Precise Interprocedural Dataflow Analysis via Graph Reachability

(Extended Abstract)

Thomas Reps, Susan Horwitz, and Mooly Sagiv[†]
University of Wisconsin

1. Introduction

This paper shows how to find precise solutions to a large class of interprocedural dataflow-analysis problems in polynomial time. In contrast with *intra*procedural dataflow analysis, where "precise" means "meet-over-all-paths"[16], a precise *inter*procedural dataflow-analysis algorithm must provide the "meet-over-all-*valid*-paths" solution. (A path is *valid* if it respects the fact that when a procedure finishes it returns to the site of the most recent call [25,5,20,17]—see Section 2.)

Relevant previous work on precise interprocedural dataflow analysis can be categorized as follows:

- Polynomial-time algorithms for specific individual problems (*e.g.*, constant propagation [4,12], flow-sensitive summary information [5], and pointer analysis [20]).
- A polynomial-time algorithm for a limited class of problems: the *locally separable* problems (the interprocedural versions of the classical "bit-vector" or "gen-kill" problems), which include reaching definitions, available expressions, and live variables [18].
- Algorithms for a very general class of problems [9,25,17].

The work cited in the third category concentrated on generality and did not provide polynomial-time algorithms.

In contrast to this previous work, the present paper provides a polynomial-time algorithm for finding precise solutions to a general class of interprocedural dataflow-analysis problems. This class consists of all problems in which the set of dataflow facts D is a finite set and the dataflow functions (which are in $2^D \rightarrow 2^D$) distribute over the meet operator (either union or intersection, depending on the problem). We will call this class the *interprocedural, finite, distributive, subset problems*, or *IFDS problems*, for short. All of the locally separable problems are IFDS problems. In addition, many non-separable problems of practical importance are also IFDS problems—for example: truly-live variables [11], copy constant propagation [10, pp. 660], and possibly-uninitialized variables.

The most important aspects of our work can be summarized as follows:

- In Section 3, we show that all IFDS problems can be solved precisely by transforming them into a special kind of graph-reachability problem: reachability along ***interprocedurally realizable paths***. In contrast with ordinary reachability problems in directed graphs (*e.g.*, transitive closure), realizable-path reachability problems involve some constraints on which paths are considered. A realizable path mimics the call-return structure of a program's execution, and only paths in which "returns" can be matched with corresponding "calls" are considered.
- In Section 4, we present a new polynomial-time algorithm for the realizable-path reachability problem. The algorithm runs in time $O(ED^3)$; this is ***asymptotically faster*** than the best previously known algorithm for the problem [13], which runs in time $O(D^3 \sum_p Call_p E_p + D^4 \sum_p Call_p^3)$.
- As discussed in Section 5, the new realizable-path reachability algorithm is ***adaptive***, with asymptotically better performance when applied to common kinds of problem instances that have restricted form. For example, there is an asymptotic improvement in the algorithm's performance for the common case of locally separable problems. Our work generalizes that of Knoop and Steffen [18] in the sense that our algorithm handles a much larger class of problems, yet on the locally separable problems the algorithm runs in the same time as that used by the Knoop-Steffen algorithm—$O(ED)$.

[†]On leave from IBM Israel, Haifa Research Laboratory.

Authors' address: Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706.
Electronic mail: {reps, horwitz, sagiv}@cs.wisc.edu.

- Imprecise (overly conservative) answers to interprocedural dataflow-analysis problems could be obtained by treating each interprocedural dataflow-analysis problem as if it were essentially one large intraprocedural problem. In graph-reachability terminology, this amounts to considering all paths rather than considering only the interprocedurally realizable paths. For the IFDS problems, ***we can bound the extra cost needed to obtain the more precise (realizable-path) answers***. In the important special case of locally separable problems, there is no "penalty" at all—both kinds of solutions can be obtained in time $O(ED)$. In the distributive case, the penalty is a factor of D: the running time of our realizable-path reachability algorithm is $O(ED^3)$, whereas all-paths reachability solutions can be found in time $O(ED^2)$. In the preliminary experiments reported in Section 6, which involve examples where D is in the range 70-142, the penalty is at most a factor of 3.4.
- Callahan has given algorithms for several "interprocedural flow-sensitive side-effect problems"[5]. Although these problems are (from a certain technical standpoint) of a slightly different character from the IFDS dataflow-analysis problems, with small adaptations the algorithm from Section 4 can be applied to these problems and is ***asymptotically faster*** than the algorithm given by Callahan. In addition, our algorithm handles a natural generalization of Callahan's problems (which are locally separable problems) to a class of distributive flow-sensitive side-effect problems. (This and other related work is described in Section 7.)
- The realizable-path reachability problem is also the heart of the problem of interprocedural program slicing, and the fastest previously known algorithm for the problem is the one given by Horwitz, Reps, and Binkley [13]. The realizable-path reachability algorithm described in this paper yields an ***improved interprocedural-slicing algorithm***—one whose running time is asymptotically faster than the Horwitz-Reps-Binkley algorithm. This algorithm has been found to run six times as fast as the Horwitz-Reps-Binkley algorithm [24].
- Our dataflow-analysis algorithm has been implemented and used to analyze several C programs. Experimental results are reported in Section 6.

(Due to space constraints, some of the above material is omitted from this extended abstract. An analysis of the running time of the dataflow-analysis algorithm is given in the Appendix. Omitted proofs can be found in [23].)

2. The IFDS Framework for Distributive Interprocedural Dataflow-Analysis Problems

The IFDS framework is a variant of Sharir and Pnueli's "functional approach" to interprocedural dataflow analysis [25], with an extension similar to the one given by Knoop and Steffen in order to handle programs in which recursive procedures have local variables and parameters [17]. These frameworks generalize Kildall's concept of the "meet-over-all-paths" solution of an *intra*procedural dataflow-analysis problem [16] to the "meet-over-all-valid-paths" solution of an *inter*procedural dataflow-analysis problem.

The IFDS framework is designed to be as general as possible (in particular, to support languages with procedure calls, parameters, and both global and local variables). Any problem that can be specified in this framework can be solved efficiently using our algorithms; semantic correctness is an orthogonal issue. A problem designer who wishes to take advantage of our results has two obligations: (i) to encode the problem so that it meets the conditions of our framework; (ii) to show that the encoding is consistent with the programming language's semantics [8,9].

To specify the IFDS framework, we need the following definitions:

Definition 2.1. In the IFDS framework, a program is represented using a directed graph $G^* = (N^*, E^*)$ called a ***super graph***. G^* consists of a collection of flow graphs $G_1, G_2, \cdots$ (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 the statements and predicates of the procedure in the usual way, except that a procedure call is represented by two nodes, a ***call*** node and a ***return-site*** node. (The sets of call and return-site nodes of procedure p will be denoted by $Call_p$ and Ret_p, respectively; the sets of all call and return-site nodes in the super graph will be denoted by $Call$ and Ret, respectively.)

