

Interprocedural Dataflow Analysis via Graph Reachability

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

This paper shows how a large class of interprocedural dataflow-analysis problems can be solved precisely in polynomial time. The only restrictions are that the set of dataflow facts is a finite set, and that the dataflow functions distribute over the confluence operator (either union or intersection). This class of problems includes—but is not limited to—the classical separable problems (also known as “gen/kill” or “bit-vector” problems)—*e.g.*, reaching definitions, available expressions, and live variables. In addition, the class of problems that our techniques handle includes many non-separable problems, including truly-live variables, copy constant propagation, and possibly-uninitialized variables.

A novel aspect of our approach is that an interprocedural dataflow-analysis problem is transformed into a special kind of graph-reachability problem (reachability along *interprocedurally realizable paths*). The paper presents three polynomial-time algorithms for the realizable-path reachability problem: an exhaustive version, a second exhaustive version that may be more appropriate in the incremental and/or interactive context, and a demand version. The first and third of these algorithms are asymptotically faster than the best previously known realizable-path reachability algorithm.

An additional benefit of our techniques is that they lead to improved algorithms for two other kinds of interprocedural-analysis problems: interprocedural flow-sensitive side-effect problems (as studied by Callahan) and interprocedural program slicing (as studied by Horwitz, Reps, and Binkley).

CR Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors – *compilers, optimization*; E.1 [Data Structures] – *graphs*; F.2.2 [Analysis of Algorithms and Problem Complexity]: Complexity of Algorithms, Nonnumerical Algorithms and Problems – *computations on discrete structures*; G.2.2 [Discrete Mathematics]: Graph Theory – *graph algorithms*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: demand dataflow analysis, distributive dataflow framework, flow-sensitive side-effect analysis, function representation, graph reachability, interprocedural dataflow analysis, interprocedural program slicing, interprocedurally realizable path, interprocedurally valid path, meet-over-all-valid-paths solution

1. Introduction

This paper shows how to find precise (*i.e.*, meet-over-all-valid-paths) solutions to a large class of interprocedural dataflow-analysis problems in polynomial time. We give several efficient algorithms for solving a large subclass of interprocedural dataflow-analysis problems in the framework proposed by Sharir and Pnueli [31]. Our techniques also apply to the extension of the Sharir-Pnueli framework proposed by Knoop and Steffen, which covers programs in which recursive procedures may have local variables and call-by-value parameters [21].

Our techniques apply to all problem instances in the above-mentioned interprocedural frameworks in which the set of dataflow facts D is a finite set, and where the dataflow functions (which are in $2^D \rightarrow 2^D$) distribute over the confluence operator (either union or intersection, depending on the problem). This class of problems—which we will call the *interprocedural, finite, distributive, subset problems*, or *IFDS problems*, for short—includes, but is not limited to, the classical separable problems (also known as “gen/kill” or “bit-vector” problems)—*e.g.*, reaching definitions, available expressions, and live variables; however, the class also includes many non-separable problems, including truly-live variables [12], copy constant propagation [11, pp. 660], and possibly-uninitialized variables. (These problems are defined in Appendix A.)

[†]On sabbatical leave from the University of Wisconsin–Madison, Madison, WI, USA.

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

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants CCR-8958530 and CCR-9100424, 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), by the Air Force Office of Scientific Research under grant AFOSR-91-0308, and by a grant from Xerox Corporate Research.

Authors’ address: Datalogisk Institut, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark.

Electronic mail: {reps, sagiv, horwitz}@diku.dk.

Our approach involves transforming a dataflow problem into a special kind of graph-reachability problem (reachability along *interprocedurally realizable paths*), which is then solved by an efficient graph algorithm. 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. We show that the problem of finding a precise (i.e., meet-over-all-valid-paths) solution to an instance of an IFDS problem can be solved by solving an instance of a realizable-path reachability problem.

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 to ***realizable-path reachability problems***.
- In Sections 4, 5, and 6, we present three new polynomial-time algorithms for the realizable-path reachability problem. Two of the algorithms are ***asymptotically faster*** than the best previously known algorithm for the problem [18]. One of these algorithms permits demand interprocedural dataflow analysis to be carried out. The remaining algorithm, although asymptotically not as fast in all cases as the other two, may be the preferred method in the incremental and/or interactive context (see below).
- The three realizable-path reachability algorithms are ***adaptive***, with asymptotically better performance when they are applied to common kinds of problem instances that have restricted form. For example, there is an asymptotic improvement in the algorithms’ performance for the common case of “locally separable” problems—the interprocedural versions of the classical separable problems.
- Imprecise 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 versus 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 distributive case, the “penalty” is a factor of $|D|$, where D is the set underlying the dataflow lattice 2^D : the running time of our realizable-path reachability algorithm is $O(E|D|^3)$, where E is the size of the program’s control-flow graph, whereas all-paths reachability solutions can be found in time $O(E|D|^2)$. However, in the important special case of locally separable problems, there is no penalty at all—both kinds of solutions can be obtained in time $O(E|D|)$.
- In Section 6, we present a ***demand algorithm*** for answering individual realizable-path reachability queries. With this algorithm, information from previous queries can be accumulated and used to compute the answers to later queries, thereby further reducing the amount of work performed to answer subsequent queries. Furthermore, the total cost of any request sequence that poses all possible queries is no more than the cost of a single run of the exhaustive realizable-path reachability algorithm from Section 4.
- It is possible to perform a kind of ***“compression transformation”*** that turns an IFDS problem into a compressed problem whose size is related to the number of call sites in the program. Compression is particularly useful when dealing with the incremental and/or interactive context, where the program undergoes a sequence of small changes, after each of which dataflow information is to be reported. The advantage of the compression technique stems from two factors:
 - (i) It may be possible to solve the compressed version more efficiently than the original uncompressed problem. The speed-up factor depends on the total number of “program points” and the total number of call sites.
 - (ii) It is possible to reuse the compressed structure for each unchanged procedure, and thus only changed procedures need to be re-compressed. In the incremental and/or interactive context changes are ordinarily made to no more than a small percentage of a program’s procedures.
- Callahan has given algorithms for several “interprocedural flow-sensitive side-effect problems”[6]. As we will see in Section 7, these problems are (from a certain technical standpoint) of a slightly different character from the IFDS dataflow-analysis problems. However, with small adaptations the algorithms from Sections 4, 5, and 6 can be applied to these problems. Two of the algorithms are ***asymptotically faster*** than the algorithm given by Callahan. In addition, each of our algorithms handles a natural generalization of Callahan’s problems (which are locally separable problems) to a class of distributive flow-sensitive side-effect problems.
- 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 [18]. The realizable-path reachability algorithms described in this paper yield ***improved interprocedural-slicing algorithms***—ones whose running times are asymptotically faster than the Horwitz-Reps-Binkley algorithm.

The remainder of the paper is organized as follows: Section 2 defines the IFDS framework for distributive interprocedural dataflow-analysis problems. Section 3 shows how the problems in the IFDS framework can be formulated as graph-reachability problems. Section 4 presents the first of our three algorithms for the realizable-path reachability problem. Section 5 discusses an alternative algorithm for the realizable-path reachability problem that, while asymptotically slower than the algorithm from Section 4, may be the algorithm of choice in certain situations. Section 6 discusses demand interprocedural dataflow analysis, and presents an efficient algorithm for the problem. Section 7 describes how our techniques can be adapted to yield new algorithms for interprocedural flow-sensitive side-effect analysis and interprocedural program slicing. Section 8 discusses related work. Appendix A defines several dataflow-analysis problems that are distributive but not separable. Appendix B provides an index of the terms and notation used in the paper.

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 [31], with an extension similar to the one given by Knoop and Steffen in order to handle programs in which recursive procedures may have local variables and parameters [21]. These frameworks generalize Kildall’s concept of the “meet-over-all-paths” solution of an *intraprocedural* dataflow-analysis problem [20] to the “meet-over-all-valid-paths” solution of an *interprocedural* 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.

To specify the IFDS framework, we need the following definitions of graphs used to represent programs:

Definition 2.1. Define the set $\{G_0, G_1, \dots, G_k\}$ to be a collection of **flow graphs**, where each G_p is a directed graph corresponding to a procedure of the program, and $G_p = (N_p, E_p, s_p, e_p)$. The **node sets** N_p , $p \in \{0, \dots, k\}$, are pairwise disjoint, as are the **edge sets** E_p . Node s_p is the unique **start node** of G_p ; node e_p is the unique **exit node** of G_p . Every procedure call contributes two nodes: a **call** node and a **return-site** node. $Call_p \subseteq N_p$ and $Ret_p \subseteq N_p$ are the sets of G_p ’s call and return-site nodes, respectively.

G_p ’s edges are divided into two disjoint subsets: $E_p = E_p^0 \cup E_p^1$; an edge $(m, n) \in E_p^0$ is an ordinary control-flow edge—it represents a direct transfer of control from one node to another; an edge $(m, n) \in E_p^1$ iff m is a call node and n is the corresponding return-site node. (Observe that node n is *within* p as well—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 start nodes have no incoming edges, and that a return-site node in any G_p graph has exactly one incoming edge: the E_p^1 edge from the corresponding call node.

We define the **super-graph** G^* as follows: $G^* = (N^*, E^*, s_{main})$, where $N^* = \bigcup_{p \in \{0, \dots, k\}} N_p$ and $E^* = E^0 \cup E^1 \cup E^2$, where $E^0 = \bigcup_{p \in \{0, \dots, k\}} E_p^0$ is the collection of all ordinary control-flow edges, $E^1 = \bigcup_{p \in \{0, \dots, k\}} E_p^1$ is the collection of all edges linking call nodes with their respective return-site nodes, and an edge $(m, n) \in E^2$ represents either a **call edge** or a **return edge**. Edge $(m, n) \in E^2$ is a call edge iff m is a call node and n is the start node of the called procedure; edge $(m, n) \in E^2$ is a return edge iff m is an exit node of some procedure p and n is a return-site node for a call on p . A call edge (m, s_p) and return edge (e_q, n) **correspond** to each other if $p = q$ and $(m, n) \in E^1$.

We identify four special classes of nodes in super-graph G^* :

- **Call**, the set of all call nodes, defined as $\bigcup_{p \in \{0, \dots, k\}} Call_p$;
- **Ret**, the set of all return-site nodes, defined as $\bigcup_{p \in \{0, \dots, k\}} Ret_p$;
- **Start**, the set of all start nodes, defined as $\{s_p \mid p \in \{0, \dots, k\}\}$;
- **Exit**, the set of all exit nodes, defined as $\{e_p \mid p \in \{0, \dots, k\}\}$.

Finally, we define the following functions:

- **source**: $E^* \rightarrow N^*$, where $source(m, n) =_{df} m$.
- **target**: $E^* \rightarrow N^*$, where $target(m, n) =_{df} n$.

- For $i \in \{0, 1, 2\}$, $\text{succ}^i: N^* \rightarrow 2^{N^*}$, where $\text{succ}^i(m) =_{df} \{n \mid (m, n) \in E^i\}$;
- For $i \in \{0, 1, 2\}$, $\text{pred}^i: N^* \rightarrow 2^{N^*}$, where $\text{pred}^i(n) =_{df} \{m \mid (m, n) \in E^i\}$;
- $\text{fg}: N^* \rightarrow \{0, \dots, k\}$, where $\text{fg}(n) =_{df} p$ iff $n \in N_p$;
- $\text{calledProc}: \text{Call} \rightarrow \{0, \dots, k\}$, where $\text{calledProc}(n) =_{df} p$ iff n represents a call on procedure p ;
- $\text{callers}: \{0, \dots, k\} \rightarrow 2^{\text{Call}}$, where $\text{callers}(p) =_{df} \{n \mid \text{calledProc}(n) = p\}$.

Source and *target* map edges to their endpoints; *pred* and *succ* map nodes to their predecessors and successors, respectively; *fg* maps a node to the index of its corresponding flow graph; *calledProc* maps a call node to the index of the called procedure; *callers* maps a procedure index to the set of call nodes that call that procedure.

□

Example. Figure 1 shows an example program and its super graph G^* . Edges in E^0 are shown using solid arrows; edges in E^1 are shown using bold arrows; edges in E^2 are shown using dotted arrows.

□

```

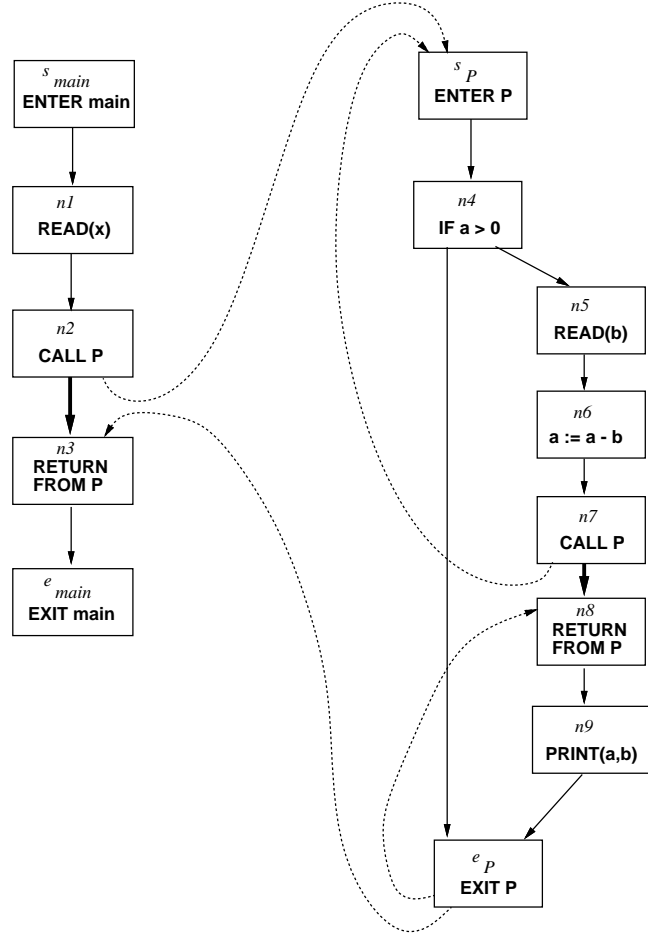
program main
begin
  declare x, y: integer
  read(x)
  call P(x, y)
end

```

```

procedure P(a, b: integer)
begin
  if (a > 0) then
    read(b)
    a := a - b
    call P(a, b)
    print(a, b)
  fi
end

```



(a) Example program

(b) Its super-graph G^*

Figure 1. An example program and its super-graph G^* . Edges in E^0 are shown using solid arrows; edges in E^1 are shown using bold arrows; edges in E^2 are shown using dotted arrows.

Definition 2.2. A *path* of length j from node m to node n is a sequence of j edges, which will be denoted by $[e_1, e_2, \dots, e_j]$ (or by $[m: e_1, e_2, \dots, e_j: n]$ when we wish to emphasize the path’s *endpoints* $m = \text{source}(e_1)$ and $n = \text{target}(e_j)$), such that for all i , $1 \leq i \leq j-1$, $\text{target}(e_i) = \text{source}(e_{i+1})$. The *empty path* from m to m (of length 0) will be denoted by $[m: \varepsilon: m]$. For each $m, n \in N^*$, we denote the set of all paths in G^* from m to n by $\text{path}_{G^*}(m, n)$.

□

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 in the range $[1..|Call|]$. For each such indexed call node c_i , label c_i ’s outgoing E^2 edge $(c_i, s_{\text{calledProc}(c_i)})$ by the symbol “ $(_i$ ”. Label the corresponding return edge $(e_{\text{calledProc}(c_i)}, \text{succ}^1(c_i))$ by the symbol “ $)_i$ ”.

