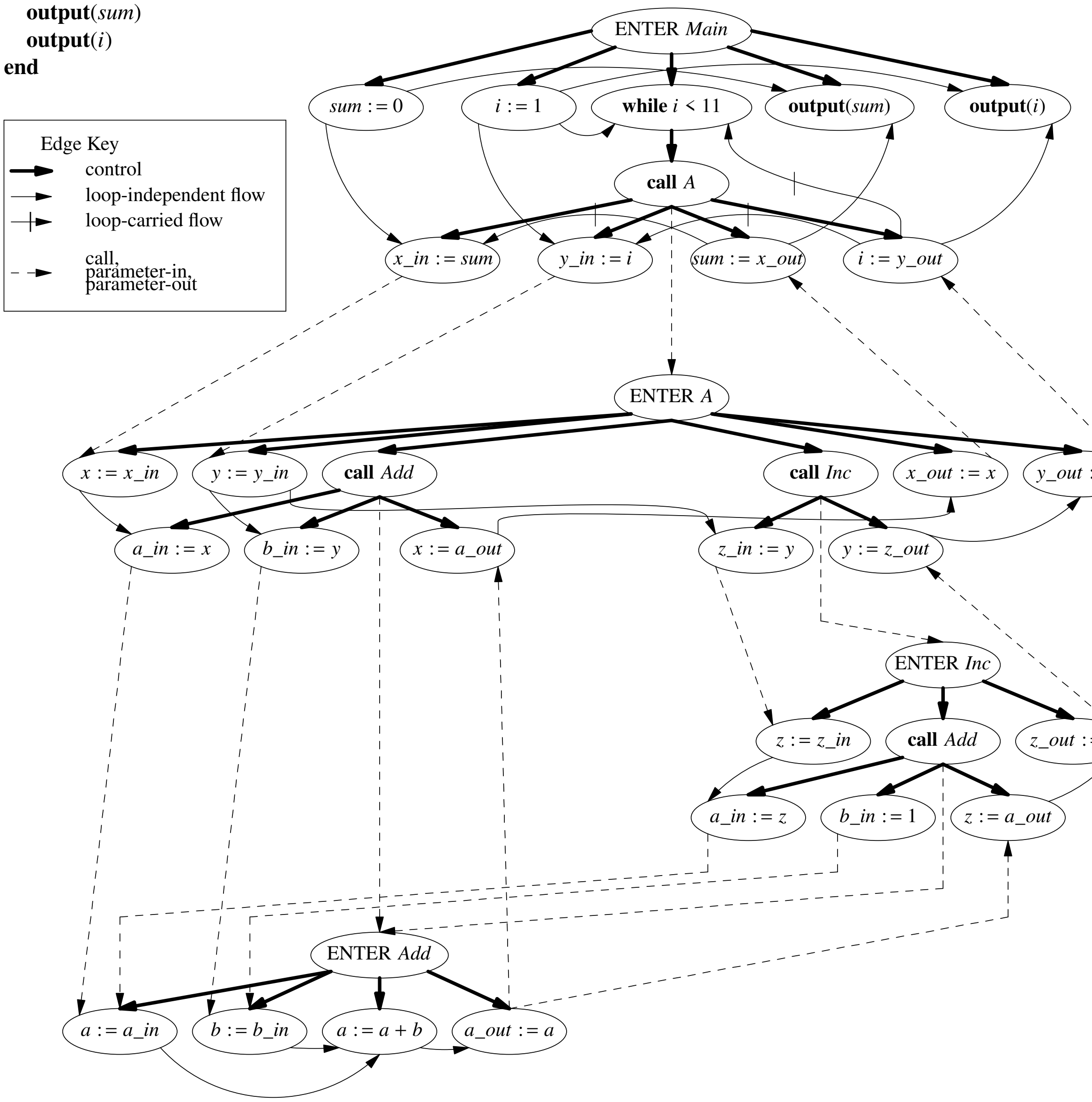

Figure 1. Example program and corresponding procedure dependence graphs connected with parameter-in, parameter-out, and call edges. Edges representing control dependences are shown in bold; edges representing intraprocedural flow dependences are shown using arcs; parameter-in edges, parameter-out edges, and call edges are shown using dashed lines.

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:

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

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):

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

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 Figure 2. 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.

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 vertex v in the slice of p’s system dependence graph with respect to vertex 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 [5], which is defined as follows:

Definition 3.2. 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 $\bar{a}(n) = a(n)$.