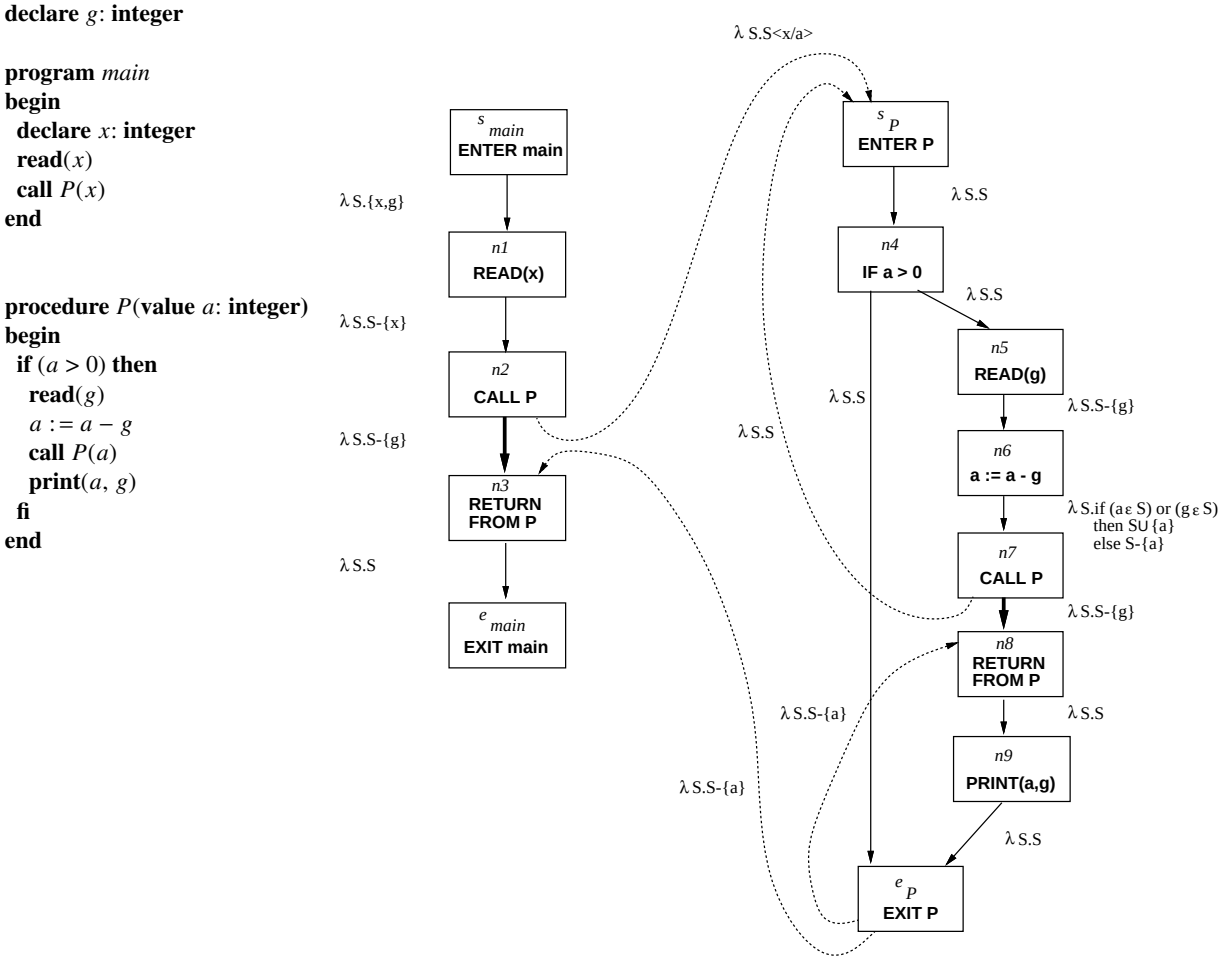
In addition to the ordinary intraprocedural edges that connect the nodes of the individual flowgraphs, for each procedure call, represented by call-node c and return-site node r, G^* has three edges:

- 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. □

(The call-to-return-site edges are included so that the IFDS framework 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.)

Example. Figure 1 shows an example program and its super graph G^*. □

Definition 2.2. A ***path*** of length j from node m to node n is a (possibly empty) sequence of j edges, which will be denoted by $[e_1, e_2, \cdots, e_j]$, such that for all i, $1 \le i \le j-1$, the target of edge e_i is the source of edge e_{i+1}. □



(a) Example program (b) Its super-graph G^*

Figure 1. An example program and its super-graph G^*. The super-graph is annotated with the dataflow functions for the "possibly-uninitialized variables" problem. The notation S<x/a> denotes the set S with x renamed to a.

The notion of an "interprocedurally valid path" captures the idea that not all paths in G^* represent potential execution paths:

Definition 2.3. Let each call node in G^* be given a unique index i. For each such indexed call node c_i, label c_i's outgoing call-to-start edge by the symbol "$(_i$". Label the incoming exit-to-return-site edge of the corresponding return-site node by the symbol "$)_i$".

For each pair of nodes m, n in the same procedure, a path from m to n is a ***same-level interprocedurally valid path*** iff the sequence of labeled edges in the path is a string in the language of balanced parentheses generated from nonterminal *matched* by the following context-free grammar:

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

For each pair of nodes m, n in super-graph G^*, a path from m to n is an ***interprocedurally valid path*** iff the sequence of labeled edges in the path is a string in the language generated from nonterminal *valid* in the following grammar (where *matched* is as defined above):

$$\begin{array}{lll} valid & \rightarrow valid \;\; (_i \;\; matched & \text{for } 1 \le i \le |Call| \\ & \mid \; matched & \end{array}$$

We denote the set of all interprocedurally valid paths from m to n by IVP(m, n). □

In the formulation of the IFDS dataflow-analysis framework (see Definitions 2.4–2.6 below), the same-level valid paths from m to n will be used to capture the transmission of effects from m to n, where m and n are in the same procedure, via sequences of execution steps during which the call stack may temporarily grow deeper—because of calls—but never shallower than its original depth, before eventually returning to its original depth. The valid paths from s_{main} to n will be used to capture the transmission of effects from s_{main}, the program's start node, to n via some sequence of execution steps. Note that, in general, such an execution sequence will end with some number of activation records on the call stack; these correspond to "unmatched" $(_i$'s in a string of language $L(valid)$.

Example. In super-graph G^* shown in Figure 1, the path

$$[(s_{main}, n1), (n1, n2), (n2, s_p), (s_p, \; n4), (n4, e_p), (e_p, n3)]$$

is a (same-level) interprocedurally valid path; however, the path

$$[(s_{main}, n1), (n1, n2), (n2, s_p), (s_p, n4), (n4, e_p), (e_p, n8)]$$

is not an interprocedurally valid path because the return edge $(e_p, n8)$ does not correspond to the preceding call edge $(n2, s_p)$. □

We now define the notion of an instance of an IFDS problem:

Definition 2.4. An ***instance*** *IP* of an ***interprocedural, finite, distributive, subset problem*** (or ***IFDS problem***, for short) is a five-tuple, $IP = (G^*, D, F, M, \sqcap)$, where

(i) G^* is a super-graph as defined in Definition 2.1.

(ii) D is a finite set.

(iii) $F \subseteq 2^D \rightarrow 2^D$ is a set of distributive functions.

(iv) $M\colon E^* \rightarrow F$ is a map from G^*'s edges to dataflow functions.

(v) The meet operator $\sqcap$ is either union or intersection. □

In the remainder of the paper we consider only IFDS problems in which the meet operator is union. It is not hard to show that IFDS problems in which the meet operator is intersection can always be handled by transforming them to the complementary union problem. Informally, if the "must-be-X" problem is an intersection IFDS problem, then the "may-not-be-X" problem is a union IFDS problem. Furthermore, for each node $n \in N^*$, the solution to the "must-be-X" problem is the complement (with respect to D) of the solution to the "may-not-be-X" problem.