For each $m, n \in N_p$, we denote the set of all *same-level interprocedurally valid paths* in G^* that lead from m to n by $\text{SLIVP}(m, n)$. A path $q \in \text{path}_{G^*}(m, n)$ is in $\text{SLIVP}(m, n)$ iff the sequence of symbols labeling the E^2 edges in the path is a string in the language of balanced parentheses generated from nonterminal *matched* by the following context-free grammar:

$$\begin{aligned} \text{matched} &\rightarrow \text{matched matched} \\ &\quad | \text{ } (_i \text{ matched })_i && \text{for } 1 \leq i \leq |Call| \\ &\quad | \varepsilon \end{aligned}$$

For each $m, n \in N^*$, we denote the set of all *interprocedurally valid paths* in G^* that lead from m to n by $\text{IVP}(m, n)$. A path $q \in \text{path}_{G^*}(m, n)$ is in $\text{IVP}(m, n)$ iff the sequence of symbols labeling the E^2 edges is a string in the language generated from nonterminal *valid* in the following grammar (where *matched* is as defined above):

$$\begin{aligned} \text{valid} &\rightarrow \text{matched } (_i \text{ valid} && \text{for } 1 \leq i \leq |Call| \\ &\quad | \text{ matched} \end{aligned}$$

□

We are primarily interested in $\text{IVP}(s_{\text{main}}, n)$, the set of interprocedurally valid paths from s_{main} to n .

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

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

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

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

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

□

In the formulation of the IFDS dataflow-analysis framework (see Definition 2.4 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 $L(\text{valid})$.

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 nine-tuple:

$$IP = (G^*, D, F, B, T, M, Tr, C, \sqcap)$$

where

- (i) G^* is a super-graph as defined in Definition 2.1.
- (ii) $D = \{ D_0, D_1, \dots, D_k \}$ is a collection of finite sets.
- (iii) $F = \{ F_0, F_1, \dots, F_k \}$ is a collection of sets of distributive functions such that $F_p \subseteq 2^{D_p} \rightarrow 2^{D_p}$.

- (iv) B is a positive constant that bounds the “bandwidth” for the transmission of dataflow information between procedures (see clause (v)).
- (v) $T = \{ T_{i,j} \mid i, j \in \{0, \dots, k\} \}$ is a collection of sets of distributive functions such that $T_{p,q} \subseteq 2^{D_p} \rightarrow 2^{D_q}$ and with the following further restrictions:
 - a. For all $t \in T_{p,q}$ and $x \in D_p$, $|t(\{x\}) - t(\emptyset)| \leq B$;
 - b. For all $t \in T_{p,q}$ and $y \in D_q$, if $y \notin t(\emptyset)$ then $|\{x \mid y \in t(\{x\})\}| \leq B$.
- (vi) $M: (E^0 \cup E^1) \rightarrow \bigcup_{i \in \{0, \dots, k\}} F_i$ is a map from G^* ’s E^0 and E^1 edges to dataflow functions, such that $(m, n) \in E_p$ implies that $M(m, n) \in F_p$.
- (vii) $Tr: E^2 \rightarrow \bigcup_{i,j \in \{0, \dots, k\}} T_{i,j}$ is a map from G^* ’s call and return edges to dataflow functions such that $(m, n) \in E^2$ implies that $Tr(m, n) \in T_{fg(m), fg(n)}$.
- (viii) $C \subseteq D_0$ is a distinguished value associated with node s_{main} of G^* .
- (ix) The meet operator \sqcap is either union or intersection.

□

A few words of explanation to clarify Definition 2.4 are called for. One factor that complicates the definitions of clauses (ii), (iii), (v), (vi), and (vii) is that there is potentially a different dataflow domain 2^{D_p} for each procedure p . Thus, for example, D is a *collection* of finite sets $\{D_0, D_1, \dots, D_k\}$, where each D_p is associated with procedure p . In addition,

- F is a collection of sets of functions; each F_p contains the dataflow functions for procedure p .
- T is also a collection of sets of functions; each $T_{p,q}$ contains the “transfer” functions for mapping between the domain 2^{D_p} of procedure p and the domain 2^{D_q} of procedure q . Typically, the functions of $T_{p,q}$ represent binding changes necessary for transmitting dataflow information from procedure p to procedure q .
- Map M labels each E_p^0 and E_p^1 edge with a function of the appropriate type (*i.e.*, in $2^{D_p} \rightarrow 2^{D_p}$).
- Map Tr labels each E^2 edge with a function of the appropriate type (*i.e.*, in $2^{D_p} \rightarrow 2^{D_q}$).

The constant B is a parameter that represents the “bandwidth” of the functions in T for mapping dataflow information between different scopes. In the worst case, B is $\max_p |D_p|$, but it is typically a small constant, and for many problems it is 1. For example, B is 1 when the D_p are sets of variable names and the functions in T map the names of one scope to corresponding names of another scope. (We will postpone further discussion of B and conditions (v)a and (v)b until after Definition 3.8 in Section 3.2.)

Finally, the distinguished value C of clause (viii) represents the special dataflow value associated with the program’s start node—the dataflow facts that hold before execution begins.

Example. The super-graph from Figure 1, annotated with the dataflow functions for the “possibly-uninitialized variables” problem, is shown in Figure 2. The “possibly-uninitialized variables” problem is to determine, for each node $n \in N^*$, the set of program variables that may be uninitialized when 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 2 adds a to the set of possibly-uninitialized variables if either a or b is in the set of possibly-uninitialized variables before node $n6$. In this problem instance $B = 1$, the meet operator is union, and the special value C associated with s_{main} is \emptyset . (The dataflow function associated with edge (s_{main}, n_1) puts all variables into the possibly-uninitialized set at n_1 .)

□

Definition 2.5. Let $IP = (G^*, D, F, B, T, M, Tr, C, \sqcap)$ be an IFDS problem instance, and let $q = [e_1, e_2, \dots, e_j]$ be a non-empty path in $\text{path}_{G^*}(m, n)$. The **path function** that corresponds to q , denoted by pf_q , is the function

$$pf_q =_{df} f_j \circ \dots \circ f_2 \circ f_1,$$

where for all i , $1 \leq i \leq j$,

$$f_i = \begin{cases} M(e_i) & \text{if } e_i \in (E^0 \cup E_1) \\ Tr(e_i) & \text{if } e_i \in E^2 \end{cases}$$

The path function for an empty path $[m: \varepsilon : m]$ is the identity function, $\lambda x. x$.

□

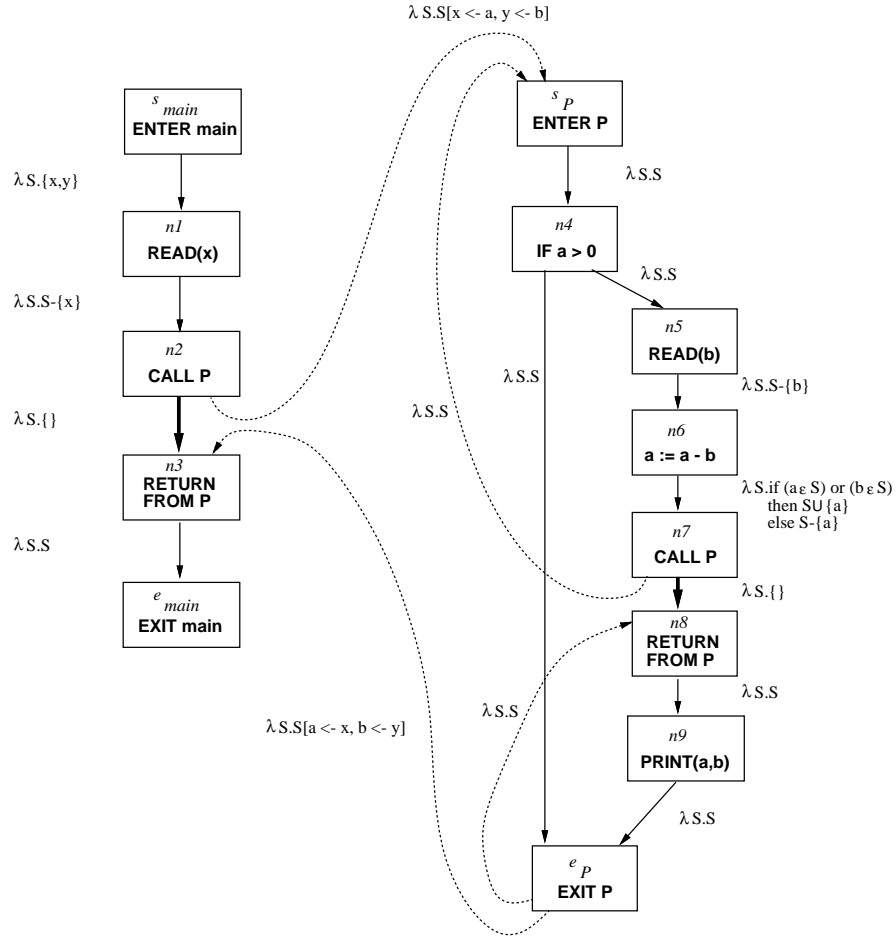


Figure 2. The super-graph from Figure 1, annotated with the dataflow functions for the “possibly-uninitialized variables” problem. The notation $S[x \leftarrow a, y \leftarrow b]$ denotes the set S with x renamed to a and y renamed to b .

Definition 2.6. Let $IP = (G^*, D, F, B, T, M, Tr, C, \sqcap)$ 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:

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

□

Except for Section 3.4, in the remainder of the paper we consider only IFDS problems in which the meet operator is union. As discussed in Section 3.4, 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 $D_{fg(n)}$) of the solution to the “may-not-be- X ” problem.

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 $2^{D_1} \rightarrow 2^{D_2}$ in a compact fashion—each function can be represented as a graph with at most $(|D_1|+1)(|D_2|+1)$ edges (or, equivalently, as an adjacency matrix with at most $(|D_1|+1)(|D_2|+1)$ bits). Throughout this section, we assume that f and g denote functions in

$2^{D_1} \rightarrow 2^{D_2}$ and $2^{D_2} \rightarrow 2^{D_3}$, respectively, where D_1 , D_2 , and D_3 are finite sets, and that f and g distribute over \cup .

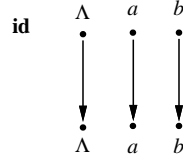
Definition 3.1. The *representation relation of f* , denoted by R_f , is a binary relation (i.e., graph) defined as follows:

$$\begin{aligned} R_f &\subseteq (D_1 \cup \{\Lambda\}) \times (D_2 \cup \{\Lambda\}) \\ R_f &=_{df} \{ (\Lambda, \Lambda) \} \\ &\quad \cup \{ (\Lambda, y) \mid y \in f(\emptyset) \} \\ &\quad \cup \{ (x, y) \mid y \in f(\{x\}) \text{ and } y \notin f(\emptyset) \}. \end{aligned}$$

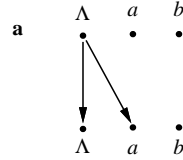
□

R_f can be thought of as a graph with $|D_1| + |D_2| + 2$ nodes, where each node represents an element of D_1 or D_2 (except for the two Λ nodes, which (roughly) stand for \emptyset).

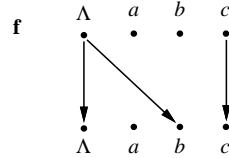
Example. The identity function $\mathbf{id}: 2^{\{a,b\}} \rightarrow 2^{\{a,b\}}$, defined by $\mathbf{id} =_{df} \lambda S. S$, is represented as follows:



The constant function $\mathbf{a}: 2^{\{a,b\}} \rightarrow 2^{\{a,b\}}$, defined by $\mathbf{a} =_{df} \lambda S. \{a\}$, is represented as follows:



The function $\mathbf{f}: 2^{\{a,b,c\}} \rightarrow 2^{\{a,b,c\}}$, defined by $\mathbf{f} =_{df} \lambda S. (S - \{a\}) \cup \{b\}$, is represented as follows:



□

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 (Λ, y) , for $y \in (D_2 \cup \{\Lambda\})$, there is never an edge (x, y) , for any $x \in D_1$. For example, in constant-function \mathbf{a} , edge (Λ, a) subsumes the need for edges (a, a) and (b, a) .

Representation relations—and, in fact, all relations in $(D_1 \cup \{\Lambda\}) \times (D_2 \cup \{\Lambda\})$ —can be thought of as representations of functions in $2^{D_1} \rightarrow 2^{D_2}$:

Definition 3.2. The *interpretation* of a relation $R \subseteq (D_1 \cup \{\Lambda\}) \times (D_2 \cup \{\Lambda\})$, denoted by $\llbracket R \rrbracket$, is the function defined as follows:

$$\begin{aligned} \llbracket R \rrbracket &: 2^{D_1} \rightarrow 2^{D_2} \\ \llbracket R \rrbracket &=_{df} \lambda X. (\{y \mid \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y \mid (\Lambda, y) \in R\}) - \{\Lambda\}. \end{aligned}$$

□

Theorem 3.3.¹ $\llbracket R_f \rrbracket = f$.

¹This is similar to Lemma 14 of Cai and Paige [4], but the notion of representation relation defined in Definition 3.1 is different from the one that Cai and Paige use.

Proof. We must show that for all $X \in 2^{D_1}$, $\llbracket R_f \rrbracket(X) = f(X)$.

$$\begin{aligned}
 f(X) &= \bigcup_{x \in X} f(\{x\}) \cup f(\emptyset) && \text{by the distributivity of } f \\
 &= \bigcup_{x \in X} \{y \mid y \in f(\{x\})\} \cup \{y \mid y \in f(\emptyset)\} \\
 &= \{y \mid \exists x \in X \text{ such that } y \in f(\{x\}) \text{ or } y \in f(\emptyset)\} \\
 \llbracket R_f \rrbracket(X) &= (\{y \mid \exists x \in X \text{ such that } (x, y) \in R_f\} \cup \{y \mid (\Lambda, y) \in R_f\}) - \{\Lambda\} \\
 &= (\{y \mid \exists x \in X \text{ such that } y \in f(\{x\}) \text{ and } y \notin f(\emptyset)\} \cup \{y \mid y \in f(\emptyset)\} \cup \{\Lambda\}) - \{\Lambda\} \\
 &= \{y \mid (\exists x \in X \text{ such that } y \in f(\{x\}) \text{ and } y \notin f(\emptyset)) \text{ or } y \in f(\emptyset)\} \\
 &= \{y \mid \exists x \in X \text{ such that } y \in f(\{x\}) \text{ or } y \in f(\emptyset)\} \\
 &= f(X)
 \end{aligned}$$

□

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. The *composition* of two relations $R_f \subseteq S_1 \times S_2$, $R_g \subseteq S_2 \times S_3$, denoted by $R_f; R_g$, is defined as follows:

$$\begin{aligned}
 R_f; R_g &\subseteq S_1 \times S_3 \\
 R_f; R_g &\stackrel{\text{df}}{=} \{(x, y) \in S_1 \times S_3 \mid \exists z \in S_2 \text{ such that } (x, z) \in R_f \text{ and } (z, y) \in R_g\}.
 \end{aligned}$$

□

Theorem 3.5. For all $f \in 2^{D_1} \rightarrow 2^{D_2}$ and $g \in 2^{D_2} \rightarrow 2^{D_3}$, $\llbracket R_f; R_g \rrbracket = g \circ f$.