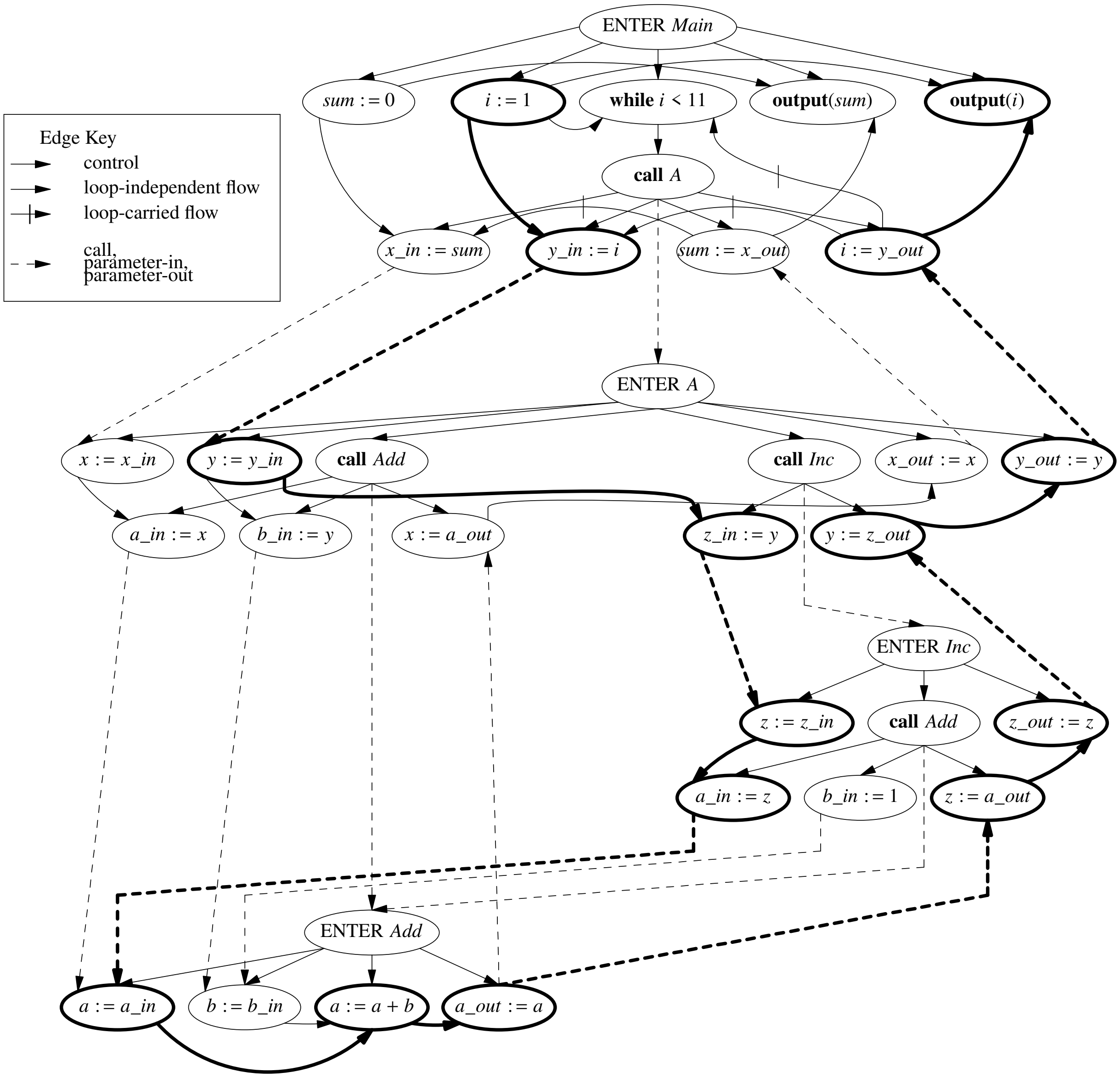

Figure 2. The path shown in bold is a same-level valid path from $i := 1$ to the output vertex for variable i in procedure *Main*.

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

Proof. It was shown in [9] that the Horwitz-Reps-Binkley algorithm for interprocedural slicing runs in polynomial time.[3]

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.[4] 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 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 3. Each instance of a gate in C is translated to a procedure instance of the appropriate kind (two procedure instances in the case of a fan-out gate). Gate names are used to fill in the subscripts on procedure names as gadget instances 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 g 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 g—which is to be found in the portion of the input tape that contains a—and an appropriate procedure is written out for g. Finally, the index recorded on the work tape is used to resume the translation process at the appropriate place in the circuit. Because a gate name and an index position each take no more than $O(\log n)$ bits, where n 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 g in C (with corresponding procedure P_g), there is a same-level valid path in the program's system dependence graph from the formal-in vertex $x := x\_in$ in P_g to the formal-out vertex $x := x\_out$ in P_g iff $\bar{a}(g) = \textbf{true}$. 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 3) or the system dependence graph of P_C (using the graph representation of gadgets shown in column 2.)