Example. In Figure 1, the super-graph is annotated with the dataflow functions for the "possibly-uninitialized variables" problem. The "possibly-uninitialized variables" problem is to determine, for

each node $n \in N^*$, the set of program variables that may be uninitialized just before execution reaches n. A variable x is possibly uninitialized at n either if there is an x-definition-free valid path to n or if there is a valid path to n on which the last definition of x uses some variable y that itself is possibly uninitialized. For example, the dataflow function associated with edge $(n6, n7)$ shown in Figure 1 adds a to the set of possibly-uninitialized variables if either a or g is in the set of possibly-uninitialized variables before node $n6$. □

Definition 2.5. Let $IP = (G^*, D, F, M, \sqcap)$ be an IFDS problem instance, and let $q = [e_1, e_2, \cdots, e_j]$ be a non-empty path in G^*. The ***path function*** that corresponds to q, denoted by pf_q, is the function $pf_q =_{df} f_j \circ \cdots \circ f_2 \circ f_1$, where for all i, $1 \le i \le j$, $f_i = M(e_i)$. The path function for an empty path is the identity function, $\lambda x \,.\, x$. □

Definition 2.6. Let $IP = (G^*, D, F, M, \sqcap)$ be an IFDS problem instance. The ***meet-over-all-valid-paths*** solution to IP consists of the collection of values MVP_n defined as follows:

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

□

3. Interprocedural Dataflow Analysis as a Graph-Reachability Problem

3.1. Representing Distributive Functions

In this section, we show how to represent distributive functions in $2^D \to 2^D$ in a compact fashion—each function can be represented as a graph with at most $(|D|+1)^2$ edges (or, equivalently, as an adjacency matrix with at most $(|D|+1)^2$ bits). Throughout this section, we assume that f and g denote functions in $2^D \to 2^D$ and that f and g distribute over $\cup$.

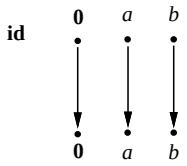
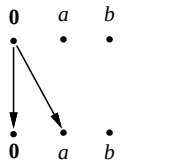
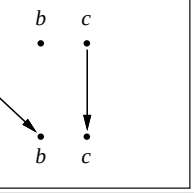
Definition 3.1. The ***representation relation of f***, $R_f \subseteq (D \cup \{\, \mathbf{0} \,\}) \times (D \cup \{\, \mathbf{0} \,\})$, is a binary relation (*i.e.*, graph) defined as follows:

$$\begin{aligned} R_f =_{df} \quad & \{\, (\mathbf{0}, \mathbf{0}) \,\} \\ & \cup \{\, (\mathbf{0}, y) \mid y \in f(\emptyset) \,\} \\ & \cup \{\, (x, y) \mid y \in f(\{\, x \,\}) \text{ and } y \notin f(\emptyset) \,\}. \end{aligned}$$

□

R_f can be thought of as a graph with $2|D|+2$ nodes, where each node represents an element of D (except for the two **0** nodes, which (roughly) stand for $\emptyset$).

Example. The following table shows three functions and their representation relations:

id: $2^{\{a,b\}} \to 2^{\{a,b\}}$ **id** $=_{df} \lambda S \,.\, S$	**a**: $2^{\{a,b\}} \to 2^{\{a,b\}}$ **a** $=_{df} \lambda S \,.\, \{\, a \,\}$	**f**: $2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ **f** $=_{df} \lambda S \,.\, (S - \{\, a \,\}) \cup \{\, b \,\}$
		

Note that a consequence of Definition 3.1 is that edges in representation relations obey a kind of "subsumption property". That is, if there is an edge $(\mathbf{0}, y)$, for $y \in (D \cup \{\, \mathbf{0} \,\})$, there is never an edge (x, y), for any $x \in D$. For example, in constant-function **a**, edge $(\mathbf{0}, a)$ subsumes the need for edges (a, a) and (b, a). □

Representation relations—and, in fact, all relations in $(D \cup \{\, \mathbf{0} \,\}) \times (D \cup \{\, \mathbf{0} \,\})$—can be interpreted as functions in $2^D \to 2^D$, as follows:

Definition 3.2. Given a relation $R \subseteq (D \cup \{\, \mathbf{0} \,\}) \times (D \cup \{\, \mathbf{0} \,\})$, its ***interpretation*** $[\![R]\!]\colon 2^D \to 2^D$ is the function defined as follows:

$$[\![R]\!] =_{df} \lambda X . (\{ y \mid \exists x \in X \text{ such that } (x, y) \in R \} \cup \{ y \mid (\mathbf{0}, y) \in R \}) - \{ \mathbf{0} \}. \qquad \square$$

Theorem 3.3. $[\![R_f]\!] = f$.[1]

Our next task is to show how the relational composition of two representation relations R_f and R_g relates to the function composition $g \circ f$.

Definition 3.4. Given two relations $R_f \subseteq S \times S$ and $R_g \subseteq S \times S$, their ***composition*** $R_f ; R_g \subseteq S \times S$ is defined as follows:

$$R_f ; R_g =_{df} \{ (x, y) \in S \times S \mid \exists z \in S \text{ such that } (x, z) \in R_f \text{ and } (z, y) \in R_g \}. \qquad \square$$

Theorem 3.5. *For all* $f, g \in 2^D \to 2^D$, $[\![R_f ; R_g]\!] = g \circ f$.

Corollary 3.6. *Given a collection of functions* $f_i \colon 2^D \to 2^D$, *for* $1 \le i \le j$,

$$f_j \circ f_{j-1} \circ \cdots \circ f_2 \circ f_1 = [\![R_{f_1} ; R_{f_2} ; \cdots ; R_{f_j}]\!].$$

3.2. From Dataflow-Analysis Problems to Realizable-Path Reachability Problems

In this section, we show how to convert IFDS problems to "realizable-path" graph-reachability problems. In particular, for each instance *IP* of an IFDS problem, we construct a graph $G^{\#}_{IP}$ and an instance of a realizable-path reachability problem in $G^{\#}_{IP}$. The edges of $G^{\#}_{IP}$ correspond to the representation relations of the dataflow functions on the edges of G^*. Because of the relationship between function composition and paths in composed representation-relation graphs (Corollary 3.6), the path problem can be shown to be equivalent to *IP*: dataflow-fact d holds at super-graph node n iff there is a "realizable path" from a distinguished node in $G^{\#}_{IP}$ (which represents the fact that $\emptyset$ holds at the start of procedure *main*) to the node in $G^{\#}_{IP}$ that represents fact d at node n (see Theorem 3.8).

Definition 3.7. Let $IP = (G^*, D, F, M, \cup)$ be an IFDS problem instance. We define the ***exploded super-graph*** for *IP*, denoted by $G^{\#}_{IP}$, as follows:

$$\begin{aligned} &G^{\#}_{IP} = (N^{\#}, E^{\#}), \text{ where} \\ &\quad N^{\#} = N^* \times (D \cup \{ \mathbf{0} \}), \\ &\quad E^{\#} = \{ ((m, d_1), (n, d_2)) \mid (m, n) \in E^* \text{ and } (d_1, d_2) \in R_{M(m,n)} \}. \end{aligned} \qquad \square$$

The nodes of $G^{\#}_{IP}$ are pairs of the form (n, d); each node n of N_p is "exploded" into $|D|+1$ nodes of $G^{\#}_{IP}$. Each edge e of E^* with dataflow function f is "exploded" into a number of edges of $G^{\#}_{IP}$ according to representation relation R_f. Dataflow-problem *IP* corresponds to a (single-source) reachability problem in $G^{\#}_{IP}$, where the source node is $(s_{main}, \mathbf{0})$.

Example. The exploded super-graph that corresponds to the instance of the "possibly-uninitialized variables" problem shown in Figure 1 is shown in Figure 2. $\square$

Throughout the remainder of the paper, we use the terms "(same-level) realizable path" and "(same-level) valid path" to refer to two related concepts in the exploded super-graph and the super-graph. For both "realizable paths" and "valid paths", the idea is that not every path corresponds to a potential execution path: the constraints imposed on paths mimic the call-return structure of a program's execution, and only paths in which "returns" can be matched with corresponding "calls" are permitted. However, the term "realizable paths" will always be used in connection with paths in the exploded super-graph; the term "valid paths" will always be used in connection with paths in the super-graph.

We now state the main theorem of this section, Theorem 3.8, which shows that an IFDS problem instance *IP* is equivalent to a realizable-path reachability problem in graph $G^{\#}_{IP}$:

---

[1] This is similar to Lemma 14 of Cai and Paige [3]; however, the relation that Cai and Paige define for representing distributive functions does not have the subsumption property.