Proof.

$$\begin{aligned}
 R_f; R_g &= \{(x, y) \in (D_1 \cup \{\Lambda\}) \times (D_3 \cup \{\Lambda\}) \mid \exists z \in (D_2 \cup \{\Lambda\}) \text{ such that } (x, z) \in R_f \text{ and } (z, y) \in R_g\} \\
 &= \{(\Lambda, \Lambda)\} \\
 &\quad \cup \{(\Lambda, y) \mid y \in g(\emptyset)\} \\
 &\quad \cup \{(\Lambda, y) \mid \exists z \in (D_2 \cup \{\Lambda\}) \text{ such that } z \in f(\emptyset) \text{ and } y \in g(\{z\}) \text{ and } y \notin g(\emptyset)\} \\
 &\quad \cup \{(x, y) \mid \exists z \in (D_2 \cup \{\Lambda\}) \text{ such that } z \in f(\{x\}) \text{ and } z \notin f(\emptyset) \text{ and } y \in g(\{z\}) \text{ and } y \notin g(\emptyset)\}
 \end{aligned}$$

$$\begin{aligned}
 \llbracket R_f; R_g \rrbracket(X) &= (\{\Lambda\} \\
 &\quad \cup \{y \mid y \in g(\emptyset)\} \\
 &\quad \cup \{y \mid \exists z \in (D_2 \cup \{\Lambda\}) \text{ such that } z \in f(\emptyset) \text{ and } y \in g(\{z\}) \text{ and } y \notin g(\emptyset)\} \\
 &\quad \cup \{y \mid \exists x \in X \exists z \in (D_2 \cup \{\Lambda\}) \text{ such that } z \in f(\{x\}) \text{ and } z \notin f(\emptyset) \text{ and } y \in g(\{z\}) \text{ and } y \notin g(\emptyset)\}) - \{\Lambda\} \\
 &= \{y \mid y \in g(\emptyset)\} \\
 &\quad \cup \{y \mid (\exists z \in (D_2 \cup \{\Lambda\}) \text{ such that } z \in f(\emptyset) \text{ and } y \in g(\{z\})) \text{ or } y \in g(\emptyset)\} \\
 &\quad \cup \{y \mid \exists x \in X \exists z \in (D_2 \cup \{\Lambda\}) \text{ such that } z \in f(\{x\}) \text{ and } z \notin f(\emptyset) \text{ and } y \in g(\{z\})\} \\
 &= \{y \mid y \in g(f(\emptyset))\} \\
 &\quad \cup \{y \mid \exists x \in X \exists z \in (D_2 \cup \{\Lambda\}) \text{ such that } z \in f(\{x\}) \text{ and } z \notin f(\emptyset) \text{ and } y \in g(\{z\})\} \\
 &= \{y \mid (\exists x \in X \exists z \in (D_2 \cup \{\Lambda\}) \text{ such that } z \in f(\{x\}) \text{ and } z \notin f(\emptyset) \text{ and } y \in g(\{z\})) \text{ or } y \in g(f(\emptyset))\} \\
 &= \{y \mid \exists x \in X \text{ such that } y \in g(f(\{x\})) \text{ or } y \in g(f(\emptyset))\} \\
 &= (g \circ f)(X)
 \end{aligned}$$

□

Corollary 3.6. For all $f \in 2^{D_1} \rightarrow 2^{D_2}$ and $g \in 2^{D_2} \rightarrow 2^{D_3}$, $\llbracket R_f; R_g \rrbracket = \llbracket R_g \rrbracket \circ \llbracket R_f \rrbracket$.

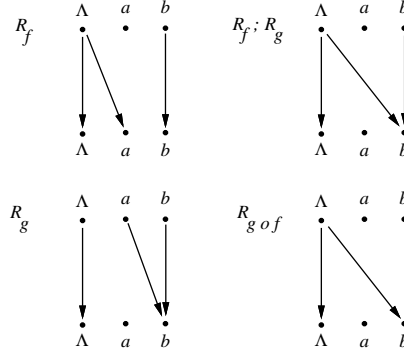
Proof. Follows immediately from Theorems 3.3 and 3.5.

□

Remark. It should be noted that Theorem 3.5 is stated in terms of $\llbracket R_f; R_g \rrbracket$, the *interpretation* of $R_f; R_g$. In particular, Theorem 3.5 does *not* claim that the following property holds:

$$\text{For all } f \in 2^{D_1} \rightarrow 2^{D_2} \text{ and } g \in 2^{D_2} \rightarrow 2^{D_3}, R_f; R_g \equiv R_{g \circ f} \quad (\dagger)$$

In fact, property (\dagger) does not hold for representation relations as we have defined them. This can be seen from the following example:



Property (\dagger) does not hold because $R_f; R_g$ contains edge (b, b) , which is not found in $R_{g \circ f}$. Edge (b, b) does not occur in $R_{g \circ f}$ because it is subsumed by edge (Λ, b) .

It is not even the case that representation relations are closed under relational composition. In particular, in the above example, because the edge (b, b) occurs in relation $R_f; R_g$, $R_f; R_g$ is not the representation relation of *any* function $h \in 2^{\{a, b\}} \rightarrow 2^{\{a, b\}}$. There are two reasons why we do not need representation relations to be closed under relational composition: (i) we have defined the notion of the interpretation of a relation for *all* relations in $(D_1 \cup \{\Lambda\}) \times (D_2 \cup \{\Lambda\})$, not just ones that are representation relations; (ii) as shown by Corollary 3.6, relational composition $(;)$ and functional composition (\circ) are related by homomorphism $\llbracket \cdot \rrbracket$.

It would have been possible to work with a different definition of the “representation relation of f ” under which (i) representation relations would be closed under relational composition, and (ii) property (\dagger) would hold. In particular, we could have defined R_f as follows:

$$\begin{aligned} R_f &\subseteq (D_1 \cup \{\Lambda\}) \times (D_2 \cup \{\Lambda\}) \\ R_f &=_{df} \{ (\Lambda, \Lambda) \} \\ &\quad \cup \{ (x, \Lambda) \mid x \in D_1 \} \\ &\quad \cup \{ (\Lambda, y) \mid y \in f(\emptyset) \} \\ &\quad \cup \{ (x, y) \mid y \in f(\{x\}) \}. \end{aligned}$$

With this definition of R_f , Theorem 3.5 (*i.e.*, $f \in 2^{D_1} \rightarrow 2^{D_2}$ and $g \in 2^{D_2} \rightarrow 2^{D_3}$, $\llbracket R_f; R_g \rrbracket = g \circ f$) holds as an immediate corollary of property (\dagger) . All of the arguments that we present to demonstrate the correctness of our techniques are based on Theorem 3.5, and therefore all of our results would continue to hold if we were to use the alternative definition of R_f .

One drawback of the alternative definition of R_f is that it would cause R_f to contain more edges than when R_f is defined as in Definition 3.1, which in turn causes there to be more edges in the realizable-path problems that we construct to solve interprocedural dataflow-analysis problems (see Definition 3.8 in Section 3.2). Using the alternative definition of representation relations would not change the asymptotic complexity of our algorithms; however, it would probably degrade the actual performance of the algorithms. Furthermore, the additional edges would add clutter to the diagrams that we present in subsequent sections to illustrate our ideas.

□

Corollary 3.7. *Given a collection of functions $f_i: 2^{D_i} \rightarrow 2^{D_{i+1}}$, for $1 \leq i \leq j$,*

$$f_j \circ f_{j-1} \circ \cdots \circ f_2 \circ f_1 = \llbracket R_{f_1}; R_{f_2}; \cdots; R_{f_j} \rrbracket.$$

Proof. The proof is by induction on j .

Base case. The base case, $j = 1$, is Theorem 3.3.

Induction step. Assume that the property holds for all $j \leq \bar{j}$; we need to show that the property holds for $\bar{j} + 1$.

$$\begin{aligned} f_{\bar{j}} \circ f_{\bar{j}-1} \circ \cdots \circ f_2 \circ f_1 &= \llbracket R_{f_1}; R_{f_2}; \cdots; R_{f_{\bar{j}}} \rrbracket \\ f_{\bar{j}+1} \circ f_{\bar{j}} \circ \cdots \circ f_2 \circ f_1 &= f_{\bar{j}+1} \circ \llbracket R_{f_1}; R_{f_2}; \cdots; R_{f_{\bar{j}}} \rrbracket \\ &= \llbracket R_{f_{\bar{j}+1}} \rrbracket \circ \llbracket R_{f_1}; R_{f_2}; \cdots; R_{f_{\bar{j}}} \rrbracket && \text{by Theorem 3.3} \\ &= \llbracket (R_{f_1}; R_{f_2}; \cdots; R_{f_{\bar{j}}}); R_{f_{\bar{j}+1}} \rrbracket && \text{by Corollary 3.6} \\ &= \llbracket R_{f_1}; R_{f_2}; \cdots; R_{f_{\bar{j}}}; R_{f_{\bar{j}+1}} \rrbracket && \text{by the associativity of relational composition} \end{aligned}$$

□

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

In this section, we show how IFDS problems correspond to a certain kind of graph-reachability problem. In particular, for each instance IP of an IFDS problem, we construct a graph $G_{IP}^\#$ and an instance of what we call a “realizable-path” reachability problem in $G_{IP}^\#$. This path problem is equivalent to IP , in a well-defined sense that

is captured by Theorem 3.10 (dataflow-fact d holds at super-graph node n iff there is a “realizable path” from a node in $G_{IP}^\#$ that represents a fact true at the start node of procedure *main* to the node in $G_{IP}^\#$ that represents fact d at node n).

Definition 3.8. Let $IP = (G^*, D, F, B, T, M, Tr, C, \cup)$ be an IFDS problem instance. We define the *exploded super-graph* for IP , denoted by $G_{IP}^\#$, as follows:

$$\begin{aligned} G_{IP}^\# &= (N^\#, E^\#, C^\#), \text{ where} \\ N^\# &= \bigcup_{p \in \{0, \dots, k\}} N_p^\#, \text{ where } N_p^\# = N_p \times (D_p \cup \{\Lambda\}), \\ E^\# &= \{ ((m, d_1), (n, d_2)) \mid (m, n) \in (E^0 \cup E^1) \text{ and } (d_1, d_2) \in R_{M(m, n)} \} \\ &\quad \cup \{ ((m, d_1), (n, d_2)) \mid (m, n) \in E^2 \text{ and } (d_1, d_2) \in R_{Tr(m, n)} \}, \\ C^\# &= \{ (s_{main}, c) \mid c \in (C \cup \{\Lambda\}) \}. \end{aligned}$$

□

The nodes of $G_{IP}^\#$ are pairs of the form (n, d) ; each node n of N_p^* is “exploded” into $|D_p| + 1$ 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 . Set $C^\#$, consisting of all pairs (s_{main}, c) where $c \in (C \cup \{\Lambda\})$, corresponds to the distinguished value C of problem instance IP ; these nodes will be the distinguished sources in the (multi-source) reachability problem that corresponds to dataflow-problem IP .

Example. The exploded super-graph that corresponds to the instance of the “possibly-uninitialized variables” problem shown in Figure 2 is shown in Figure 3.

□

We can now clarify the reasons for conditions (v)a and (v)b in Definition 2.4:

(v)a. For all $t \in T_{p,q}$ and $x \in D_p$, $|t(\{x\}) - t(\emptyset)| \leq B$;

(v)b. For all $t \in T_{p,q}$ and $y \in D_q$, if $y \notin t(\emptyset)$ then $|\{x \mid y \in t(\{x\})\}| \leq B$.

These conditions ensure that nodes in the representation relations of the transfer functions associated with E^2 edges have limited indegree and outdegree. In particular, each node of the form (m, d) where $m \in (Call \cup Exit)$ and $d \neq \Lambda$, has at most B outgoing edges; each node of the form (n, d) where $n \in (Enter \cup Ret)$ and $d \neq \Lambda$, has at most B incoming edges. (There is no restriction on the number of outgoing edges from nodes of the form (m, Λ) where $m \in (Call \cup Exit)$; the definition of representation relations ensures that nodes of the form (n, Λ) where $n \in (Enter \cup Ret)$ have at most one incoming edge.)

These restrictions have an impact on the complexity arguments that we make in Sections 4.1, 5.1, and 6.2. They allow us to give more precise bounds using the parameter B rather than the worst-case parameter $\max_p |D_p|$.

Throughout the remainder of the paper, we use the terms “realizable path” and “valid path” to refer to two related concepts in the super-graph and the exploded super-graph. In both cases, 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 “valid paths” will always be used in connection with paths in the super-graph, whereas the term “realizable paths” will always be used in connection with paths in the exploded super-graph.

Definition 3.9. Let $G_{IP}^\# = (N^\#, E^\#, C^\#)$ be the exploded super-graph for IFDS problem instance $IP = (G^*, D, F, B, T, M, Tr, C, \cup)$. Let each call node in G^* be given a unique index in the range $[1 \dots |Call|]$. For each such indexed call node c_i , label the edges of $G_{IP}^\#$ that correspond to c_i ’s outgoing E^2 edge $(c_i, s_{calledProc(c_i)})$ (i.e., edges of the form $((c_i, d_1), (s_{calledProc(c_i)}, d_2))$) by the symbol “ i ”. Label the edges of $G_{IP}^\#$ that correspond to return edge $(e_{calledProc(c_i)}, succ^1(c_i))$ (i.e., edges of the form $((e_{calledProc(c_i)}, d_3), (succ^1(c_i), d_4))$) by the symbol “ i ”.

A *same-level realizable path* in $G_{IP}^\#$ is a path such that the sequence of symbols on the labeled edges is a string in the language of balanced parentheses generated from nonterminal *matched* of Definition 2.3.

A *realizable path* in $G_{IP}^\#$ is a path such that the sequence of symbols on the labeled edges is a string in the language generated from nonterminal *valid* of Definition 2.3.

□

As with same-level valid paths, the same-level realizable paths from nodes of the form (m, d_1) to nodes of the form (n, d_2) , where m and n are in the same procedure, will be used to capture the transmission of effects from m to n via sequences of execution steps during which the call stack may temporarily grow deeper—because of

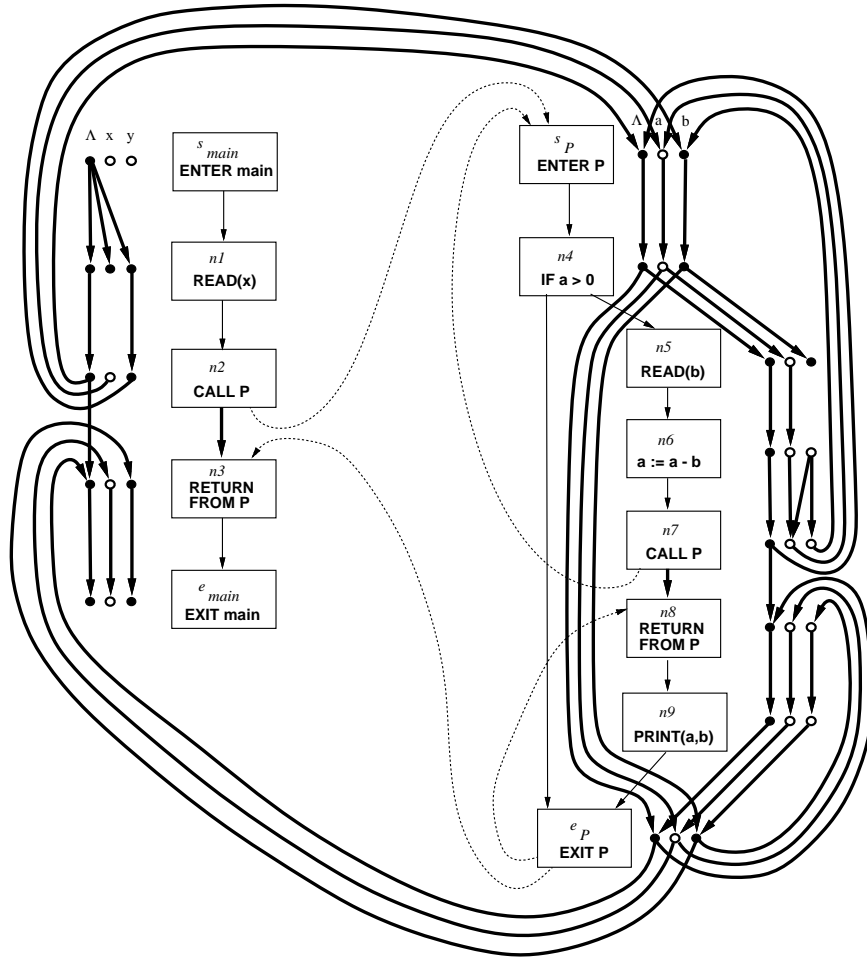


Figure 3. The exploded super-graph that corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 2. Closed circles represent nodes of $G_{IP}^\#$ that are reachable along realizable paths from (s_{main}, Λ) (see Definition 3.9). Open circles represent nodes not reachable along such paths.

calls—but never shallower than its original depth before eventually returning to its original depth.