Example. Figure 4 shows an example circuit, together with the system dependence graph for the program created via the construction described in the proof of Theorem 3.3. 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.
End of Example.

It is interesting to note that the construction used in Theorem 3.3 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).

Remark. Theorem 3.3 also has consequences for the computation of IO graphs of attribute grammars [13,3,18] 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].

To establish the connection between Theorem 3.3 and attribute grammars it is necessary to introduce the concept of a "linkage grammar"—an attribute grammar that models the call structure of a program together with the (*intra*procedural) transitive dependences among a procedure's parameter vertices. Linkage grammars were introduced by Horwitz, Reps, and Binkley as an abstraction of the system dependence graph [9].[5]

---

[3]More precisely, it was shown in [9] that the running time of the Horwitz-Reps-Binkley algorithm is bounded by $O(\mathrm{P} \cdot (\mathrm{Sites} + 1)^3 \cdot (\mathrm{Globals} + \mathrm{Params})^4)$, where P is the number of procedures in the program, Sites is the largest number of call sites in any procedure, Params is the largest number of formal parameters in any procedure, and Globals is the number of global variables in the program. (This result holds even when the program being sliced uses recursive—or mutually recursive—procedures.)

[4]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.

[5]In general, it is entirely possible that the linkage grammar that corresponds to a given system dependence graph will be a circular attribute grammar (*i.e.* there may be attributes in some derivation tree of the grammar that depend on themselves); additionally, the grammar may not be well-formed (*e.g.*, a production may have equations for synthesized attribute occurrences of right-hand-side symbols). This does not create any difficulties because the linkage grammar is used *only* to represent dependences and *not* for computing any attribute values. (The linkage grammars that arise in the proof of Corollary 3.4 are all well-formed.)