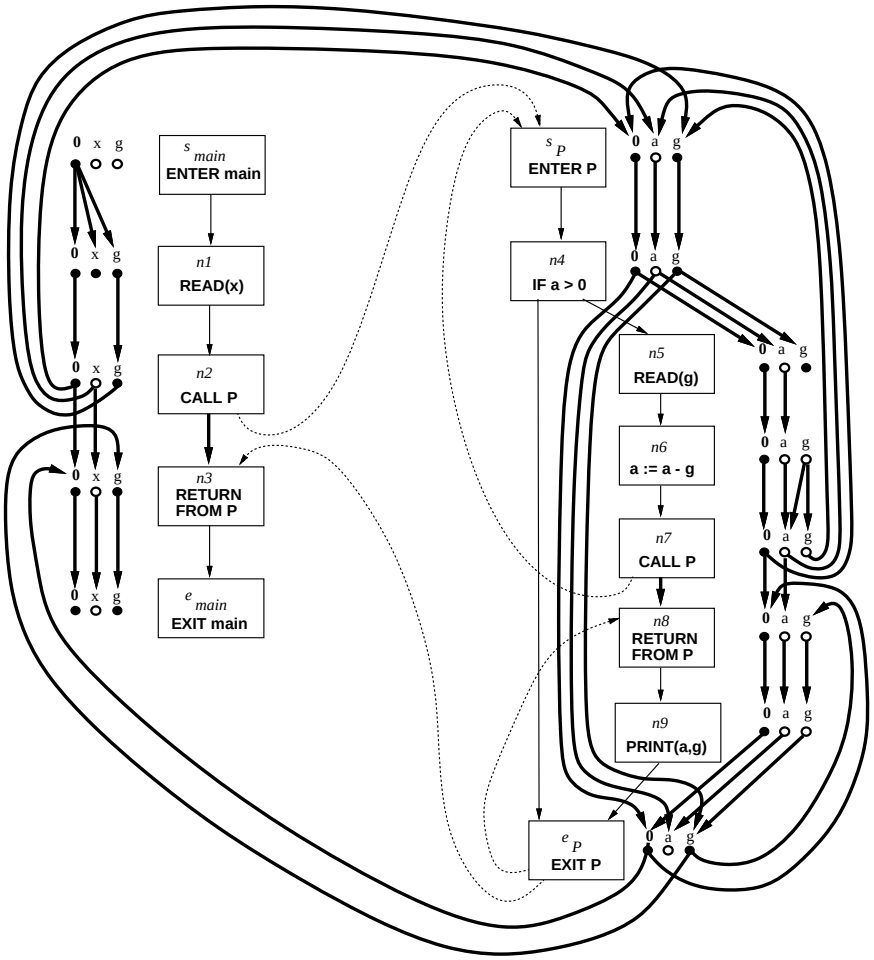


Figure 2. The exploded super-graph that corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 1. Closed circles represent nodes of $G^{\#}_{IP}$ that are reachable along realizable paths from $(s_{main}, \mathbf{0})$. Open circles represent nodes not reachable along such paths.

Theorem 3.8. *Let $G^{\#}_{IP} = (N^{\#}, E^{\#})$ be the exploded super-graph for IFDS problem instance $IP = (G^{*}, D, F, M, \cup)$, and let n be a program point in N^{*}. Then $d \in MVP_n$ iff there is a realizable path in graph $G^{\#}_{IP}$ from node $(s_{main}, \mathbf{0})$ to node (n, d).*

The practical consequence of this theorem is that we can find the meet-over-all-valid-paths solution to *IP* by solving a realizable-path reachability problem in graph $G^{\#}_{IP}$.

Example. In the exploded super-graph shown in Figure 2, which corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 1, closed circles represent nodes that are reachable along realizable paths from $(s_{main}, \mathbf{0})$. Open circles represent nodes not reachable along realizable paths. (For example, note that nodes $(n8, g)$ and $(n9, g)$ are reachable only along non-realizable paths from $(s_{main}, \mathbf{0})$.)

This information indicates the nodes' values in the meet-over-all-valid-paths solution to the dataflow-analysis problem. For instance, in the meet-over-all-valid-paths solution, $MVP_{e_p} = \{ g \}$. (That is, variable g is the only possibly-uninitialized variable just before execution reaches the exit node of procedure p.) In Figure 2, this information can be obtained by determining that there is a realizable path from $(s_{main}, \mathbf{0})$ to (e_p, g), but not from $(s_{main}, \mathbf{0})$ to (e_p, a). □

4. An Efficient Algorithm for the Realizable-Path Reachability Problem

In this section, we present our new algorithm for the realizable-path reachability problem. The algorithm is a dynamic-programming algorithm that tabulates certain kinds of same-level realizable paths. As discussed in Section 5 and the Appendix, the algorithm's running time is polynomial in various parameters of the problem, and it is asymptotically faster than the best previously known algorithm for the problem.

The algorithm, which we call the ***Tabulation Algorithm***, is presented in Figure 3. The algorithm uses the following functions:

- *returnSite*: maps a call node to its corresponding return-site node;
- *procOf*: maps a node to the name of its enclosing procedure;
- *calledProc*: maps a call node to the name of the called procedure;
- *callers*: maps a procedure name to the set of call nodes that represent calls to that procedure.

The Tabulation Algorithm uses a set named PathEdge to record the existence of ***path edges***, which represent a subset of the same-level realizable paths in graph $G^{\#}_{IP}$. In particular, the source of a path edge is always a node of the form (s_p, d_1) such that a realizable path exists from node $(s_{main}, \mathbf{0})$ to (s_p, d_1). In other words, a path edge from (s_p, d_1) to (n, d_2) represents the suffix of a realizable path from node $(s_{main}, \mathbf{0})$ to (n, d_2).

The Tabulation Algorithm uses a set named SummaryEdge to record the existence ***summary edges***, which represent same-level realizable paths that run from nodes of the form (n, d_1), where $n \in Call$, to $(returnSite(n), d_2)$. In terms of the dataflow problem being solved, summary edges represent (partial) information about how the dataflow value after a call depends on the dataflow value before the call.

The Tabulation Algorithm is a worklist algorithm that accumulates sets of path edges and summary edges. The initial set of path edges consists of the single 0-length same-level realizable path of the form $((s_{main}, \mathbf{0}), (s_{main}, \mathbf{0}))$ (see line [2]). On each iteration of the main loop in procedure ForwardTabulateSLRPs (lines [10]-[39]), the algorithm deduces the existence of additional path edges (and summary edges). The configurations that are used by the Tabulation Algorithm to deduce the existence of additional path edges are depicted in Figure 4.

Once it is known that there is a realizable path from $(s_{main}, \mathbf{0})$ to (s_p, d), path edge $((s_p, d), (s_p, d))$ is inserted into WorkList (lines [14]-[16]). (The idea of inserting only relevant $((s_p, d), (s_p, d))$ edges into WorkList is similar to the idea of avoiding unnecessary function applications during abstract interpretation, known as the minimal function-graph approach [14].) In general, a path edge from $(s_{procOf(n)}, d_1)$ to (n, d_2) represents the suffix of a *realizable path* from $(s_{main}, \mathbf{0})$ to (n, d_2).

It is important to note the role of lines [26]-[28] of Figure 3, which are executed only when a new summary edge is discovered:

```
[26]  for each d3 such that ((s_procOf(c), d3), (c, d4)) ∈ PathEdge do
[27]    Propagate(((s_procOf(c), d3), (returnSite(c), d5)))
[28]  od
```

Unlike edges in $E^{\#}$, edges are inserted into SummaryEdge on-the-fly. The purpose of line [27] is to restart the processing that finds same-level realizable paths from $(s_{procOf(c)}, d_3)$ as if summary edge $((c, d_4), (returnSite(c), d_5))$ had been in place all along.