As with valid paths, we are primarily interested in realizable paths from the start node (*i.e.*, realizable paths whose source node is of the form (s_{main}, d_1)). The realizable paths from nodes of the form (s_{main}, d_1) to nodes of the form (n, d_2) will be used to capture the transmission of effects from s_{main} to n via some sequence of execution steps.

Example. In the exploded super-graph shown in Figure 3, the path $[((s_{main}, \Lambda), (n1, y)), ((n1, y), (n2, y)), ((n2, y), (s_p, b)), ((s_p, b), (n4, b)), ((n4, b), (e_p, b)), ((e_p, b), (n3, y))]$ is a same-level realizable path; the path $[((s_{main}, \Lambda), (n1, y)), ((n1, y), (n2, y)), ((n2, y), (s_p, b)), ((s_p, b), (n4, b))]$ is a realizable path (but not a same-level realizable path); the path $[((s_{main}, \Lambda), (n1, y)), ((n1, y), (n2, y)), ((n2, y), (s_p, b)), ((s_p, b), (n4, b)), ((n4, b), (e_p, b)), ((e_p, b), (n8, b))]$ is not a realizable path.

□

We now turn to the main theorem of this section, Theorem 3.10, which shows that an IFDS problem instance IP is equivalent to a multi-source realizable-path reachability problem in graph $G_{IP}^\#$. The practical consequence of this theorem is that we can find the meet-over-all-valid-paths solution to IP by solving a multi-source realizable-path reachability problem in graph $G_{IP}^\#$.

Example. In the exploded super-graph shown in Figure 3, which corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 2, closed circles represent nodes that are reachable along realiz-

able paths from (s_{main}, Λ) . Open circles represent nodes not reachable along realizable paths. Note that nodes $(n8, b)$ and $(n9, b)$ are reachable only along non-realizable paths from (s_{main}, Λ) .

As shown in Theorem 3.10, 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, $MVP_{e_p} = \{b\}$. (That is, variable b is the only possibly-uninitialized variable when execution reaches the exit node of procedure p .) In Figure 3, this information can be obtained by determining that there is a realizable path from (s_{main}, Λ) to (e_p, b) , but not from (s_{main}, Λ) to (e_p, a) . \square

Theorem 3.10. *Let $G_{IP}^\# = (N^\#, E^\#, C^\#)$ be the exploded super-graph for IFDS problem instance $IP = (G^*, D, F, B, T, M, Tr, C, \cup)$, 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 one of the nodes in $C^\#$ to node (n, d) .*

Proof. In what follows, we only need to discuss non-empty paths; empty paths arise only when $n = s_{main}$, and in that case it is easy to see that the theorem holds.

“Only if” case. Suppose $d \in MVP_n$; we need to demonstrate the existence of a realizable path in graph $G_{IP}^\#$ from one of the nodes in $C^\#$ to node (n, d) .

Because $MVP_n = \bigcup_{r \in IVP(s_{main}, n)} pf_r(C)$, there must be a valid path $q = [s_{main}: (n_1, n_2), (n_2, n_3), \dots, (n_j, n_{j+1}): n]$ in G^* such that $\{d\} \subseteq pf_q(C)$. For all i , $1 \leq i \leq j$, let f_i be defined as follows:

$$f_i = \begin{cases} M(n_i, n_{i+1}) & \text{if } (n_i, n_{i+1}) \in (E^0 \cup E_1) \\ Tr(n_i, n_{i+1}) & \text{if } (n_i, n_{i+1}) \in E^2 \end{cases}$$

We have:

$$\begin{aligned} \{d\} &\subseteq pf_q(C) \\ &= (f_j \circ \dots \circ f_2 \circ f_1)(C) && \text{by Definition 2.5} \\ &= \llbracket R_{f_j}; \dots; R_{f_2}; R_{f_1} \rrbracket(C) && \text{by Corollary 3.7} \\ &= (\{y \mid \exists c \in C \text{ such that } (c, y) \in (R_{f_j}; \dots; R_{f_2}; R_{f_1})\} \cup \{y \mid (\Lambda, y) \in (R_{f_j}; \dots; R_{f_2}; R_{f_1})\}) - \{\Lambda\} \\ &&& \text{by Definition 3.2} \end{aligned}$$

Recall that edge-function f_1 is associated with edge (s_{main}, n_2) , and that edge-function f_j is associated with edge (n_j, n_{j+1}) . Consequently, the last line in the above group of equations implies that there is a path $q^\#$ in graph $G_{IP}^\#$ of the following form:

$$q^\# = [(s_{main}, c): ((n_1, d_1), (n_2, d_2)), ((n_2, d_2), (n_3, d_3)), \dots, ((n_j, d_j), (n_{j+1}, d_{j+1})): (n, d)],$$

where $c \in (C \cup \{\Lambda\})$ and for all i , $1 \leq i \leq j$, $(d_i, d_{i+1}) \in R_{f_i}$. That is, $q^\#$ is a path in $G_{IP}^\#$ from node (s_{main}, c) , one of the nodes of $C^\#$, to node (n, d) . Because path q is a valid path in G^* , path $q^\#$ is a realizable path in $G_{IP}^\#$.

Therefore, $q^\#$ satisfies the conditions for the path whose existence we were required to demonstrate.

“If” case. Suppose that there is a realizable path $q^\# = [c^\#: e_1^\#, e_2^\#, \dots, e_j^\#: (n, d)]$ in graph $G_{IP}^\#$ from node $c^\# \in C^\#$ to node (n, d) . We need to demonstrate that $d \in MVP_n$.

For all i , $1 \leq i \leq j$, let $e_i^\# = ((n_i, d_i), (n_{i+1}, d_{i+1}))$, and let f_i be defined as follows:

$$f_i = \begin{cases} M(n_i, n_{i+1}) & \text{if } (n_i, n_{i+1}) \in (E^0 \cup E_1) \\ Tr(n_i, n_{i+1}) & \text{if } (n_i, n_{i+1}) \in E^2 \end{cases}$$

Let q be the following path in G^* :

$$q = [s_{main}: (n_1, n_2), (n_2, n_3), \dots, (n_j, n_{j+1}): n].$$

By supposition, $q^\#$ is a realizable path in $G_{IP}^\#$ from node $c^\# \in C^\#$ to node (n, d) ; therefore, q must be a valid path in G^* .

By the construction of $G_{IP}^\#$ (Definition 3.8), and the supposition that $q^\#$ is a realizable path in $G_{IP}^\#$ from node $c^\# \in C^\#$ to node (n, d) , we have:

$$d \in ((\{y \mid \exists x \in C \text{ such that } (x, y) \in (R_{f_j}; \dots; R_{f_2}; R_{f_1})\} \cup \{y \mid (\Lambda, y) \in (R_{f_j}; \dots; R_{f_2}; R_{f_1})\}) - \{\Lambda\}).$$

Consequently,

$$\begin{aligned}
\{d\} &\subseteq ((\{y \mid \exists x \in C \text{ such that } (x, y) \in (R_{f_j}; \dots; R_{f_2}; R_{f_1})\} \cup \{y \mid (\Lambda, y) \in (R_{f_j}; \dots; R_{f_2}; R_{f_1})\}) - \{\Lambda\}) \\
&= \llbracket R_{f_j}; \dots; R_{f_2}; R_{f_1} \rrbracket(C) && \text{by Definition 3.2} \\
&= (f_j \circ \dots \circ f_2 \circ f_1)(C) && \text{by Corollary 3.7} \\
&= pf_q(C) && \text{by Definition 2.5} \\
&\subseteq \bigcup_{t \in \text{IVP}(s_{\text{main}}, n)} pf_t(C) \\
&= MVP_n
\end{aligned}$$

Therefore $d \in MVP_n$, as was to be shown.

□

3.3. Restricted Subclasses of IFDS Problems

In this section, we define two special classes of IFDS problems: *sparse* problems and *locally separable* problems. As discussed in Sections 4.1, 5.1, and 6.2, our dataflow-analysis algorithms are more efficient when applied to problems in these classes.

Definition 3.11. An IFDS problem instance $IP = (G^*, D, F, B, T, M, Tr, C, \cup)$ is ***h-sparse*** if there is a constant h such that for all $F_p \in F$ and $f \in F_p$, $\sum_{d \in D_p} |f(\{d\}) - f(\emptyset)| \leq h|D_p|$.

□

That is, in h -sparse problems, the total number of edges emanating from the non- Λ nodes in the representation relation of an E^0 or E^1 edge’s function is at most $h|D_p|$.

In general, when the nodes of the control-flow graph represent individual statements and predicates (rather than basic blocks) we expect most problems that are distributive to be h -sparse (with $h \ll \max_p |D_p|$): 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 $|D_p|$ edges.

Definition 3.12. An IFDS problem instance $IP = (G^*, D, F, B, T, M, Tr, C, \cup)$ is ***locally separable*** if

- (i) For all $F_p \in F$, all $f \in F_p$, and all $x \in D_p$, $f(\{x\}) \subseteq \{x\} \cup f(\emptyset)$;
- (ii) $B = 1$;
- (iii) If (c, s_q) and (e_q, r) are corresponding call and return edges then, for all $x \in D_{fg(c)}$ and $y \in D_q$,
 - a. If $Tr(c, s_q)(\{x\}) - Tr(c, s_q)(\emptyset) = \{y\}$ then $Tr(e_q, r)(\{y\}) - Tr(e_q, r)(\emptyset) \subseteq \{x\}$.
 - b. If $Tr(e_q, r)(\{y\}) - Tr(e_q, r)(\emptyset) = \{x\}$ then $Tr(c, s_q)(\{x\}) - Tr(c, s_q)(\emptyset) \subseteq \{y\}$.

□

The locally separable problems are the interprocedural versions of the classical separable problems from intraprocedural dataflow analysis (also known as gen/kill or bit-vector problems). All locally separable problems are 1-sparse, but not vice versa.

Clause (i) restricts the dataflow functions in F , which map dataflow information across E^0 and E^1 edges, to have essentially component-wise dependences. That is, in the representation relation $R_{M(m, n)}$ for an edge $(m, n) \in (E^0 \cup E^1)$, each node (n, d) can only be the target of a single edge—whose source is either of the form (m, d) or of the form (m, Λ) .

Clause (ii) restricts the binding functions in T , which map dataflow information across call and return edges, to be simple one-to-one renamings of domain elements. (However, different call sites on the *same* procedure may make use of *different* T functions to map dataflow information across to the called procedure.)

Clause (iii) restricts the mapping Tr so that corresponding call and return edges have related binding functions. In particular, if there is an edge $((c, d_1), (s_q, d_2))$, where $d_1 \neq \Lambda$, in the representation relation of the function $Tr(c, s_q)$, then in the representation relation of the function $Tr(e_q, r)$ there is either an edge $((e_q, d_2), (r, d_1))$ or (e_q, d_2) has no outgoing edge. The analogous restriction holds for $Tr(e_q, r)$ with respect to $Tr(c, s_q)$.

Example. In general, an instance of the possibly-uninitialized variables problem will not be locally separable. For example, in Figure 3, a ’s initialization status after the statement $a := a - b$ depends on that of b before the statement. Thus, the representation relation for the dataflow function on edge $(n6, n7)$ has an edge from node $(n6, b)$ to $(n7, a)$. This violates clause (i) of Definition 3.12.

However, when the nodes of the control-flow graph represent individual statements and predicates, 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- Λ 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.

□

3.4. Intersection Problems

An \cap -distributive IFDS problem instance (with distinguished value C) can be solved by solving a \cup -distributive IFDS problem instance (with distinguished value \bar{C}) and then complementing the answer. The problem transformation is carried out merely by replacing each \cap -distributive function f by the \cup -distributive function $f' =_{df} \lambda S. \overline{f(\bar{S})}$. This transformation is justified by the following derivation:

$$\begin{aligned} \overline{MVP_n} &= \overline{\bigcap_{q \in IVP(s_{main}, n)} pf_q(C)} \\ &= \bigcup_{q \in IVP(s_{main}, n)} \overline{pf_q(C)} \\ &= \bigcup_{q \in IVP(s_{main}, n)} pf'_q(\bar{C}) \quad \text{where if } pf_q = f_j \circ \dots \circ f_1 \text{ then } pf'_q =_{df} f'_j \circ \dots \circ f'_1 \end{aligned}$$

Thus, the solution of the original problem is the complement of the solution of the transformed problem.

The functions that arise in gen/kill intersection problems are all of the form $f = \lambda x. (x - kill) \cup gen$. It is easy to show that $f' = \lambda x. (x - gen) \cup (kill - gen)$, which is also in gen/kill form. Thus, when the transformation described above is used to transform an \cap -distributive gen/kill IFDS problem into a \cup -distributive IFDS problem, every edge function satisfies clause (i) of Definition 3.12. It is reasonable to expect that every \cap -distributive gen/kill IFDS problem will also satisfy clauses (ii) and (iii) of Definition 3.12. Consequently, when an \cap -distributive gen/kill IFDS problem is transformed into a \cup -distributive IFDS problem, the result is a locally separable problem.

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

In this section, we present the first of our three algorithms for the realizable-path reachability problem. The algorithm described in this section is a dynamic-programming algorithm that tabulates certain kinds of same-level realizable paths. As we show in Section 4.1, its running time is polynomial in various parameters of the problem, and the algorithm is asymptotically faster than the best previously known algorithm for the problem.

The algorithm, which we call the **Tabulation Algorithm**, is presented in Figure 4. 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 some node $c^\# \in C^\#$ 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 some $c^\# \in C^\#$ to (n, d_2) .

The Tabulation Algorithm uses another set, named SummaryEdge, to record the existence of **summary edges**, which represent same-level realizable paths that run from nodes of the form (n, d_1) , where $n \in Call$, to $(succ^1(n), d_2)$.

The Tabulation Algorithm is a worklist algorithm that starts with a certain set of path edges, and on each iteration of the main loop in procedure ForwardTabulateSLRPs (lines [14]-[43]) deduces the existence of additional path edges (and summary edges). The initial set of path edges corresponds to the 0-length same-level realizable paths of the form $((s_{main}, c), (s_{main}, c))$, for all $(s_{main}, c) \in C^\#$ (see lines [5]-[8]). The configurations that are used by the Tabulation Algorithm to deduce the existence of additional path edges are depicted in Figure 5; the five diagrams of Figure 5 correspond to lines [18]-[20], [21]-[23], [29], [30]-[32], and [38]-[40] of Figure 4. In Figure 5, the bold dotted arrows represent edges that are inserted into set PathEdge if they were not previously in that set.

It is important to note the role of lines [30]-[32] of Figure 4, which are executed only when a new summary edge is discovered:

```
[30] for each  $d_3$  such that  $((s_{fg(c)}, d_3), (c, d_4)) \in \text{PathEdge}$  do
[31]   Propagate(PathEdge,  $((s_{fg(c)}, d_3), (succ^1(c), d_5))$ , WorkList)
[32] od
```

Unlike edges in $E^\#$, edges are inserted into SummaryEdge on-the-fly. The purpose of line [31] is to restart the processing that finds same-level realizable paths from $(s_{fg(c)}, d_3)$ as if summary edge $((c, d_4), (succ^1(c), d_5))$ had been in place all along.