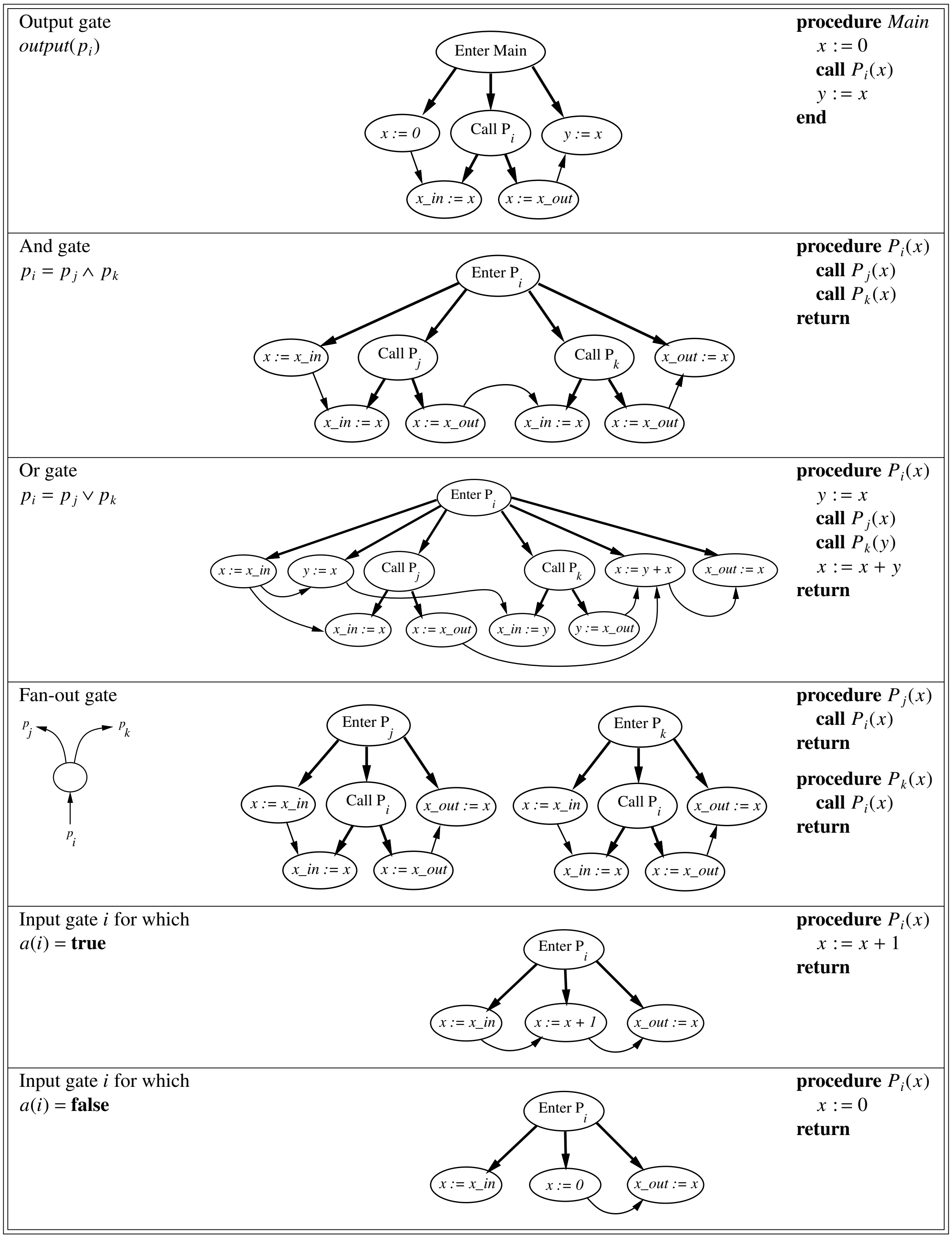

Figure 3. Gadgets used in the construction of a program in which a particular same-level slice simulates a given monotone Boolean circuit under a given assignment of truth values to input gates. The output gate of the circuit has value **true** iff the statement $x := 0$ in procedure *Main* is in the same-level slice of the program with respect to statement $y := x$ in procedure *Main*.