The final step of the Tabulation Algorithm (lines [6]-[8]) is to create values X_n, for each $n \in N^*$, by gathering up the set of nodes associated with n in $G^{\#}_{IP}$ that are targets of path edges discovered by procedure ForwardTabulateSLRPs:

[7] $X_n := \{\, d_2 \in D \mid \exists\, d_1 \in (D \cup \{\, \mathbf{0} \,\}) \text{ such that } ((s_{procOf(n)}, d_1), (n, d_2)) \in \text{PathEdge} \,\}$

As mentioned above, the fact that $((s_{procOf(n)}, d_1), (n, d_2))$ is in PathEdge implies that there is a realizable path from $(s_{main}, \mathbf{0})$ to (n, d_2). Consequently, by Theorem 3.8, when the Tabulation Algorithm terminates, the value of X_n is the value for node n in the meet-over-all-valid-paths solution to *IP*; *i.e.*, for all $n \in N^*$, $X_n = MVP_n$.

```
      declare PathEdge, WorkList, SummaryEdge: global edge set

      algorithm Tabulate(G#_IP)
      begin
[1]     Let (N#, E#) = G#_IP
[2]     PathEdge := { ((s_main, 0), (s_main, 0)) }
[3]     WorkList := { ((s_main, 0), (s_main, 0)) }
[4]     SummaryEdge := ∅
[5]     ForwardTabulateSLRPs()
[6]     for each n ∈ N* do
[7]       X_n := { d_2 ∈ D | ∃ d_1 ∈ (D ∪ { 0 }) such that ((s_procOf(n), d_1), (n, d_2)) ∈ PathEdge }
[8]     od
      end

      procedure Propagate(e)
      begin
[9]     if e ∉ PathEdge then Insert e into PathEdge; Insert e into WorkList fi
      end

      procedure ForwardTabulateSLRPs()
      begin
[10]    while WorkList ≠ ∅ do
[11]      Select and remove an edge ((s_p, d_1), (n, d_2)) from WorkList
[12]      switch n
[13]        case n ∈ Call_p :
[14]          for each d_3 such that ((n, d_2), (s_calledProc(n), d_3)) ∈ E# do
[15]            Propagate(((s_calledProc(n), d_3), (s_calledProc(n), d_3)))
[16]          od
[17]          for each d_3 such that ((n, d_2), (returnSite(n), d_3)) ∈ (E# ∪ SummaryEdge) do
[18]            Propagate(((s_p, d_1), (returnSite(n), d_3)))
[19]          od
[20]        end case
[21]        case n = e_p :
[22]          for each c ∈ callers(p) do
[23]            for each d_4, d_5 such that ((c, d_4), (s_p, d_1)) ∈ E# and ((e_p, d_2), (returnSite(c), d_5)) ∈ E# do
[24]              if ((c, d_4), (returnSite(c), d_5)) ∉ SummaryEdge then
[25]                Insert ((c, d_4), (returnSite(c), d_5)) into SummaryEdge
[26]                for each d_3 such that ((s_procOf(c), d_3), (c, d_4)) ∈ PathEdge do
[27]                  Propagate(((s_procOf(c), d_3), (returnSite(c), d_5)))
[28]                od
[29]              fi
[30]            od
[31]          od
[32]        end case
[33]        case n ∈ (N_p − Call_p − { e_p }) :
[34]          for each (m, d_3) such that ((n, d_2), (m, d_3)) ∈ E# do
[35]            Propagate(((s_p, d_1), (m, d_3)))
[36]          od
[37]        end case
[38]      end switch
[39]    od
      end
```

Figure 3. The Tabulation Algorithm determines the meet-over-all-valid-paths solution to *IP* by determining whether certain same-level realizable paths exist in $G^{\#}_{IP}$.

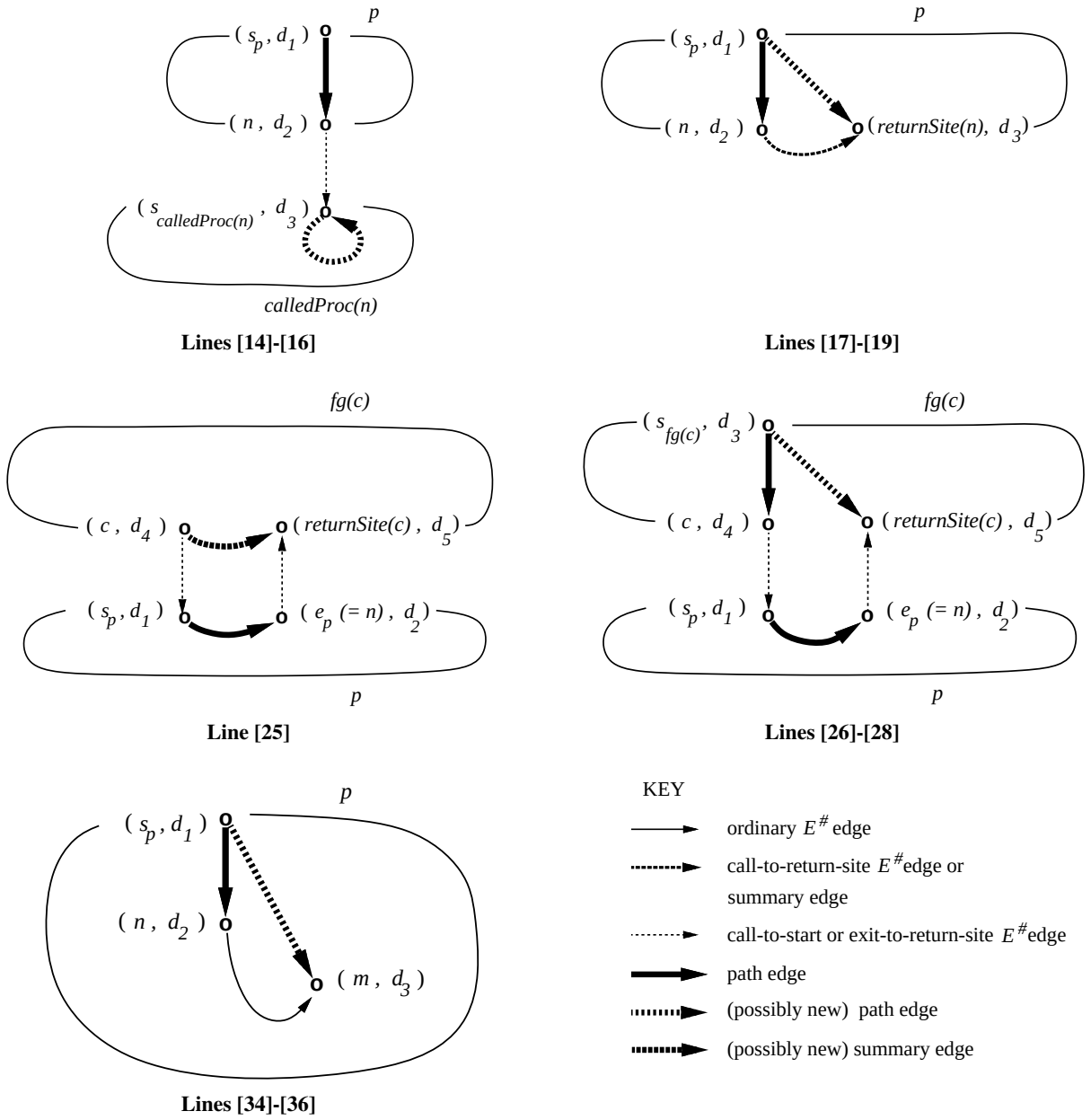


Figure 4. The above five diagrams show the situations handled in lines [14]-[16], [17]-[19], [25], [26]-[28], and [34]-[36] of the Tabulation Algorithm.