The final step of the Tabulation Algorithm (lines [10]-[12]) 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 For-

```

procedure SolveViaTabulation( $G_{IP}^\#$ )
begin
[1]  Let  $(N^\#, E^\#, C^\#) = G_{IP}^\#$ 
[2]  PathEdge :=  $\emptyset$ 
[3]  SummaryEdge :=  $\emptyset$ 
[4]  WorkList :=  $\emptyset$ 
[5]  for each  $(s_{main}, c) \in C^\#$  do
[6]    Insert  $((s_{main}, c), (s_{main}, c))$  into PathEdge
[7]    Insert  $((s_{main}, c), (s_{main}, c))$  into WorkList
[8]  od
[9]  ForwardTabulateSLRPs(WorkList)
[10] for each  $n \in N^*$  do
[11]   $X_n := \{ d_2 \in D_{fg(n)} \mid \exists d_1 \in (D_{fg(n)} \cup \{ \Lambda \}) \text{ such that } ((s_{fg(n)}, d_1), (n, d_2)) \in \text{PathEdge} \}$ 
[12] od
end

procedure Propagate(EdgeSet,  $e$ , WorkList)
begin
[13] if  $e \notin \text{EdgeSet}$  then Insert  $e$  into EdgeSet; Insert  $e$  into WorkList fi
end

procedure ForwardTabulateSLRPs(WorkList)
begin
[14] while WorkList  $\neq \emptyset$  do
[15]  Select and remove an edge  $((s_p, d_1), (n, d_2))$  from WorkList
[16]  switch  $n$ 
[17]    case  $n \in \text{Call}_p$  :
[18]      for each  $d_3$  such that  $((n, d_2), (s_{calledProc(n)}, d_3)) \in E^\#$  do
[19]        Propagate(PathEdge,  $((s_{calledProc(n)}, d_3), (s_{calledProc(n)}, d_3))$ , WorkList)
[20]      od
[21]      for each  $d_3$  such that  $((n, d_2), (succ^1(n), d_3)) \in (E^\# \cup \text{SummaryEdge})$  do
[22]        Propagate(PathEdge,  $((s_p, d_1), (succ^1(n), d_3))$ , WorkList)
[23]      od
[24]    end case
[25]    case  $n = e_p$  :
[26]      for each  $c \in \text{callers}(p)$  do
[27]        for each  $d_4, d_5$  such that  $((c, d_4), (s_p, d_1)) \in E^\#$  and  $((e_p, d_2), (succ^1(c), d_5)) \in E^\#$  do
[28]          if  $((c, d_4), (succ^1(c), d_5)) \notin \text{SummaryEdge}$  then
[29]            Insert  $((c, d_4), (succ^1(c), d_5))$  into SummaryEdge
[30]          for each  $d_3$  such that  $((s_{fg(c)}, d_3), (c, d_4)) \in \text{PathEdge}$  do
[31]            Propagate(PathEdge,  $((s_{fg(c)}, d_3), (succ^1(c), d_5))$ , WorkList)
[32]          od
[33]        fi
[34]      od
[35]    od
[36]  end case
[37]  case  $n \in (N_p - \text{Call}_p - \{ e_p \})$  :
[38]    for each  $(m, d_3)$  such that  $((n, d_2), (m, d_3)) \in E^\#$  do
[39]      Propagate(PathEdge,  $((s_p, d_1), (m, d_3))$ , WorkList)
[40]    od
[41]  end case
[42] end switch
[43] od
end

```

Figure 4. The Tabulation Algorithm determines the meet-over-all-valid-paths solution to IP by determining whether certain same-level realizable paths exist in $G_{IP}^\#$.

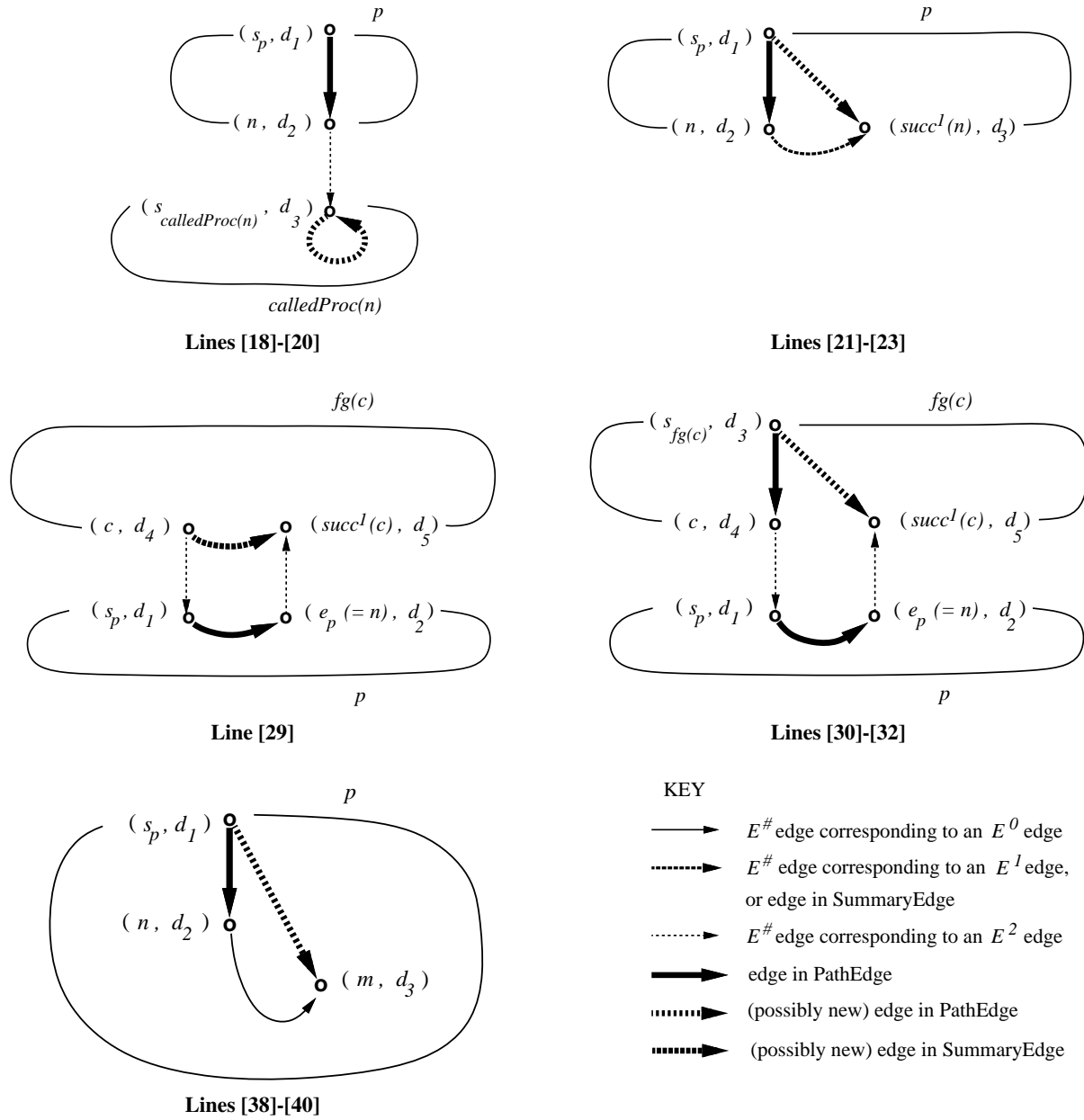


Figure 5. The above five diagrams show the situations handled in lines [18]-[20], [21]-[23], [29], [30]-[32], and [38]-[40] of Figure 4.

wardTabulateSLRPs:

$$[11] \quad X_n := \{ d_2 \in D_{fg(n)} \mid \exists d_1 \in (D_{fg(n)} \cup \{ \Lambda \}) \text{ such that } ((s_{fg(n)}, d_1), (n, d_2)) \in \text{PathEdge} \}$$

At first glance, this may not appear to be correct. According to line [11], d_2 is put into X_n if there is a *same-level* realizable path (*i.e.*, a path edge) from $(s_{fg(n)}, d_1)$ to (n, d_2) , rather than a *realizable* path from some $c^\# \in C^\#$ to (n, d_2) . However, as we show in the proof of Theorem 4.1 below, the Tabulation Algorithm correctly identifies all nodes in $G_{IP}^\#$ that are reachable along realizable paths from nodes in $C^\#$ because same-level realizable paths from $(s_{fg(n)}, d_1)$ are only considered once it is known that $(s_{fg(n)}, d_1)$ is reachable from some $c^\# \in C^\#$. Consequently, by Theorem 3.10, 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 $n \in N^*$, $X_n = MVP_n$.

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

Proof. PathEdge is initially empty, and there are only a finite number of possible path edges. A path edge is inserted into WorkList exactly once, when it is inserted into PathEdge (see lines [6]-[7] and line [13] of Figure 4). Consequently, the outermost loop of procedure ForwardTabulateSLRPs (lines [14]-[43]) can execute at most a finite number of times, and the Tabulation Algorithm is guaranteed to halt.

To show that $X_n = MVP_n$, for all $n \in N^*$, we appeal to Theorem 3.10. Because the value of X_n is defined in line [11] of Figure 4 by the assignment:

$$X_n := \{ d_2 \in D_{fg(n)} \mid \exists d_1 \in (D_{fg(n)} \cup \{ \Lambda \}) \text{ such that } ((s_{fg(n)}, d_1), (n, d_2)) \in \text{PathEdge} \},$$

we know from Theorem 3.10 that $X_n = MVP_n$ will hold if we can show the following:

For all $d_2, \exists d_1 \in (D_{fg(n)} \cup \{ \Lambda \})$ such that $((s_{fg(n)}, d_1), (n, d_2)) \in \text{PathEdge}$ when the Tabulation Algorithm terminates iff there is a realizable path in $G_{IP}^\#$ from one of the nodes in $C^\#$ to node (n, d_2) .

“Only if” case. Suppose $\exists d_1 \in (D_{fg(n)} \cup \{ \Lambda \})$ such that $((s_{fg(n)}, d_1), (n, d_2)) \in \text{PathEdge}$ when the Tabulation Algorithm terminates; we need to show that there is a realizable path in $G_{IP}^\#$ from one of the nodes in $C^\#$ to node (n, d_2) .

This case is proven by showing that the following invariant holds for the outermost loop of procedure ForwardTabulateSLRPs:

$$\begin{aligned} \forall n, d_1, d_2, ((s_{fg(n)}, d_1), (n, d_2)) \in \text{PathEdge} \Rightarrow \\ \text{(a) } \exists c^\# \in C^\# \text{ such that there is a realizable path from } c^\# \text{ to } (n, d_2), \text{ and} \\ \text{(b) there is a same-level realizable path from } (s_{fg(n)}, d_1) \text{ to } (n, d_2). \end{aligned}$$

PathEdge is first initialized to \emptyset in line [2]. Then, in lines [5]-[8], all edge of the form $((s_{main}, c), (s_{main}, c))$, where $(s_{main}, c) \in C^\#$, are inserted into PathEdge. All such edges meet the conditions of the invariant, and so the invariant holds when ForwardTabulateSLRPs is first called.

Now assume that the invariant holds true after j iterations. Because a path edge is inserted into WorkList only when it is inserted into PathEdge, when edge $((s_p, d_1), (n, d_2))$ is removed from WorkList in line [15] at the beginning of iteration $j + 1$, we know that $((s_p, d_1), (n, d_2)) \in \text{PathEdge}$. Because it was on some previous iteration that $((s_p, d_1), (n, d_2))$ was inserted into PathEdge, from the invariant we know that (a) there is some $c^\# \in C^\#$ such that there is a realizable path $q^\#$ from $c^\#$ to (n, d_2) , and (b) there is a same-level realizable path $s^\#$ from (s_p, d_1) to (n, d_2) .

Propagate is called from four places in procedure ForwardTabulateSLRPs (lines [19], [22], [31], and [39]). In each case, we can demonstrate that if the call on Propagate actually does insert a new edge into PathEdge, then there is a realizable path $q^{\#\#}$ and a same-level realizable path $s^{\#\#}$ as required to re-establish the invariant. For example, in the call in line [39], $q^{\#\#}$ is $q^\#$ extended with edge $((n, d_2), (m, d_3))$, and $s^{\#\#}$ is $s^\#$ extended with edge $((n, d_2), (m, d_3))$.

For the call on Propagate in line [31], the argument is more subtle. In addition to what was observed above about $q^\#$ and $s^\#$, we know that edge $((s_{fg(c)}, d_3), (c, d_4))$ is in PathEdge, and hence from the invariant we know that there is some $\bar{c}^\# \in C^\#$ such that there is a realizable path $t^\#$ from $\bar{c}^\#$ to (c, d_4) . Thus, the realizable path $q^{\#\#}$ required by the invariant is $t^\#$ extended with edge $((c, d_4), (s_p, d_1))$, same-level realizable path $s^\#$, and edge $((e_p, d_2), (succ^1(c), d_5))$; that is,

$$q^{\#\#} = t^\# \parallel [((c, d_4), (s_p, d_1))] \parallel s^\# \parallel [((e_p, d_2), (succ^1(c), d_5))],$$

where “ \parallel ” denotes path concatenation. Because edges $((c, d_4), (s_p, d_1))$ and $((e_p, d_2), (succ^1(c), d_5))$ are corresponding call and return edges, $q^{\#\#}$ is a realizable path in $G_{IP}^\#$. Similarly, by the invariant we know that there is a same-level realizable path $u^\#$ from $(s_{fg(c)}, d_3)$ to (c, d_4) . The same-level realizable path $s^{\#\#}$ required by the invariant is

$$s^{\#\#} = u^\# \parallel [((c, d_4), (s_p, d_1))] \parallel s^\# \parallel [((e_p, d_2), (succ^1(c), d_5))].$$

The arguments for the other two places where Propagate is invoked are similar and are left to the reader.

Consequently, the invariant holds after iteration $j + 1$.

“If” case. Suppose there is a realizable path in $G_{IP}^\#$ from one of the nodes in $C^\#$ to node (n, d_2) ; we need to show $\exists d_1 \in (D_{fg(n)} \cup \{ \Lambda \})$ such that $((s_{fg(n)}, d_1), (n, d_2)) \in \text{PathEdge}$ when the Tabulation Algorithm terminates.

This case is proven by induction on the lengths of realizable paths in $G_{IP}^\#$. The induction hypothesis is:

$\forall n, d_2$, if there is a realizable path of length j from one of the nodes in $C^\#$ to node (n, d_2) then $\exists d_1$ such that $((s_{fg(n)}, d_1), (n, d_2)) \in \text{PathEdge}$ when the Tabulation Algorithm terminates.

Base case. For the case $j = 0$ (realizable paths of length 0), the induction hypothesis holds because of the way PathEdge is initialized in the loop on lines [5]-[8] of Figure 4.

Induction step. Assume that the induction hypothesis holds true for all realizable paths of length less than or equal to \bar{j} . Let $q^\#$ be a realizable path of length $\bar{j} + 1$ from one of the nodes in $C^\#$ to node (n, d) , and suppose $q^\#$ has the form

$$q^\# = [(s_{\text{main}}, c): ((n_1, d_1), (n_2, d_2)), \dots, ((n_{\bar{j}}, d_{\bar{j}}), (n_{\bar{j}+1}, d_{\bar{j}+1})), ((n_{\bar{j}+1}, d_{\bar{j}+1}), (n, d))],$$

where $c \in (C \cup \{\Lambda\})$.

Let $q^\#$ denote the prefix of $q^\#$ of length \bar{j} . Then by the induction hypothesis, we know that there exists a d' such that $((s_{fg(n_{\bar{j}+1}}), d'), (n_{\bar{j}+1}, d_{\bar{j}+1})) \in \text{PathEdge}$. Because whenever an edge is inserted into PathEdge it is also inserted into WorkList, we know that at some point during the execution of procedure ForwardTabulateSLRPs, the edge $((s_{fg(n_{\bar{j}+1}}), d'), (n_{\bar{j}+1}, d_{\bar{j}+1}))$ was the edge removed from WorkList in line [15].