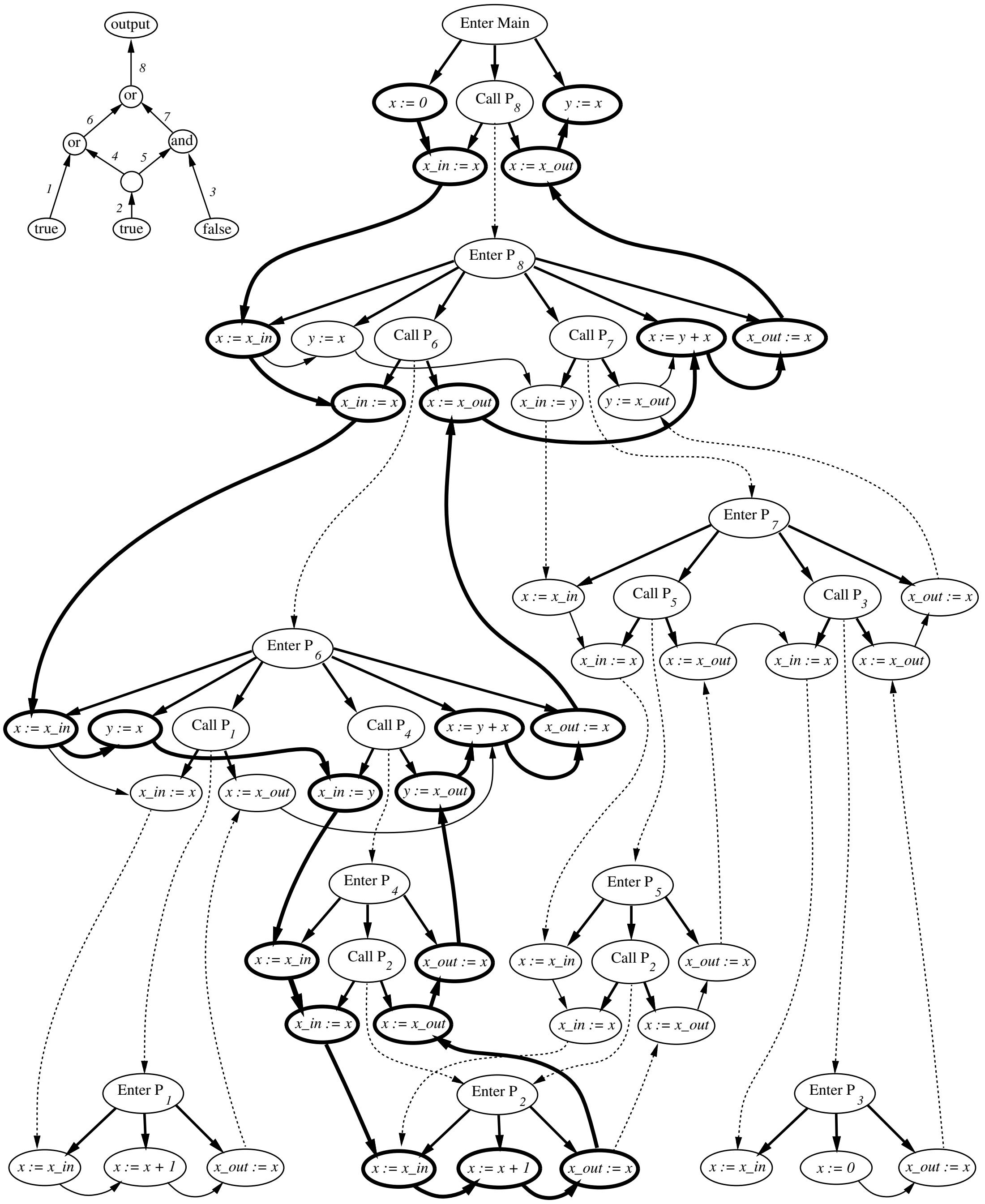

Figure 4. An example circuit (upper-left-hand corner), together with the system dependence graph for the program created via the construction used the proof of Theorem 3.3. The circuit evaluates to **true**, and one of the same-level valid paths from $x := 0$ in *Main* to $y := x$ in *Main* is outlined in bold.

The context-free part of a program's linkage grammar models the program's procedure-call structure. The linkage grammar includes one nonterminal and one production for each procedure in the system. If procedure P contains no calls, the right-hand side of the production for P is ε; otherwise, there is one right-hand-side nonterminal for each call site in P.

Example. For the example shown in Figure 1, the productions of the linkage grammar are as follows:

$$Main \rightarrow A \qquad A \rightarrow Add \;\; Increment \qquad Add \rightarrow \varepsilon \qquad Increment \rightarrow Add$$

End of Example.

The attributes in the linkage grammar correspond to the parameters of the procedures. Procedure inputs are modeled as inherited attributes; procedure outputs are modeled as synthesized attributes.

More precisely, the program's linkage grammar has the following elements:

- For each procedure P, the linkage grammar contains a nonterminal P.
- For each procedure P, there is a production $p: \; P \rightarrow \beta$, where for each site of a call on procedure Q in P there is a distinct occurrence of Q in β.
- For each formal-in vertex $v := v\_in$ of P, there is an inherited attribute v of nonterminal P.
- For each formal-out vertex $v\_out := v$ of P, there is a synthesized attribute v' of nonterminal P.

Dependences among the attributes of a linkage-grammar production are used to model the (possibly transitive) *intra*procedural dependences (both control and flow dependences) among the parameter vertices of the corresponding procedure.

Example. Figure 5 shows the linkage grammar for the example from Figure 1.

Corollary 3.4. *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. By an argument along the lines of the one given by Kastens that shows that TDS graphs of ordered attribute grammars can be computed in polynomial time [12], it can be shown that the IO graphs of an attribute grammar's nonterminals can be computed in polynomial time.

The proof that the problem is $\mathcal{P}$-hard is by reduction from the monotone circuit value problem. A given circuit is transformed into the linkage grammar for the program constructed in the proof of Theorem 3.3: there is one production per gate of the circuit; the attribute dependences in each production reflect the dependences among parameter vertices in the procedure dependence graphs shown in column 2 of Figure 3. (As can be seen from Figure 3, this will always be a well-formed attribute grammar.)

Let procedure P be the procedure called from procedure *Main*. Let v be the attribute corresponding to the formal-in vertex of P, and w be the attribute corresponding to the formal-out vertex. The output gate of the circuit has value **true** iff the IO graph for nonterminal P has an edge from v to w.

□

End of Remark.

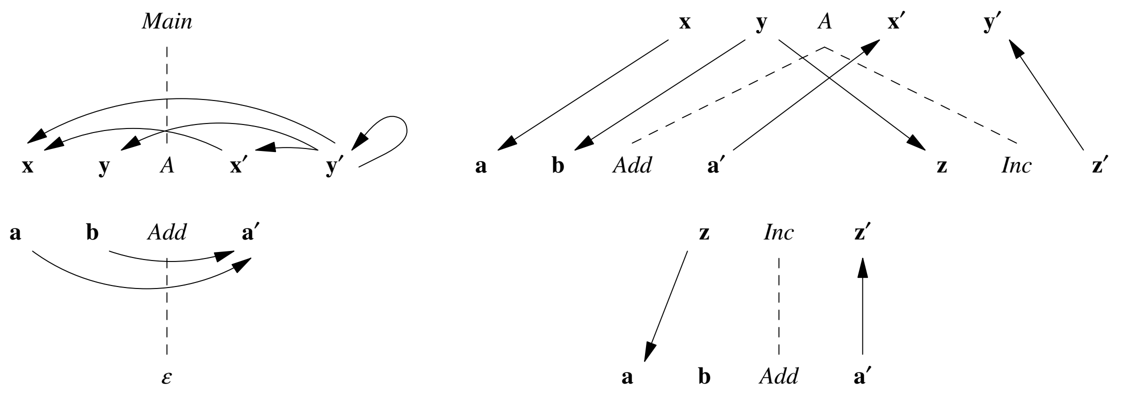

Figure 5. The linkage grammar for the example from Figure 1.