Theorem 4.1. (Correctness of the Tabulation Algorithm.) *The Tabulation Algorithm always terminates, and upon termination,* $X_n = MVP_n$, *for all* $n \in N^*$.

5. The Running Time of the Tabulation Algorithm

In discussing the running time of the Tabulation Algorithm, we will use the name of a set to denote the set's size. For example, we will use *Call*, rather than |*Call*|, to denote the number of call nodes in graph G^*. (We will make two small deviations from this convention, using N and E to stand for $|N^*|$ and $|E^*|$, respectively.)

The running time of the Tabulation Algorithm varies depending on what class of dataflow-analysis problems it is applied to. We have already mentioned the locally separable problems; it is also useful to define the class of h-sparse problems:

Definition 5.1. A problem is ***h-sparse*** if all problem instances have the following property: For each function on an ordinary intraprocedural edge or a call-to-return-site edge, the total number of edges in the function's representation relation that emanate from the non-**0** nodes is at most hD. □

In general, when the nodes of the control-flow graph represent individual statements and predicates (rather than basic blocks), and there is no aliasing, we expect most distributive problems to be h-sparse (with $h \ll D$): each statement changes only a small portion of the execution state, and accesses only a small portion of the state as well. The dataflow functions, which are abstractions of the statements' semantics, should therefore be "close to" the identity function, and thus their representation relations should have roughly D edges.

Example. When the nodes of the control-flow graph represent individual statements and predicates, and there is no aliasing, every instance of the possibly-uninitialized-variables problem is 2-sparse. The only non-identity dataflow functions are those associated with assignment statements. The outdegree of every non-**0** node in the representation relation of such a function is at most two: a variable's initialization status can affect itself and at most one other variable, namely the variable assigned to. □

The following table summarizes how the Tabulation Algorithm behaves (in terms of worst-case asymptotic running time) for six different classes of problems:

Class of functions	Graph-theoretic characterization of the dataflow functions' properties	Asymptotic running time	
		Intraprocedural problems	Interprocedural problems
Distributive	Up to $O(D^2)$ edges/rep.-relation	$O(ED^2)$	$O(ED^3)$
h-sparse	At most $O(hD)$ edges/rep.-relation	$O(hED)$	$O(Call\ D^3 + hED^2)$
(Locally) separable	Component-wise dependences	$O(ED)$	$O(ED)$

Table 5.2. Asymptotic running time of the Tabulation Algorithm for six different classes of dataflow-analysis problems.

The details of the analysis of the running time of the Tabulation Algorithm on distributive problems are given in the Appendix. The bounds for the other five classes of problems follow from simplifications of the argument given there.

6. Experimental Results

We have carried out a preliminary study to determine the feasibility of the Tabulation Algorithm and to compare its accuracy and time requirements with those of the naive reachability algorithm that considers all paths rather than just the interprocedurally realizable paths. The two algorithms were implemented in C and used with a front end that analyzes a C program and generates the corresponding exploded super-graph for the possibly-uninitialized-variables problem. (The current implementation of the front end does not account for aliases due to pointers.)

The study used four example C programs: *struct-beauty*, the "beautification" phase of the Unix *struct* program [2]; *twig*, a code-generator generator [1]; *ratfor*, a preprocessor that converts a structured Fortran dialect to standard Fortran [15]; and *C-parser*, a *lex*/*yacc*-generated parser for C. Tests were carried out on a Sun SPARCstation 10 Model 30 with 32 MB of RAM.

The following table gives information about the source code (lines of C, *lex*, and *yacc*) and the parameters that characterize the size of the control-flow graphs and the exploded super graph.

Example	Lines of source code	CFG statistics				$G^{\#}$ statistics			
		P	$Call$	N	E	D	$N^{\#}$	$E^{\#}$	Density
struct-beauty	897	36	214	2188	2860	90	183.9K	220.6K	8%
C-parser	1224	48	78	1637	1992	70	104.4K	112.4K	5%
ratfor	1345	52	266	2239	2991	87	179.5K	217.7K	10%
twig	2388	81	221	3692	4439	142	492.2K	561.1K	6%

In practice, most of the $E^{\#}$ edges are of the form $((m, d), (n, d))$, and our implementations take advantage of this to avoid representing all $E^{\#}$ edges explicitly. The last column of the above table (Density) gives the percentage of edges that are represented explicitly in our representation of $G^{\#}$.

The following table compares the cost and accuracy of the Tabulation Algorithm and the naive algorithm. The running times are "user cpu-time + system cpu-time"; in each case, the time reported is the average of ten executions.

Example	Tabulation Algorithm		Naive Algorithm	
	Time (seconds)	Reported uses of possibly-uninitialized variables	Time (seconds)	Reported uses of possibly-uninitialized variables
struct-beauty	4.83 + 0.75	543	1.58 + 0.04	583
C-parser	0.70 + 0.19	11	0.54 + 0.02	127
ratfor	3.15 + 0.58	894	1.46 + 0.04	998
twig	5.45 + 1.20	767	5.04 + 0.11	775

The number of uses of possibly-uninitialized variables reported by the Tabulation Algorithm ranges from 9% to 99% of those reported by the naive algorithm. Because the possibly-uninitialized-variables problem is 2-sparse, the asymptotic costs of the Tabulation Algorithm and the naive algorithm are $O(Call\ D^3 + ED^2)$ and $O(ED)$, respectively. In these examples, D ranges from 70 to 142; however, the penalty for obtaining the more precise solutions ranges from 1.3 to 3.4. Therefore, this preliminary experiment indicates that the extra precision of meet-over-all-valid-paths solutions to interprocedural dataflow-analysis problems can be obtained by the Tabulation Algorithm with acceptable cost.

7. Related Work

Previous Interprocedural Dataflow-Analysis Frameworks

The IFDS framework is based on earlier interprocedural dataflow-analysis frameworks defined by Sharir and Pnueli [25] and Knoop and Steffen [17]. It is basically the Sharir-Pnueli framework with three modifications:

(i) The dataflow functions are restricted to be distributive functions;
(ii) The dataflow domain is restricted to be a subset domain 2^D, where D is a finite set;
(iii) The edge from a call node to the corresponding return-site node can have an associated dataflow function.

Conditions (i) and (ii) are restrictions that make the IFDS framework less general than the full Sharir-Pnueli framework. Condition (iii), however, generalizes the Sharir-Pnueli framework and permits it to cover programming languages in which recursive procedures have local variables and parameters (which the Sharir-Pnueli framework does not). A different generalization to handle recursive procedures with local variables and parameters was proposed by Knoop and Steffen [17]; for distributive problems, condition (iii) also generalizes the Knoop-Steffen extension.

The IFDS problems can be solved by a number of previous algorithms, including the "elimination", "iterative", and "call-strings" algorithms given by Sharir and Pnueli [25] and the algorithm of Cousot and Cousot [9]. However, for general IFDS problems both the iterative and call-strings algorithms can

take exponential time in the worst case. Knoop and Steffen give an algorithm similar to Sharir and Pnueli's "elimination" algorithm [17]. The efficiencies of the Sharir-Pnueli and Knoop-Steffen elimination algorithms depend, among other things, on the way functions are represented. No representations are discussed in [25] and [17]. However, even if representation relations (as defined in Section 3.1) are used, because the Sharir-Pnueli and Knoop-Steffen algorithms manipulate functions as a whole, rather than element-wise, for distributive and h-sparse problems, they are not as efficient as the Tabulation Algorithm.

Dataflow Analysis via Graph Reachability and Pointwise Computation of Fixed Points

Our work shows that a large subclass of the problems in the Sharir-Pnueli and Knoop-Steffen frameworks can be posed as graph-reachability problems. Other work on solving dataflow-analysis problems by reducing them to reachability problems has been done by Kou [19] and Cooper and Kennedy [6,7]. In each case a dataflow-analysis problem is solved by first building a graph—derived from the program's flow graph and the dataflow functions to be solved—and then performing a reachability analysis on the graph by propagating simple marks. (This contrasts with standard iterative techniques, which propagate sets of values over the flow graph.)