There are three cases to consider, depending on the type of node $n_{\bar{j}+1}$. These correspond to the three cases of the switch statement of procedure ForwardTabulateSLRPs. For example, if node $n_{\bar{j}+1}$ is an exit node, then edge $((n_{\bar{j}+1}, d_{\bar{j}+1}), (n, d))$ must be a return edge. Because $q^\#$ is a realizable path, there must be a corresponding call edge $((c, d''), (s_{fg(n_{\bar{j}+1}}), d'))$ that occurs earlier in $q^\#$. (Note that $fg(c) = fg(n)$.) Because the prefix of $q^\#$ up to the occurrence of this edge is a realizable path of length less than \bar{j} , by the induction hypothesis we know that, when the Tabulation Algorithm terminates, $((s_{fg(c)}, d'''), (c, d''))$ must be in PathEdge. There are now two possibilities to consider:

- (i) Suppose that at the time edge $((s_{fg(n_{\bar{j}+1}}), d'), (n_{\bar{j}+1}, d_{\bar{j}+1}))$ is processed, $((s_{fg(c)}, d'''), (c, d'')) \in \text{PathEdge}$. Then there will be a call at line [31] of the form

$$\text{Propagate}(\text{PathEdge}, ((s_{fg(n)}, d'''), (n, d)), \text{WorkList}),$$

- (ii) Suppose that at the time edge $((s_{fg(n_{\bar{j}+1}}), d'), (n_{\bar{j}+1}, d_{\bar{j}+1}))$ is processed, $((s_{fg(c)}, d'''), (c, d'')) \notin \text{PathEdge}$. Then $((c, d''), (n, d))$ will be inserted into SummaryEdge if it is not already there, and later edge $((s_{fg(c)}, d'''), (c, d''))$ will be processed. (This must occur at some point during the Tabulation Algorithm, because edge $((s_{fg(c)}, d'''), (c, d''))$ is inserted into WorkList when it is inserted into PathEdge. We know that this happens eventually.) At this point, a call is generated at line [22] of the form

$$\text{Propagate}(\text{PathEdge}, ((s_{fg(n)}, d'''), (n, d)), \text{WorkList}).$$

Thus, in either case, $((s_{fg(n)}, d'''), (n, d))$ will at some point be inserted into PathEdge, which establishes the induction hypothesis for path $q^\#$. (The arguments for the other two cases are simpler, and are again left to the reader.)

Consequently, the induction hypothesis holds true for every realizable path in $G_{IP}^\#$.

□

4.1. Cost of the Tabulation Algorithm

In this section, we show that the running time of the Tabulation Algorithm is polynomial in various parameters of a problem instance. We also show that the algorithm is asymptotically faster than the best previously known algorithm for the realizable-path reachability problem [18].

In this section, as well as in later sections in which we derive bounds on the running times of algorithms, we use the name of a set to denote the set's size. For example, we use Call , rather than $|\text{Call}|$, to denote the number of call nodes in graph G^* . However, we make three small deviations from this convention:

- (i) We use N , rather than N^* , to stand for $|N^*|$, the number of nodes in graph G^* .
- (ii) We use E , rather than E^* , to stand for $|E^*|$, the number of edges in graph G^* .
- (iii) Rather than distinguish between the different domains D_p , for $p \in \{0, \dots, k\}$, we will express costs in terms of a single quantity D , where

$$D =_{df} \max_{p \in \{0, \dots, k\}} |D_p|.$$

The primary justification for this simplification is that in dataflow analysis problems all of the D_p sets will share a subset of values in common for representing information about global variables. Usually this common subset dominates the size of each D_p set, and thus all of the D_p sets will be of roughly the same size, namely D .

The time bounds discussed below are derived under the following assumptions about the running times of operations on $G_{IP}^\#$'s nodes and edges, and on auxiliary edge sets SummaryEdge, PathEdge, and WorkList:

- (i) An edge can be inserted into SummaryEdge, PathEdge, or WorkList in unit time.
- (ii) An edge can be tested for membership in SummaryEdge or in PathEdge in unit time.
- (iii) A random edge can be selected from WorkList in unit time.
- (iv) It is possible to iterate over the set of all PathEdge predecessors of an $N^\#$ node in time proportional to the size of the set.
- (v) It is possible to iterate over the set of all $E^\#$, SummaryEdge, or PathEdge successors of an $N^\#$ node in time proportional to the size of the set.
- (vi) In the case of nodes of the form (s_p, d_1) , we make the further assumption that predecessor sets in $E^\#$ are indexed by call site. In other words, it is possible to retrieve only those predecessors that are associated with a particular call site, without having to examine all $E^\#$ predecessors of (s_p, d_1) . Similarly, in the case of nodes of the form (e_p, d_2) we assume that successor sets in $E^\#$ are indexed by return site. This assumption permits us to assume that the time required to identify the $\langle d_4, d_5 \rangle$ pairs used in the for-loop on line [27] is proportional to the number of such pairs.

One way to satisfy assumptions (i)-(v) is to use arrays to support unit-time membership testing, and one or more linked-lists for each $N^\#$ node (to support iteration over predecessors and successors). Assumption (vi) can be satisfied by associating, with each $N^\#$ node of the form (s_p, d_1) or (e_p, d_2) , an array of linked-lists, one list for each call-site.

Running Time

The running time of the Tabulation Algorithm varies depending on what class of dataflow-analysis problem it is applied 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	$O(ED^2)$	$O(Call B^2 D^2 + ED^3)$
h -sparse	At most $O(hD)$ edges/rep.-relation	$O(hED)$	$O(Call B^2 D^2 + Call D^3 + hED^2)$
(Locally) separable	Component-wise dependences	$O(ED)$	$O(ED)$

Table 4.2. Asymptotic running time of the Tabulation Algorithm for six different classes of dataflow-analysis problems.

Some of these bounds have been achieved previously: For intraprocedural problems, the algorithms of Hecht [15], Kou [23], and Khedker and Dhamdhere [19] have the same time bounds as are shown in column three.² For locally separable problems (both intraprocedural and interprocedural), the algorithm of Knoop and Steffen [22] also runs in time $O(ED)$.

We now present derivations of the bounds given in Table 4.2 for the three classes of interprocedural problems. (The bounds for the three classes of intraprocedural problems follow from simplifications of the arguments given below.)

Distributive Problems

For distributive problems, the representation relation for each edge $e \in (E^0 \cup E^1)$ can contain up to $O(D^2)$ edges. Instead of calculating the worst-case cost-per-iteration of the loop on lines [14]-[43] of Figure 4 and multiplying by the number of iterations, we break the cost of the algorithm down into three contributing aspects and bound

²The intraprocedural dataflow-analysis algorithms of Kou [23] and Khedker and Dhamdhere [19] are described as algorithms for separable problems; however, the ideas described in Section 3.2 concerning the representation of distributive functions can be used to immediately extend those algorithms to all distributive and h -sparse intraprocedural problems.

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 [25]-[36] of Figure 4), and
- (iii) the cost of closure steps (lines [17]-[24] and [37]-[41] of Figure 4).

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 $D_p + 1$ nodes in $N_p^\#$ of the form (s_p, d) . The edges of the multi-source reachability problem associated with p are

$$\{ ((m, d_1), (n, d_2)) \in E_p^\# \mid (m, n) \in (E_p^0 \cup E_p^1) \} \cup \{ ((m, d_1), (n, d_2)) \in \text{SummaryEdge} \mid m \in \text{Call}_p \}.$$

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 now wish to compute a bound on the cost of installing summary edges at call sites (lines [25]-[36] of Figure 4). For each summary edge $((c, d_4), (\text{succ}^1(c), d_5))$, the conditional statement on lines [28]-[33] will be executed some number of times (on different iterations of the loop on lines [14]-[43]). In particular, line [28] will be executed every time the Tabulation Algorithm finds a three-edge path of the form

$$(((c, d_4), (s_p, d_1)), ((s_p, d_1), (e_p, d_2)), ((e_p, d_2), (\text{succ}^1(c), d_5))) \quad (\dagger)$$

as shown in the diagram marked “Line [29]” of Figure 5.

When we consider the set of all summary edges at a given call site c : $\{ ((c, d_4), (\text{succ}^1(c), d_5)) \}$, the executions of line [28] can be placed in three categories:

$d_4 \neq \Lambda$ and $d_5 \neq \Lambda$

There are at most D^2 choices for a $\langle d_4, d_5 \rangle$ pair, and for each such pair at most B^2 possible three-edge paths of the form (\dagger) .

$d_4 = \Lambda$ and $d_5 \neq \Lambda$

There are at most D choices for d_5 and for each such choice at most BD possible three-edge paths of the form (\dagger) .

$d_4 = \Lambda$ and $d_5 = \Lambda$

There is only one possible three-edge path of the form (\dagger) .

Thus, the total cost of all executions of line [28] is bounded by $O(\text{Call } B^2 D^2)$.

Because of the test on line [28], the code on lines [29]-[32] will be executed exactly *once* for each possible summary edge. In particular, for each summary edge the cost of the loop on lines [30]-[32] is bounded by $O(D)$. Since the total number of summary edges that can possibly be acquired by procedure p is bounded by $\text{Call}_p D^2$, the total cost of lines [29]-[32] is $O(\text{Call } D^3)$. Thus, the total cost of installing summary edges during the Tabulation Algorithm is bounded by $O(\text{Call } B^2 D^2 + \text{Call } D^3)$.

To bound the total cost of the closure steps, we use a variation on the argument used by Yellin to obtain a bound for the running time of an algorithm for dynamic transitive closure [34].³ 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 $n \notin \text{Ret}$, the only way the

³The closure steps can be thought of as a variant of Yellin’s algorithm, modified as follows:

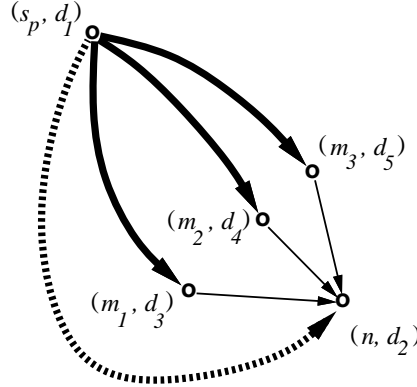
- (i) Yellin’s algorithm is an algorithm for dynamic transitive closure—*i.e.*, dynamic *all-pairs* reachability. The closure steps implement dynamic *multi-source* reachability, where the sources in each procedure’s exploded flow graph are the anchor sites.
- (ii) After a summary edge is inserted into the exploded flow graph of procedure p , rather than re-closing the graph immediately, the code in Figure 4 uses a single worklist to organize the simultaneous closure of all the exploded flow graphs.

Feature (ii) is not essential. The code could have been written to perform the closure of an exploded flow graph immediately, at the same time accumulating pending summary edges that need to be inserted in other exploded flow graphs in an auxiliary set.

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:



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 $\text{indegree}(n, d_2)$. For such an anchor site, the total cost of acquiring all its outgoing path edges can be bounded by

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

The accounting for the case of a node $n \in \text{Ret}$ 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 say that each node (n, d_2) , where $n \in \text{Ret}$, has the maximum possible number of incoming summary edges, namely D . Because there are at most $\text{Call}_p D$ nodes of $N_p^\#$ of the form (n, d_2) , where $n \in \text{Ret}$, 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

$$O\left(\sum_{\substack{(n, d_2) \in N_p^\# \\ \text{and } n \in \text{Ret}}} (\text{indegree}(n, d_2) + \text{summary-edge-indegree}(n, d_2))\right) = O(\text{Call}_p D^2).$$

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

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

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

h-Sparse Problems

The bound on the cost of the Tabulation Algorithm for *h*-sparse problems is obtained by an argument similar to the one given for distributive functions, except that we modify the accounting to take into account the fact that each representation relation can contain at most $O(hD)$ edges.

As before, the total cost of installing summary edges is bounded by $O(\text{Call } B^2 D^2 + \text{Call } D^3)$. However, the bound on the total cost of closure steps is slightly different from the bound obtained in the case of distributive problems:

$$\begin{aligned} \text{Cost of closure steps} &= \sum_p (\# \text{ anchor sites}) \times O(\text{Call}_p D^2 + \sum_{(n, d) \in N_p^\#} \text{indegree}(n, d)) \\ &= O(D) \times O(\sum_p (\text{Call}_p D^2 + hE_p D)) \\ &= O(\text{Call } D^3 + hED^2). \end{aligned}$$

Thus, on *h*-sparse problems, the total running time of the Tabulation Algorithm is bounded by

$$O(Call B^2 D^2 + Call D^3 + hED^2).$$

Locally Separable Problems

For locally separable problems, within a given procedure of the exploded super-graph all representation relations for edge functions have essentially component-wise dependences. More precisely, for all edges $(m, n) \in (E^0 \cup E^1)$, in the representation relation $R_{M(m, n)}$ each node (n, d) can only be the target of a single edge—one whose source is (m, d) or one whose source is (m, Λ) .

Recall that in locally separable problems different call sites on the same procedure may make use of different T functions to map dataflow information across to the called procedure. However, the Tabulation Algorithm tabulates only *same-level* realizable paths, and because of the additional restrictions on the binding functions in T for locally separable problems (see Definition 3.12), same-level realizable paths are always of the form $[(m, d): \dots (n, d)]$, $[(m, \Lambda): \dots (n, d)]$, or $[(m, \Lambda): \dots (n, \Lambda)]$. Note that the maximum number of summary edges in a procedure p is $O(Call_p D)$ (rather than $O(Call_p D^2)$, as we have in the case of distributive and h -sparse problems).

For locally separable problems, we can give a better bound on the number of times line [28] is executed. Again, we consider the set of all summary edges at a given call site c : $\{((c, d_4), (succ^1(c), d_5))\}$, and place the executions of line [28] in three categories:

$d_4 \neq \Lambda$ and $d_5 \neq \Lambda$

Because summary edges represent same-level realizable paths, it must be that $d_4 = d_5$, and thus there are at most D choices for a $\langle d_4, d_5 \rangle$ pair. For each such pair there is at most one possible three-edge path of the form (\dagger) .

$d_4 = \Lambda$ and $d_5 \neq \Lambda$

There are at most D choices for d_5 and for each such choice at most three possible three-edge paths of the form (\dagger) .

$d_4 = \Lambda$ and $d_5 = \Lambda$

There is only one possible three-edge path of the form (\dagger) .

Thus, the total number of executions of line [28] is bounded by $O(Call D)$.

Again, because of the test on line [28], the code on lines [29]-[32] will be executed exactly once for each possible summary edge, but for locally separable problems there will be at most two calls on Propagate generated at line [31]:

[31] Propagate(PathEdge, $((s_{fg(c)}, d_3), (succ^1(c), d_5))$, WorkList).

In one call, $d_3 = d_5$; in the other, $d_3 = \Lambda$. Because the total number of summary edges in all procedures is bounded by $O(Call D)$, the total cost of lines [29]-[32] is $O(Call D)$.

Thus, the total cost of installing summary edges is bounded by $O(Call D)$.

To bound the total cost of closure steps, we use the same argument as in the distributive case, except that we observe that for locally separable problems the indegree and summary-edge-indegree of a node can both be at most two. Thus,

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

Therefore on locally separable problems, the total running time of the Tabulation Algorithm is bounded by $O(ED)$.

Intraprocedural Problems

Because $Call = 0$ for intraprocedural problems, the bounds on the Tabulation Algorithm's running time on distributive, h -sparse, and separable interprocedural problems simplify to $O(ED^3)$, $O(hED^2)$, and $O(ED)$, respectively, in the intraprocedural case. In the case of distributive and h -sparse problems, we can sharpen these bounds by observing that in the program's one procedure, the Tabulation Algorithm uses at most $C + 1$ (rather than D) anchor sites—the $C^\#$ nodes of $G_{Ip}^\#$. Thus, it is more precise to say that the bounds for distributive and h -sparse intraprocedural problems are $O(CED^2)$ and $O(hCED)$, respectively.

These bounds can be reduced further if we assume that C , the distinguished dataflow value associated with node s_{main} of the IFDS problem, has the value \emptyset . This assumption involves no loss of generality, because if $C \neq \emptyset$, the problem can always be turned into one of that form as follows: a new start node \bar{s}_{main} is added to pro-

cedure *main* with a single outgoing edge $(\bar{s}_{main}, s_{main})$ and edge-function assignment $M(\bar{s}_{main}, s_{main}) = \lambda S.C$. After this preprocessing step is applied, $|C| = 1$, and the running time on distributive and h -sparse problems drops to $O(ED^2)$ and $O(hED)$, respectively, as reported in Table 4.2.

The transformation can also be applied when dealing with separable intraprocedural problems and the three kinds of interprocedural problems. In these cases it reduces the number of tabulation steps carried out to analyze the main procedure of the program; however, it does not lead to improvements in the worst-case running time.

We will rely on this transformation again in Section 5.1, and a related one in Section 7.1.

Storage Costs

The storage requirements for the Tabulation Algorithm consist of the storage for graph $G_{IP}^\#$ and for the three sets WorkList, PathEdge, and SummaryEdge.