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

In this section, we assume that $(L, \sqcap)$ is a meet semilattice of dataflow values with a smallest element $\perp$ and a largest element $\top$. We also assume that dataflow functions are members of a space of monotonic (or distributive) functions $F \subseteq L \to L$ and that F contains the identity function.

Sharir and Pnueli's "functional approach" to interprocedural dataflow analysis 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 [22]. 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 loss of generality.)

Sharir and Pnueli make use of two different graph representations of programs, which are defined below.

Definition 4.1. (Sharir and Pnueli [22]). Define $G = \cup \{ G_p \mid p \text{ is a procedure in the program} \}$, where, for each p, $G_p = (N_p, E_p, r_p)$. Vertex r_p is the entry block of p; N_p is the set of basic blocks in p; $E_p = E_p^0 \cup E_p^1$ is the set of edges of G_p. An edge $(m, n) \in E_p^0$ is an ordinary control-flow edge; it represents a direct transfer of control from one block to another via a goto or an if statement. An edge $(m, n) \in E_p^1$ iff m is a call block and n is the return-site block in p for that call. Observe that vertex n is *within* p as well; it is important to understand that an edge in E_p^1 does *not* run from p to the called procedure, or vice versa.

Without loss of generality, we assume that (i) a return-site block in any G_p graph has exactly one incoming edge: the E_p^1 edge from the corresponding call block; (ii) the entry block r_p in any G_p graph is never the target of an E_p^0 edge.

This first representation, in which the flow graphs of individual procedures are kept separate from each other, is used in an algorithm that Sharir and Pnueli give for interprocedural dataflow analysis algorithms (see the proof of Corollary 4.6). The second graph representation, in which the flow graphs of the different procedures are connected together, is used to define the notion of interprocedurally valid paths.

Definition 4.2. (Sharir and Pnueli [22]). Define $G^* = (N^*, E^*, r_{main})$, where $N^* = \bigcup_p N_p$ and $E^* = E^0 \cup E^2$, where $E^0 = \bigcup_p E_p^0$ is the collection of all ordinary control-flow edges, and an edge $(m, n) \in E^2$ represents either a ***call*** or ***return*** edge. Edge $(m, n) \in E^2$ is a call edge iff m is a call block and n is the entry block of the called procedure; edge $(m, n) \in E^2$ is a return edge iff m is an exit block of some procedure p and n is a return-site block for a call on p. A call edge (m, r_p) and return edge (e_q, n) ***correspond*** to each other if $p = q$ and $(m, n) \in E_s^1$ for some procedure s.

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

Definition 4.3. (Sharir and Pnueli [22]). For each $n \in N$, we define IVP(r_{main}, n) as the set of all ***interprocedurally valid paths*** in G^* that lead from r_{main} to n. A path $q \in \text{path}_{G^*}(r_{main}, n)$ is in IVP(r_{main}, n) iff the sequence of all edges in q that are in E^2, which we will denote by q_2, is ***proper*** in the following recursive sense:[6]

(i) A sequence q_2 that contains no return edges is proper.
(ii) If q_2 contains return edges, and i is the smallest index in q_2 such that $q_2(i)$ is a return edge, then q_2 is proper if $i > 1$ and $q_2(i-1)$ is a call edge corresponding to the return edge $q_2(i)$, and after deleting those two components from q_2, the remaining sequence is also proper.

Definition 4.4. (Sharir and Pnueli [22]). If q is a path in G^*, let f_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:

[6] The notion of a proper sequence is equivalent to the notion of *unbalanced_left* given in Definition 3.1.

$$\Phi_n = \bigsqcap_{q \in \text{IVP}(r_{main}, n)} f_q \qquad \text{for each } n \in N^*$$
$$y_n = \Phi_n(\bot) \qquad \text{for each } n \in N^*$$

Theorem 4.5. *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 Sharir-Pnueli dataflow framework. 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 3.3: 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 6. In general, the only edge functions that are not the identity function are in the procedure for the output gate and in the procedures for variables with value **false**. In particular, the dataflow fact of interest is introduced in procedure *Main* (*e.g.*, in Figure 6 the initial dataflow fact is "x is defined at statement $x := 0$ of *Main*"). The dataflow fact is killed in procedures that correspond to variables with value **false** (*e.g.*, $x := x + 1$).