Kou's paper addresses only intraprocedural problems. (Although he only discusses the live-variable problem, his ideas immediately carry over to all the separable intraprocedural problems.) Cooper and Kennedy show how certain flow-insensitive interprocedural dataflow-analysis problems can be converted to reachability problems. Because they deal only with flow-insensitive problems, the solution method involves ordinary reachability rather than the more difficult question of reachability along realizable paths.

Graph reachability can also be thought of as an implementation of the pointwise computation of fixed points, which has been studied by Cai and Paige [3] and Nielson and Nielson [22,21]. Cai and Paige use the technique for compiling programs written in a very-high-level language (SQ+) into efficient executable code. This suggests that it might be possible to express the problem of finding meet-over-all-valid-paths solutions to IFDS problems as an SQ+ fixed-point program and then automatically compile it into an implementation that achieves the bounds established in this paper (*i.e.*, into the Tabulation Algorithm).

Nielson and Nielson investigated bounds on the cost of a general fixed-point-finding algorithm by computing the cost as "(# of iterations) $\times$ (cost per iteration)". Their main contribution was to give formulas for bounding the number of iterations based on properties of both the functional and the domain in which the fixed-point is computed. Their formula for "strict and additive" functions can be adapted to our context of (non-strict) distributive functions, and used to show that the number of iterations of the Tabulation Algorithm is at most ND^2. The cost of a single iteration can be $O(Call\ D^2 + kD^2)$, where k is the maximum outdegree of a node in the control-flow graph. Thus, this approach gives a bound for the total cost of the Tabulation Algorithm of $O((ND^2) \times (Call\ D^2 + kD^2))$ $= O(Call\ ND^4 + kND^4)$, which compares unfavorably with our bound of $O(ED^3)$.

In contrast, the bound that we have presented for the cost of the Tabulation Algorithm is obtained by breaking the cost of the algorithm into three contributing aspects and bounding the *total* cost of the operations performed for each aspect (see the Appendix).

Flow-Sensitive Side-Effect Analysis

Callahan investigated two flow-sensitive side-effect problems: must-modify and may-use [5]. The must-modify problem is to identify, for each procedure p, which variables must be modified during a call on p; the may-use problem is to identify, for each procedure p, which variables may be used before being modified during a call on p. Callahan's method involves building a *program summary graph*, which consists of a collection of graphs that represent the intraprocedural reaching-definitions information between start, exit, call, and return-site nodes—together with interprocedural linkage information.

Although the must-modify and may-use problems are not IFDS problems as defined in Definition 2.4, they can be viewed as problems closely related to the IFDS problems. The basic difference is that

IFDS problems summarize what must be true at a program point in *all calling contexts*, while the must-modify and may-use problems summarize the effects of a procedure *isolated from its calling contexts*. Consequently, Callahan's problems involve valid paths from the individual procedures' start nodes rather than just the start node of the main procedure. The must-modify problem is actually a "*same-level*-valid-path" problem rather than a "valid-path" problem; the must-modify value for each procedure involves only the same-level valid paths from the procedure's start node to its exit node. These observations suggest that Callahan's problems can be thought of as examples of problems in two more general *classes* of problems: a class of distributive valid-path problems, and a class of distributive same-level valid-path problems.

The method utilized in the present paper is to convert distributive valid-path dataflow-analysis problems into realizable-path reachability problems in an exploded super-graph. By transformations analogous to the one given in Section 3,

(i) the distributive valid-path problems can be posed as realizable-path problems;
(ii) the distributive same-level valid-path problems can be posed as same-level realizable-path problems.

In particular, the may-use problem is a locally separable problem in class (i); the must-modify problem is a locally separable problem in class (ii).

The payoff from adopting this generalized viewpoint is that, with only slight modifications, the Tabulation Algorithm can be used to solve *all* problems in the above two classes (*i.e.*, distributive and h-sparse problems, as well as the locally separable ones). The modified algorithms have the same asymptotic running time as the Tabulation Algorithm. In particular, for the locally separable problems—such as must-modify and may-use—the running time is bounded by $O(ED)$. This is an asymptotic improvement over the algorithms given by Callahan: the worst-case cost for building the program summary graph is $O(D \sum_p Call_p E_p)$; given the program summary graph, the worst-case cost for computing must-modify or may-use is $O(D \sum_p Call_p^2)$.

Appendix: The Running Time of the Tabulation Algorithm

In this section, we present a derivation of the bound given in Table 5.2 for the cost of the Tabulation Algorithm on distributive problems.

Instead of calculating the worst-case cost-per-iteration of the loop on lines [10]-[39] of Figure 3 and multiplying by the number of iterations, we break the cost of the algorithm down into three contributing aspects and bound the *total* cost of the operations performed for each aspect. In particular, the cost of the Tabulation Algorithm can be broken down into

(i) the cost of worklist manipulations,
(ii) the cost of installing summary edges at call sites (lines [21]-[32] of Figure 3), and
(iii) the cost of "closure" steps (lines [13]-[20] and [33]-[37] of Figure 3).

Because a path edge can be inserted into WorkList at most once, the cost of each worklist-manipulation operation can be charged to either a summary-edge-installation step or a closure step; thus, we do not need to provide a separate accounting of worklist-manipulation costs.

The Tabulation Algorithm can be understood as $k + 1$ simultaneous *semi-dynamic multi-source reachability problems*—one per procedure of the program. For each procedure p, the sources—which we shall call ***anchor sites***—are the $D + 1$ nodes in $N^{\#}$ of the form (s_p, d). The edges of the multi-source reachability problem associated with p are

$$\{\ ((m, d_1), (n, d_2)) \in E^{\#} \mid m, n \in N_p \text{ and } (m, n) \text{ is an intraprocedural edge or a call-to-return-site edge }\}$$
$$\cup\{\ ((m, d_1), (n, d_2)) \in \text{SummaryEdge} \mid m \in Call_p)\ \}.$$

In other words, the graph associated with procedure p is the "exploded flow graph" of procedure p, augmented with summary edges at the call sites of p. The reachability problems are semi-dynamic (insertions only) because in the course of the algorithm, new summary edges are added, but no summary edges (or any other edges) are ever removed.

We first turn to the question of computing a bound on the cost of installing summary edges at call sites (lines [21]-[32] of Figure 3). To express this bound, it is useful to introduce a quantity B that represents the "bandwidth" for the transmission of dataflow information between procedures: In particular, B is the maximum value for all call-to-start edges and exit-to-return-site edges of (i) the maximum outdegree of a non-**0** node in a call-to-start edge's representation relation; (ii) the maximum indegree of a non-**0** node in an exit-to-return-site edge's representation relation. (In the worst case, B is D, but it is typically a small constant, and for many problems it is 1.)

For each summary edge $((c, d_4), (returnSite(c), d_5))$, the conditional statement on lines [24]-[29] will be executed some number of times (on different iterations of the loop on lines [10]-[39]). In particular, line [24] will be executed every time the Tabulation Algorithm finds a three-edge path of the form

$$[((c, d_4), (s_p, d_1)),\ ((s_p, d_1), (e_p, d_2)),\ ((e_p, d_2), (returnSite(c), d_5))] \qquad (\dagger)$$

as shown in the diagram marked "Line [25]" of Figure 4.

When we consider the set of all summary edges at a given call site c: $\{\ ((c, d_4), (returnSite(c), d_5))\ \}$, the executions of line [24] can be placed in three categories:

$d_4 \neq \mathbf{0}$ and $d_5 \neq \mathbf{0}$
: There are at most D^2 choices for a $<d_4, d_5>$ pair, and for each such pair at most B^2 possible three-edge paths of the form (†).