The source of a path edge is always a node of the form (s_p, d) ; that is, it is always associated with some procedure p 's start node. Furthermore, the source and target of a path edge are always in the same procedure. Thus, the size of PathEdge is at most ND^2 . Similarly, the source and target of each summary edge are always associated with a matching call-site/return-site pair, and so the size of SummaryEdge is at most $Call D^2$. Finally, the size of $G_{IP}^\#$ itself is bounded by $O(ED^2)$. Consequently, the total storage for the linked-lists needed to implement unit-time set-insertion operations, which is proportional to $|E^\#| + |\text{PathEdge}| + |\text{SummaryEdge}|$, is bounded by $O(ED^2)$.

By the same observations about the form of path edges, $k + 1$ separate arrays of dimensions $(D_p + 1) \times N_p \times (D_p + 1)$, corresponding to procedures $0, 1, \dots, k$, can be used to implement a unit-time test for membership in PathEdge. Therefore, the total amount of storage needed for these arrays is

$\sum_{p \in \{0, \dots, k\}} N_p (D_p + 1)^2 = O(ND^2)$. Similarly, $Call$ separate arrays of dimensions $(D_p + 1) \times (D_p + 1)$, corresponding to the different call-sites in the program, can be used to implement a unit-time test for membership in SummaryEdge. Thus, the total amount of storage needed for these arrays is $O(Call D^2)$.

Because a path edge can be inserted into WorkList at most once, (*i.e.*, when the edge is inserted into PathEdge), the size of WorkList is also bounded by $O(ND^2)$.

Thus, the total amount of storage used by the Tabulation Algorithm is bounded by $O(ED^2)$.

4.2. Precise Versus Imprecise Solutions to Interprocedural Problems

Imprecise answers to interprocedural dataflow-analysis problems can always 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* versus considering only *realizable paths*.

Table 4.3 compares the cost of obtaining precise answers using the Tabulation Algorithm with the cost of obtaining imprecise answers:⁴

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

Table 4.3. Comparison of asymptotic running times for finding imprecise versus precise answers to interprocedural dataflow-analysis problems.

Note that column three of Table 4.3 is different from column three of Table 4.2. In a true intraprocedural problem the program consists of only a single flow graph G_0 ; however, when an interprocedural dataflow-analysis

⁴The bounds in column three on the cost of obtaining imprecise answers apply not only to the Tabulation Algorithm but also to the algorithms of Hecht [15], Kou [23], and Khedker and Dhamdhere [19], which can all be viewed as linear-time graph-reachability algorithms.

problem is treated as if it were one large intraprocedural problem, the “flow graph” is the super-graph, which consists of multiple flow graphs connected by edges from set E^2 . In the case of general distributive and h -sparse problems, the edges from E^2 contribute $O(\text{Call } BD)$ edges to the exploded flow graph of the “intraprocedural” problem.

When a *locally separable* interprocedural problem is treated as if it were one large intraprocedural problem, the problem may not be *separable*. Although within each procedure all representation relations for edge functions have essentially component-wise dependences, this may not be the case at procedure-call boundaries. However, the problem is always 1-sparse (since $B = 1$ in locally separable problems), so the asymptotic running time is bounded by $O(ED)$.

Because the quantity B is typically a small constant, and in many cases is simply 1, it is instructive also to consider these cost expressions after factors of B are dropped:

Class of functions	Graph-theoretic characterization of the dataflow functions’ properties	Asymptotic running time	
		Imprecise interprocedural solutions	Precise interprocedural solutions
Distributive	Up to $O(D^2)$ edges/rep.-relation	$O(ED^2)$	$O(ED^3)$
h -sparse	At most $O(hD)$ edges/rep.-relation	$O(hED)$	$O(\text{Call } D^3 + hED^2)$
Locally separable	Component-wise dependences	$O(ED)$	$O(ED)$

Table 4.4. Comparison of asymptotic running times for finding imprecise versus precise answers to interprocedural dataflow-analysis problems, neglecting factors of B .

Table 4.4 lets us get a feel for the extra cost involved in obtaining the more precise (realizable-path) answers. In the case of general distributive problems, the “penalty” is a factor of D . In the case of locally separable problems, there is no penalty at all—both kinds of solutions can be obtained in time $O(ED)$.

The fact that there is no penalty for finding the more precise answers for locally separable problems was observed by Knoop and Steffen [22]. What we have shown is that this can also be achieved by an algorithm that handles the more general distributive and h -sparse problem classes; when applied to locally separable problems, the algorithm “adapts” to exhibit $O(ED)$ behavior. No extra code is needed to handle the locally separable problems as a special case.

5. Compression

In this section, we present an alternative tabulation-based approach that involves transforming the original problem into a compressed problem whose size is related to the number of call sites in the program. This “compression step” is similar to the kind of problem transformations implicit in the creation of the “program summary graph” that is used by Callahan for solving flow-sensitive side-effect problems [6] and the “system dependence graph” that is used by Horwitz, Reps, and Binkley for interprocedural program slicing [18].

As we shall see, for a “one-shot” solution of a dataflow-analysis problem, the compression-based algorithm is probably not the algorithm of choice since the cost of the initial compression step could be more expensive than the cost of the best algorithm for solving the problem. However, as pointed out by Callahan in his work on flow-sensitive side-effect problems, compression may be useful when dealing with the incremental and/or interactive context, where the program undergoes a sequence of small changes, after each of which dataflow information is to be reported. It is possible to reuse the compressed structure for each unchanged procedure; one only has to re-compress procedures that have been changed, which is ordinarily no more than a small percentage of the entire program. The compressed version can then be solved more efficiently than the original uncompressed problem. (The factor saved depends on the ratio between the total number of “program points” and the total number of call sites.)

The **Compressed-Tabulation Algorithm** is presented in Figures 6 and 7. It carries out three steps:

Compress: Compress the Problem (lines [1], [7]-[24])

The first step is to transform the original problem to one that deals only with the original program’s start, exit, call, and return-site nodes. Procedure *Compress* is a worklist algorithm that solves a collection of purely *intraprocedural* reachability problems. *Compress* follows only the edges of $E^\#$ that correspond to E^0 edges

```

procedure SolveViaCompression( $G_{IP}^\#$ )
begin
[1]   $\bar{G} := \text{Compress}(G_{IP}^\#)$ 
[2]   $\text{SolveViaTabulation}(\bar{G})$ 
[3]   $\text{RealizablePath} := \text{FindRealizablePaths}()$ 
[4]  /* Values for  $X_n$ , for all  $n \in (\text{Start} \cup \text{Exit} \cup \text{Call} \cup \text{Ret})$  have already been determined by SolveViaTabulation */
[5]  for  $m \in N^* - (\text{Start} \cup \text{Exit} \cup \text{Call} \cup \text{Ret})$  do
[6]     $X_m := \{ d \in D_{fg(m)} \mid \exists c \text{ such that } ((s_{main}, c), (m, d)) \in \text{RealizablePath} \}$ 
[7]  od
end

function  $\text{Compress}(G_{IP}^\#)$  returns exploded super-graph
begin
[8]  Let  $(N^\#, E^\#, C^\#) = G_{IP}^\#$ 
[9]   $\text{CompressedEdges} := \emptyset$ 
[10]  $\text{WorkList} := \emptyset$ 
[11] for each  $p \in \{0, \dots, k\}$  and  $d \in (D_p \cup \Lambda)$  do
[12]   Insert  $((s_p, d), (s_p, d))$  into  $\text{CompressedEdges}$ ; Insert  $((s_p, d), (s_p, d))$  into  $\text{WorkList}$ 
[13]   for each  $m \in \text{Ret}_p$  do
[14]    Insert  $((m, d), (m, d))$  into  $\text{CompressedEdges}$ ; Insert  $((m, d), (m, d))$  into  $\text{WorkList}$ 
[15]   od
[16] od

[17] while  $\text{WorkList} \neq \emptyset$  do
[18]   Select and remove an edge  $((a, d_1), (n, d_2))$  from  $\text{WorkList}$ 
[19]   for each  $(m, d_3) \in N_{fg(a)}^\#$  such that  $(n, m) \in E_{fg(a)}^0$  and  $((n, d_2), (m, d_3)) \in E^\#$  do
[20]    Propagate( $\text{CompressedEdges}, ((a, d_1), (m, d_3)), \text{WorkList}$ )
[21]   od
[22] od

[23]  $\bar{N} := \{ (n, d) \in N^\# \mid n \in (\text{Start} \cup \text{Exit} \cup \text{Call} \cup \text{Ret}) \}$ 
[24]  $\bar{E} := \{ ((m, d_1), (n, d_2)) \mid m, n \in (\text{Start} \cup \text{Exit} \cup \text{Call} \cup \text{Ret}) \text{ and } ((m, d_1), (n, d_2)) \in \text{CompressedEdges} \}$ 
        $\cup \{ ((m, d_1), (n, d_2)) \in E^\# \mid (m, n) \in (E^1 \cup E^2) \}$ 
[25] return( $\bar{N}, \bar{E}, C^\#$ )
end

```

Figure 6. The compression phase transforms the original problem to one that deals only with the original program’s start, exit, call, and return-site nodes. The compressed problem is solved by the Tabulation Algorithm, which gives answers for the start, exit, call, and return-site nodes. From these, answers for the nodes of the full problem are determined. (See also Figure 7.)

```

function FindRealizablePaths() returns set of edges
begin
[26] RealizablePath :=  $\emptyset$ 
[27] WorkList :=  $\emptyset$ 
[28] for each  $(s_{main}, c) \in C^\#$  do
[29]   Insert  $((s_{main}, c), (s_{main}, c))$  into RealizablePath
[30]   Insert  $((s_{main}, c), (s_{main}, c))$  into WorkList
[31] od
[32] while WorkList  $\neq \emptyset$  do
[33]   Select and remove an edge  $((s_{main}, c), (n, d_1))$  from WorkList
[34]   switch  $n$ 
[35]     case  $n \in Call_p$  :
[36]       for each  $d_2$  such that  $((n, d_1), (s_{calledProc(n)}, d_2)) \in E^\#$  do
[37]         Propagate(RealizablePath,  $((s_{main}, c), (s_{calledProc(n)}, d_2))$ , WorkList)
[38]       od
[39]       for each  $d_2$  such that  $((n, d_1), (succ^1(n), d_2)) \in (E^\# \cup SummaryEdge)$  do
[40]         Propagate(RealizablePath,  $((s_{main}, c), (succ^1(n), d_2))$ , WorkList)
[41]       od
[42]     end case
[43]     case  $n \in (N_p - Call_p - \{e_p\})$  :
[44]       for each  $(m, d_2)$  such that  $((n, d_1), (m, d_2)) \in E^\#$  do
[45]         Propagate(RealizablePath,  $((s_{main}, c), (m, d_2))$ , WorkList)
[46]       od
[47]     end case
[48]   end switch
[49] od
[50] return(RealizablePath)
end

```

Figure 7. Function FindRealizablePaths determines which nodes in $G_{IP}^\#$ are reachable from a node in $C^\#$ via a realizable path. (See also Figure 6.)

of the super-graph (cf. line [18]), and returns a compressed version of exploded super-graph $G_{IP}^\#$.

Solve: Solve the Compressed Problem (line [2])

The second phase is to solve the compressed problem by invoking the Tabulation Algorithm on the compressed graph.

Extend: Determine Values for the Nodes of the Full Problem (lines [4]-[7] of Figure 6, lines [26]-[50] of Figure 7)

As noted in the comment on line [4], at the end of Step 2 values in the meet-over-all-valid-paths solution of the dataflow-analysis problem are already known for the original program's start, exit, call, and return-site nodes. The third and final step is to determine solution values for all other nodes of the super-graph.

Function FindRealizablePaths (Figure 7) is also a worklist algorithm. On each iteration of its main loop it deduces the existence of *realizable paths* from the nodes of $C^\#$. This is in contrast to the other algorithms presented in the paper so far, which tabulate only *same-level realizable paths*. These paths are recorded in set RealizablePath. The configurations that are used by FindRealizablePaths to deduce the existence of additional realizable paths from (s_{main}, c) are depicted in Figure 8.

We leave it to the reader to convince himself of the algorithm's correctness: when the Compressed-Tabulation Algorithm terminates, the value of X_n is the value for node n in the meet-over-all-valid-paths solution to IP .

5.1. Cost of the Compressed-Tabulation Algorithm

Table 5.1 compares the asymptotic running times of the the Tabulation Algorithm, the Compressed-Tabulation

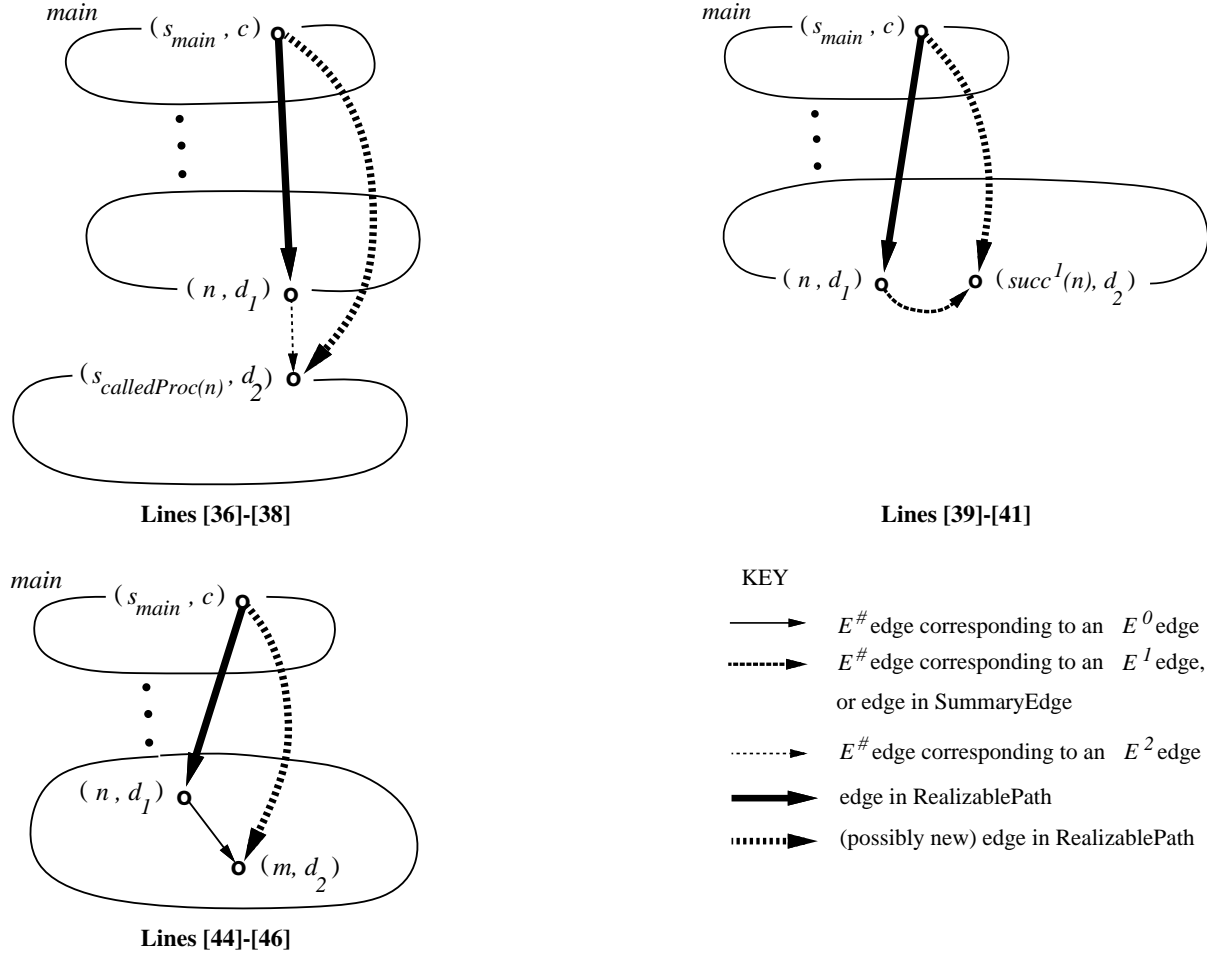


Figure 8. The above three diagrams show the situations handled in lines [36]-[38], [39]-[41], and [44]-[46] of Figure 7.