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 initial dataflow fact is *not* killed iff $\bar{a}(n) =$ **true**. Consequently, the output gate of circuit C has value **true** iff the initial dataflow fact is in the fact set for the Exit vertex of procedure *Main*.

□

Note that the construction used in the proof of Theorem 4.5 does not carry over to the case of *intra*procedural dataflow analysis. In particular, the transformation presented above uses multiple calls and same-level valid paths to simulate shared subexpressions that occur in the circuit.

Instances of the Sharir-Pnueli 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 holds for most of the so-called gen/kill problems (problems in which the edge functions are of the form $\lambda\, x\, .\, (x - kill) \cup gen$, where *kill* and *gen* are subsets):

Corollary 4.6. *Every interprocedural gen/kill dataflow analysis problem in which the domain of dataflow values L 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, is log-space complete for $\mathcal{P}$.*

Proof. Sharir and Pnueli have shown that if the edge functions are distributive, the meet-over-all-valid-paths solution can be determined via a two-phase process, where each phase involves finding the greatest solution to a set of equations [22].

The meet-over-all-valid-paths solution is not obtained directly from the equations of Definition 4.4, but from two other systems of equations, which are solved in separate phases. In Phase I, the equations deal with *summary dataflow functions*, which are defined in terms of dataflow functions and other summary dataflow functions. In Phase II, the equations deal with actual dataflow *values*.

Phase I of the process is to compute summary functions $\phi_{(r_p, n)}(x)$ that map a set of dataflow facts at r_p—the entry point of procedure p—to the set of dataflow facts at point n within p. These functions are defined as the greatest solution to the following set of equations (computed over a (bounded) meet semilattice of functions):

$$\phi_{(r_p, r_p)} = \lambda\, x\, .\, x \qquad \text{for each procedure } p$$
$$\phi_{(r_p, n)} = \bigsqcap_{(m, n) \in E_p} (f_{(m,n)} \circ \phi_{(r_p, m)}) \qquad \text{for each } n \in N_p \text{ not representing a return-site block}$$
$$\phi_{(r_p, n)} = \phi_{(r_q, e_q)} \circ \phi_{(r_p, m)} \qquad \text{for each } n \in N_p \text{ representing a return-site block, where } (m, n) \in E_p^1 \text{ and } m \text{ calls procedure } q$$

Phase II of the process uses the summary functions from Phase I to obtain a solution to the dataflow analysis problem proper. This solution is obtained from the greatest solution of the following set of equations:

$$x_{r_{main}} = \bot$$
$$x_{r_p} = \bigsqcap \{\, \phi_{(r_q, c)}(x_{r_q}) \mid c \text{ is a call to } p \text{ in procedure } q \,\} \qquad \text{for each procedure } p$$
$$x_n = \phi_{(r_p, n)}(x_{r_p}) \qquad \text{for each procedure } p \text{ and } n \in (N_p - \{\, r_p \,\})$$