$d_4 = \mathbf{0}$ and $d_5 \neq \mathbf{0}$
: There are at most D choices for d_5 and for each such choice at most BD possible three-edge paths of the form (†).

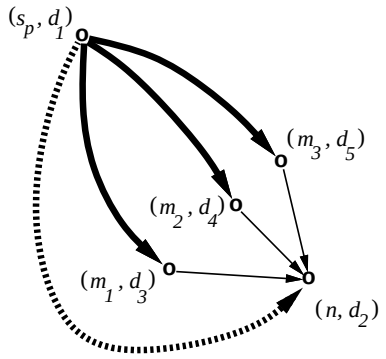
$d_4 = \mathbf{0}$ and $d_5 = \mathbf{0}$
: There is only one possible three-edge path of the form (†).

Thus, the total cost of all executions of line [24] is bounded by $O(Call\ B^2 D^2)$.

Because of the test on line [24], the code on lines [25]-[28] will be executed exactly *once* for each possible summary edge. In particular, for each summary edge the cost of the loop on lines [26]-[28] is bounded by $O(D)$. Since the total number of summary edges is bounded by $Call\ D^2$, the total cost of

lines [25]-[28] is $O(Call\ D^3)$. Thus, the total cost of installing summary edges during the Tabulation Algorithm is bounded by $O(Call\ B^2D^2 + Call\ D^3)$.

To bound the total cost of the closure steps, the essential observation is that there are only a certain number of "attempts" the Tabulation Algorithm makes to "acquire" a path edge $((s_p, d_1), (n, d_2))$. The first attempt is successful—and $((s_p, d_1), (n, d_2))$ is inserted into PathEdge; all remaining attempts are redundant (but seem unavoidable). In particular, in the case of a node $n \notin Ret$, the only way the Tabulation Algorithm can obtain a path edge $((s_p, d_1), (n, d_2))$ is when there are one or more two-edge paths of the form $[((s_p, d_1), (m, d)), ((m, d), (n, d_2))]$, where $((s_p, d_1), (m, d))$ is in PathEdge and $((m, d), (n, d_2))$ is in $E^{\#}$, as depicted below:



Consequently, for a given anchor site (s_p, d_1), the cost of the closure steps involved in acquiring path edge $((s_p, d_1), (n, d_2))$ can be bounded by $indegree(n, d_2)$. For distributive problems, the representation relation of the function on an ordinary intraprocedural edge or a call-to-return-site edge can contain up to $O(D^2)$ edges. Thus, for each anchor site, the total cost of acquiring all its outgoing path edges can be bounded by

$$O\Big(\sum_{(n,d) \in N_p^{\#} \text{and } n \notin Ret} indegree(n, d)\Big) = O(E_p D^2).$$

The accounting for the case of a node $n \in Ret$ is similar. The only way the Tabulation Algorithm can obtain a path edge $((s_p, d_1), (n, d_2))$ is when there is an edge in PathEdge of the form $((s_p, d_1), (m, d))$ and either there is an edge $((m, d), (n, d_2))$ in $E^{\#}$ or an edge $((m, d), (n, d_2))$ in SummaryEdge. In our cost accounting, we will pessimistically assume that each node (n, d_2), where $n \in Ret$, has the maximum possible number of incoming summary edges, namely D. Because there are at most $Call_p D$ nodes of $N_p^{\#}$ of the form (n, d_2), where $n \in Ret$, for each anchor site (s_p, d_1) the total cost of acquiring path edges of the form $((s_p, d_1), (n, d_2))$ is

$$O\Big(\sum_{(n,d_2) \in N_p^{\#} \text{and } n \in Ret} (indegree(n, d_2) + summary\text{-}edge\text{-}indegree(n, d_2))\Big) = O(Call_p D^2).$$

Therefore we can bound the total cost of the closure steps performed by the Tabulation Algorithm as follows:

$$\begin{aligned}
\text{Cost of closure steps} &= \sum_p (\#\text{ anchor sites}) \times O(Call_p D^2 + E_p D^2) \\
&= O(D^3 \sum_p (Call_p + E_p)) \\
&= O(D^3(Call + E)) \\
&= O(ED^3).
\end{aligned}$$

Thus, the total running time of the Tabulation Algorithm is bounded by $O(Call\ B^2D^2 + ED^3)$.

In the full paper, we will present a variation on the argument given above that improves the bound to $O(Call\ BD^2 + ED^3)$. Because $Call \le E$ and $B \le D$, this simplifies to $O(ED^3)$, the bound reported in Table 5.2.

References

1. Aho, A.V., Ganapathi, M., and Tjiang, S.W.K., "Code generation using tree matching and dynamic programming," *ACM Trans. Program. Lang. Syst.* **11**(4) pp. 491-516 (October 1989).
2. Baker, B., "An algorithm for structuring flowgraphs," *J. ACM* **24**(1) pp. 98-120 (January 1977).
3. Cai, J. and Paige, R., "Program derivation by fixed point computation," *Science of Computer Programming* **11** pp. 197-261 (1988/89).
4. Callahan, D., Cooper, K.D., Kennedy, K., and Torczon, L., "Interprocedural constant propagation," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction,* (Palo Alto, CA, June 25-27, 1986)*, ACM SIGPLAN Notices* **21**(7) pp. 152-161 (July 1986).
5. 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).
6. Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," *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. 57-66 (July 1988).
7. Cooper, K.D. and Kennedy, K., "Fast interprocedural alias analysis," pp. 49-59 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
8. Cousot, P. and Cousot, R., "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," pp. 238-252 in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages,* (Los Angeles, CA, January 17-19, 1977), ACM, New York, NY (1977).
9. Cousot, P. and Cousot, R., "Static determination of dynamic properties of recursive procedures," pp. 237-277 in *Formal Descriptions of Programming Concepts,* (IFIP WG 2.2, St. Andrews, Canada, August 1977), ed. E.J. Neuhold, North-Holland, New York, NY (1978).
10. Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler,* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).
11. Giegerich, R., Moncke, U., and Wilhelm, R., "Invariance of approximative semantics with respect to program transformation," pp. 1-10 in *Informatik-Fachberichte 50*, Springer-Verlag, New York, NY (1981).
12. Grove, D. and Torczon, L., "Interprocedural constant propagation: A study of jump function implementation," pp. 90-99 in *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation,* (Albuquerque, NM, June 23-25, 1993), ACM, New York, NY (1993).
13. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).
14. Jones, N.D. and Mycroft, A., "Data flow analysis of applicative programs using minimal function graphs," pp. 296-306 in *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages,* (St. Petersburg, FL, Jan. 13-15, 1986), ACM, New York, NY (1986).
15. Kernighan, B.W., "Ratfor – A preprocessor for a rational Fortran," *Software – Practice & Experience* **5**(4) pp. 395-406 (1975).
16. 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).
17. 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).
18. Knoop, J. and Steffen, B., "Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework," Bericht Nr. 9309, Institut fuer Informatik und Praktische Mathematik, Christian-Albrechts-Universitaet zu Kiel, Kiel, Germany (April 1993).
19. Kou, L.T., "On live-dead analysis for global data flow problems," *Journal of the ACM* **24**(3) pp. 473-483 (July 1977).
20. 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).
21. Nielson, F. and Nielson, H.R., "Finiteness conditions for fixed point iteration," in *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming,* (San Francisco, CA, June 22-24 1992), ACM, New York, NY (1992).
22. Nielson, H.R. and Nielson, F., "Bounded fixed point iteration," pp. 71-82 in *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages,* (Albuquerque, NM, January 1992), ACM, New York, NY (1992).
23. Reps, T., Sagiv, M., and Horwitz, S., "Interprocedural dataflow analysis via graph reachability," Technical Report 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark (April 1994).
24. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," Report TOPPS D-214, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark (June 1994). (Submitted for publication.)
25. 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).