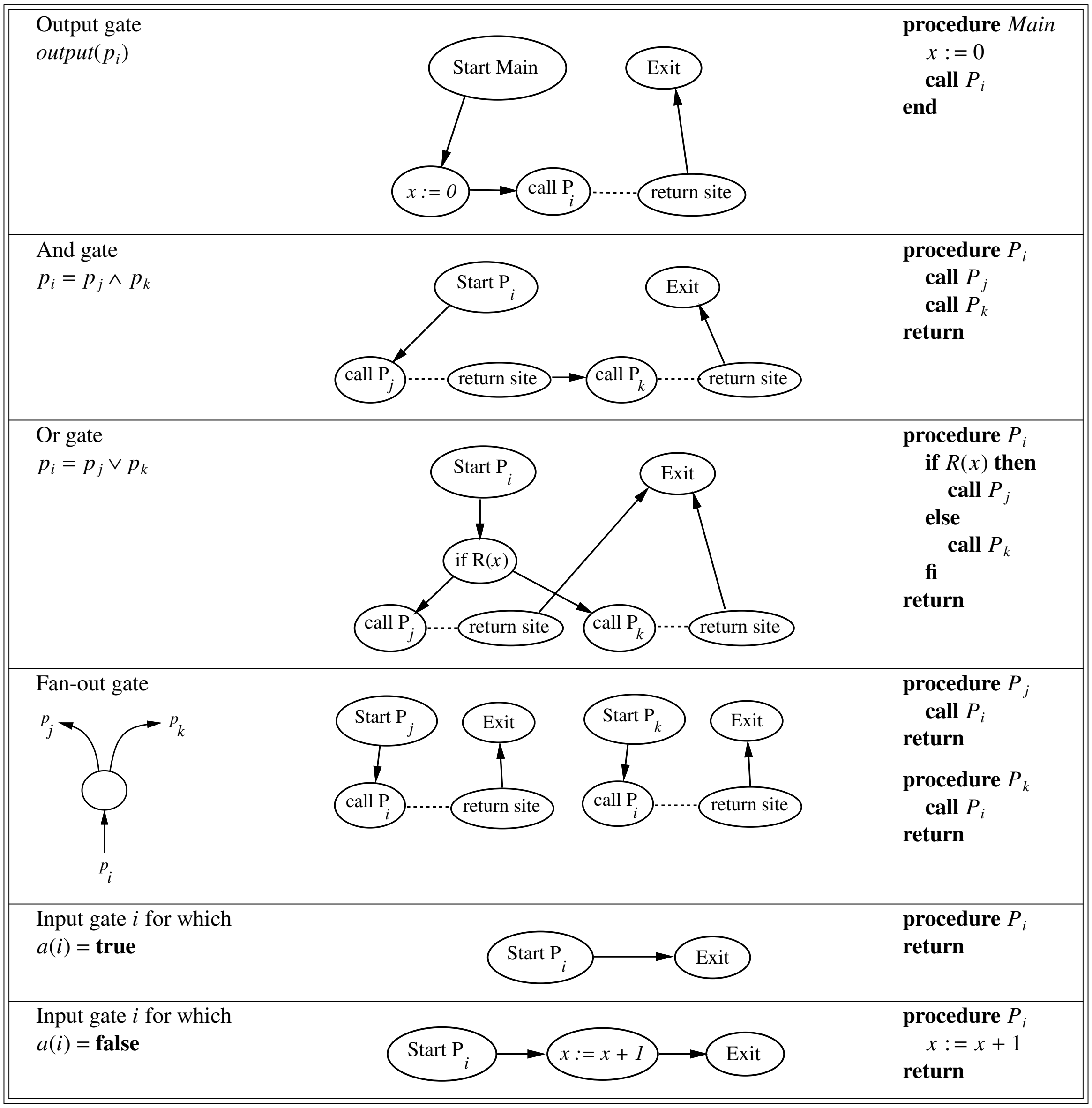

Figure 6. Gadgets used in the construction of a program in which a particular reaching-definitions problem simulates a given monotone Boolean circuit under a given assignment of truth values to input gates. Edges in E_p^0 are indicated with solid directed arrows. Edges in E_p^1, which connect call sites 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*.

Now consider gen/kill dataflow analysis problems in which the domain of dataflow values L 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. We may assume without loss of generality that $\cup$ is the meet operator and that the edge functions are of the form $\lambda x . (x \cap nkill) \cup gen$, where $nkill$ ($= \overline{kill}$) and gen are subsets of D. Note that such functions are distributive. Furthermore, each such function can be represented as a pair of sets: $(nkill, gen)$. Given this representation of edge functions, it is easy to verify that the rules for performing the composition and meet of two functions are as follows:

$$(nkill_2, gen_2) \circ (nkill_1, gen_1) = (nkill_1 \cap nkill_2, (gen_1 \cap nkill_2) \cup gen_2)$$
$$(nkill_2, gen_2) \sqcap (nkill_1, gen_1) = (nkill_1 \cup nkill_2, gen_1 \cup gen_2)$$

Using such a representation, the two sets of equations in the Sharir-Pnueli solution can both be solved in polynomial time (*e.g.*, via chaotic iteration). $\mathcal{P}$-completeness follows from Theorem 4.5.
□

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 [21].)

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 [6] 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 [16]. (Valid paths are called "realizable" paths in [16].) 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.*). 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.

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. Goldschlager, L., "The monotone and planar circuit value problems are log-space complete for P," *ACM SIGACT News* **9**(2) pp. 25-29 (1977).
6. 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).
7. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *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. 35-46 (July 1988).
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).
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. 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).

17. 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).

18. 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).

19. 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).

20. 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).

21. Robinson, J.A., "Logic and logic programming," *Commun. of the ACM* **35**(3) pp. 40-65 (March 1992).

22. 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).

23. Weiser, M., "Reconstructing sequential behavior from parallel behavior projections," *Information Processing Letters* **17** pp. 129-135 (October 1983).

24. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).