Algorithm, and the Horwitz-Reps-Binkley algorithm.⁵

⁵The Horwitz-Reps-Binkley algorithm is presented in [18] as an algorithm for interprocedural slicing. To use it for interprocedural dataflow analysis, only minor changes are necessary. Graph $G_{IP}^\#$ takes the place of the “system dependence graph” (before “characteristic-graph edges” are added). Problem instance IP is then solved as follows:

- (i) Create the “linkage grammar” for $G_{IP}^\#$, compute its characteristic-graph edges, and add them to $G_{IP}^\#$.
- (ii) Perform a forward slice of the augmented graph starting from the $C^\#$ nodes.
- (iii) For each $n \in N^\#$, set X_n to be $\{d \in D_{fg(n)} \mid (n, d) \text{ was reached in the forward slice}\}$.

Class of functions	Phase of algorithm	Asymptotic running time		
		Tabulation Algorithm	Compressed-Tabulation Algorithm	Horwitz-Reps-Binkley Algorithm
Distributive	Compress		$O(D^3 \sum_p \text{Call}_p E_p)$	$O(D^3 \sum_p \text{Call}_p E_p)$
	Solve	$O(\text{Call } B^2 D^2 + ED^3)$	$O(\text{Call } B^2 D^2 + D^3 \sum_p \text{Call}_p^2)$	$O(B^2 D^4 \sum_p \text{Call}_p^3)$
	Extend		$O(ED^2)$	$O(ED^2)$
h -sparse	Compress		$O(hD^2 \sum_p \text{Call}_p E_p)$	$O(hD^2 \sum_p \text{Call}_p E_p)$
	Solve	$O(\text{Call } B^2 D^2 + \text{Call } D^3 + hED^2)$	$O(\text{Call } B^2 D^2 + D^3 \sum_p \text{Call}_p^2)$	$O(B^2 D^4 \sum_p \text{Call}_p^3)$
	Extend		$O(\text{Call } D^2 + hED)$	$O(\text{Call } D^2 + hED)$
Locally separable	Compress		$O(D \sum_p \text{Call}_p E_p)$	$O(D \sum_p \text{Call}_p E_p)$
	Solve	$O(ED)$	$O(D \sum_p \text{Call}_p^2)$	$O(D \sum_p \text{Call}_p^3)$
	Extend		$O(ED)$	$O(ED)$

Table 5.1. Comparison of the asymptotic running times of the Tabulation Algorithm, the Compressed-Tabulation Algorithm, and the Horwitz-Reps-Binkley Algorithm for three different classes of interprocedural dataflow-analysis problems.

The arguments used to obtain the bounds for the Compressed-Tabulation Algorithm and the Horwitz-Reps-Binkley Algorithm are similar to the indegree-counting arguments given in Section 4.1. The bounds for the Solve steps in columns four and five reflect the fact that the Compress step can cause a quadratic explosion of edges to occur. For example, in the case of distributive problems, the compressed exploded flow graph of a procedure p can have $\Omega(\text{Call}_p^2 D^2)$ edges. Thus, for distributive problems, the quantity “ E ” in Table 4.2 has to be taken to be $\sum_p \text{Call}_p^2$.

There is also a subtle point involved in showing that the the worst-case running time of the Extend Step of the Compressed-Tabulation Algorithm (*i.e.*, function FindRealizablePaths and lines [4]-[7] of Figure 6) meets the bounds claimed in Table 5.1. As presented in Figure 7, FindRealizablePaths tabulates realizable paths from *all* nodes of the form $(s_{\text{main}}, c) \in C^\#$. This means that there are $C + 1$ anchor sites, and so—by the kind of indegree-counting arguments used in Section 4.1—one obtains bounds of $O(CED^2)$, $O(\text{Call } CD^2 + hCED)$, and $O(CED)$ for distributive, h -sparse, and locally separable problems, respectively, which is a factor of C greater than claimed. Fortunately, we can fall back on the problem transformation described at the end of Section 4.1 and assume that C , the distinguished dataflow value associated with node s_{main} of the IFDS problem, has the value \emptyset . This involves no loss of generality, and for problems in this form $C = 1$, so the bounds on the running time match the ones reported in Table 5.1.

The Horwitz-Reps-Binkley algorithm is also a compression algorithm; in particular, the step of constructing the “linkage grammar” is similar to the compression step of the Compressed-Tabulation Algorithm: Both steps solve a collection of purely local reachability problems to turn a realizable-path reachability problem into a compressed problem whose size is related to the number of call sites in the program. The bounds given in Table 5.1 for the Compressed-Tabulation Algorithm are better than the corresponding bounds for the Horwitz-Reps-Binkley algorithm, and suggest that the Compressed-Tabulation Algorithm is likely to outperform the Horwitz-Reps-Binkley algorithm. In fact, the Compressed-Tabulation Algorithm is an *asymptotically faster* algorithm than the Horwitz-Reps-Binkley algorithm: there is a family of examples on which the Horwitz-Reps-Binkley algorithm actually takes the number of steps shown in Table 5.1. (In other words, this is not just a case of being able to give a better bound for the Tabulation Algorithm because of better analysis techniques.)

Surprisingly, due to the cost of the compression step (Step 1) the asymptotic running times of the Compressed-Tabulation Algorithm and the Horwitz-Reps-Binkley algorithm may actually be *worse* than the running time of the Tabulation Algorithm. However, as argued earlier, in the incremental and/or interactive context, where the program undergoes a sequence of small changes, after each of which dataflow information is to be reported, an algorithm that uses compression may outperform one that does not.

In the incremental/interactive contexts only the changed procedures need to be re-compressed, and therefore the cost of the re-compression step is not likely to be the dominant cost. However, in the case of distributive

problems and h -sparse problems, it is not possible to say that the Compressed-Tabulation Algorithm is asymptotically superior to the Tabulation Algorithm or vice versa. In both cases, the expressions for the bounds on the running time have incomparable terms: $D^3 \sum_p \text{Call}_p^2$ versus ED^3 in the case of distributive problems, and $D^3 \sum_p \text{Call}_p^2$ versus hED^2 in the case of h -sparse problems.

In the case of locally separable problems, the bound on the running time for the Tabulation Algorithm is not comparable to the bound for the Solve step of the Compressed-Tabulation Algorithm ($O(ED)$ versus $O(D \sum_p \text{Call}_p^2)$), but matches the bound for the Extend step. (Because there is a family of examples on which the Compressed-Tabulation Algorithm actually takes the number of steps shown in Table 5.1, outside the incremental/interactive context—when the Compress step comes into play—we can say that the Tabulation Algorithm is an asymptotically faster algorithm than the Compressed-Tabulation Algorithm.)

It would be fair to say that this analysis of the Compressed-Tabulation Algorithm is inconclusive: experimental evaluation of the algorithm is necessary to determine whether in practice it will outperform the Tabulation Algorithm in the interactive/incremental context.

6. Demand Interprocedural Dataflow Analysis

This section concerns the solution of demand versions of interprocedural dataflow-analysis problems. In demand problems, dataflow information is to be reported only for a single program element of interest (or a small number of elements of interest).

Because the dataflow information at one program point typically depends on dataflow information from other points, a **demand algorithm** must minimize the number of *other* points for which (transient) summary information is computed and/or the amount of information computed at those points. Because multiple demands may be issued, another important goal is to maximize reuse of intermediate results computed in the course of answering previous queries. Algorithms that are able to do this we call **caching demand algorithms**.

There are several reasons why it is desirable to solve the demand versions of interprocedural analysis problems.

- *Narrowing the focus to specific flow-graph nodes of interest.* In program optimization, most of the gains are obtained from making improvements at a program’s “hot spots”—in particular, its innermost loops. Although the optimization phases during which transformations are applied can be organized to concentrate on hot spots, there is typically an earlier phase to determine dataflow facts during which an exhaustive algorithm for interprocedural dataflow analysis is used. Although it will not change the worst-case asymptotic cost of performing dataflow analysis, there is good reason to believe that a demand algorithm will greatly reduce the amount of extraneous information computed.
- *Narrowing the focus to specific dataflow facts of interest.* Even when dataflow information is desired for every flow-graph node n , the full set of dataflow facts at n may not be required. For example, the solution to the possibly-uninitialized variables problem is usually used to report the places in a program where uninitialized variables are used. Although only certain variables are used at each program point, the possibly-uninitialized variables problem associates the full set of uninitialized variables with each program point. Therefore it may be advantageous to use a demand algorithm for dataflow analysis (without changing the definition of the problem to be solved) and for each program point n and variable v used at n issue the query “Is v in the set of possibly-uninitialized variables at n ?”
- *Sidestepping incremental-updating problems.* An optimizing transformation performed at one point in the program can invalidate previously computed dataflow information at other points in the program. In some cases, the old information at such points is not a “safe” summary of the possible execution states that can arise there; the dataflow information needs to be updated before it is possible to perform optimizing transformations at such points. However, no good incremental algorithms for interprocedural dataflow analysis are currently known.

An alternative is to use an algorithm for the demand version of the dataflow-analysis problem and have the optimizer place appropriate demands. With each demand, the algorithm would be invoked on the *current* program. (As indicated above, any information cached from previous queries would be discarded whenever the program is modified.)

- *Demand analysis as a user-level operation.* It is desirable to have program-development tools in which the user can interactively ask questions about various aspects of a program [26,33,25,16]. Such tools are particularly useful when debugging, when trying to understand complicated code, or when trying to transform a program to execute efficiently on a parallel machine.

Because we have been able to formulate interprocedural dataflow-analysis problems as realizable-path reachability problems, an algorithm for demand interprocedural dataflow analysis can be obtained by finding a demand algorithm for the *single-sink realizable-path reachability problem*. This problem asks “What are all the points q for which there exists a realizable path from q to a given point p ?”⁶ Although the Horwitz-Reps-Binkley algorithm (for “backward slicing”) can be applied to the single-sink realizable-path reachability problem, as described in [18] that algorithm has a phase in which certain auxiliary information is computed by a preliminary *exhaustive* algorithm. In other words, although the single-sink realizable-path reachability problem is a demand problem the Horwitz-Reps-Binkley algorithm is not a true demand algorithm for that problem.

One way to obtain an improved demand algorithm for the single-sink realizable-path reachability problem has been described by Reps [27]. Another demand algorithm for this problem has been developed by Rosay [30]. The next section presents a third demand algorithm. Because Reps’s and Rosay’s algorithms are based on the Horwitz-Reps-Binkley algorithm, their running times are asymptotically worse than the running time of the algorithm presented in Section 6.1.

6.1. A Demand-Tabulation Algorithm for Realizable-Path Reachability Queries

We now present the **Demand-Tabulation Algorithm**, which permits solving a sequence of demands for dataflow information. The top-level function of the algorithm is called `IsMemberOfSolution`, and is presented in Figure 11. Each call `IsMemberOfSolution(x, d_1)` returns **true** iff $d_1 \in MVP_x$, where MVP_x is the value for node x in the meet-over-all-valid-paths solution to the dataflow problem. It is assumed below that sets `PathEdge` and `SummaryEdge` have been initialized to \emptyset before the first call on function `IsMemberOfSolution` is performed.

`IsMemberOfSolution` includes a call to procedure `BackwardTabulateSLRPs`, shown in Figure 9.

The two procedures of the Demand-Tabulation Algorithm, when viewed in the right way, can be seen as the duals of procedures we have seen before in the Tabulation and Compressed-Tabulation Algorithms. In particular, `IsMemberOfSolution` is essentially the dual of procedure `FindRealizablePaths` of Figure 7; that is, it works in the opposite direction to `FindRealizablePaths`—counter to the direction of the edges in $G_{IP}^\#$. Similarly, `BackwardTabulateSLRPs` is the dual of procedure `ForwardTabulateSLRPs` of Figure 4.

Procedure `BackwardTabulateSLRPs` is a worklist algorithm that starts with a certain set of path edges, and on each iteration of the main loop deduces the existence of additional path edges (and summary edges). In this case, “path edges” refers to the subset of the same-level realizable paths of $G_{IP}^\#$ whose targets are nodes of the form (e_p, d_1) . The configurations that are used by `BackwardTabulateSLRPs` to deduce the existence of additional path edges are depicted in Figure 10; the five diagrams of Figure 10 correspond to lines [5]-[7], [8]-[10], [16], [17]-[19], and [25]-[27] of Figure 9. In Figure 10, the bold dotted arrows represent edges that are inserted into set `PathEdge` if they were not previously in that set.

Function `IsMemberOfSolution` (Figure 11) is also a worklist algorithm that on each iteration of the main loop deduces the existence of additional realizable paths to node (x, d_1) . These paths are recorded in `RealizablePath`.

The configurations that are used by `IsMemberOfSolution` to deduce the existence of additional realizable paths to (x, d_1) are depicted in Figure 12. Lines [37]-[41] of Figure 11, where `IsMemberOfSolution` initializes `WorkList` and calls `BackwardTabulateSLRPs`, play a key role. When a node of the form (n, d_2) , where $n \in Ret$, is processed by `IsMemberOfSolution`, `BackwardTabulateSLRPs` is used to find the summary edges from nodes of $G_{IP}^\#$ associated with n ’s corresponding call node. `BackwardTabulateSLRPs` adds additional edges to `SummaryEdge`. Note that if these summary edges have previously been computed, `BackwardTabulateSLRPs` will not perform any iterations of the loop in lines [1]-[30] of Figure 9 because the call on `Propagate` in line [39] of Figure 11 only adds an edge to `WorkList` if it is not already in `PathEdge`.

The Demand-Tabulation Algorithm is a caching demand algorithm: On all calls after the first call to `IsMemberOfSolution`, edges in `PathEdge` and `SummaryEdge` that have been inserted during earlier calls on `IsMemberOfSolution` are used to avoid redoing work that has been previously performed. Edges in those two sets signal that a previous call on `IsMemberOfSolution` had occasion to deduce the existence of a same-level realizable path in $G_{IP}^\#$, and this work is not repeated. The use of such saved, previously computed information can reduce the cost of responding to demands when there is a sequence of demands in between program modifications. (The accumulation of information can go on until such time as the program is modified, whereupon all previous results—which may no longer be safe—must be discarded.)

⁶The demand interprocedural dataflow-analysis problem is actually a single-source-single-sink realizable-path reachability problem: “Is there a realizable path from a given point q to a given point p ?” However, it is not known how to solve such problems more efficiently than the single-sink realizable-path reachability problem.

```

procedure BackwardTabulateSLRPs(WorkList)
begin
[1]  while WorkList  $\neq \emptyset$  do
[2]    Select and remove an edge  $((n, d_2), (e_p, d_1))$  from WorkList
[3]    switch  $n$ 
[4]      case  $n \in Ret_p$  :
[5]        for each  $d_3$  such that  $((e_{calledProc(pred^1(n))}, d_3), (n, d_2)) \in E^\#$  do
[6]          Propagate(PathEdge,  $((e_{calledProc(pred^1(n))}, d_3), (e_{calledProc(pred^1(n))}, d_3)),$  WorkList)
[7]        od
[8]        for each  $d_3$  such that  $((pred^1(n), d_3), (n, d_2)) \in (E^\# \cup \text{SummaryEdge})$  do
[9]          Propagate(PathEdge,  $((pred^1(n), d_3), (e_p, d_1)),$  WorkList)
[10]       od
[11]     end case
[12]     case  $n = s_p$  :
[13]       for each  $c \in callers(p)$  do
[14]         for each  $d_4, d_5$  such that  $((c, d_5), (s_p, d_2)) \in E^\#$  and  $((e_p, d_1), (succ^1(c), d_4)) \in E^\#$  do
[15]           if  $((c, d_5), (succ^1(c), d_4)) \notin \text{SummaryEdge}$  then
[16]             Insert  $((c, d_5), (succ^1(c), d_4))$  into SummaryEdge
[17]           for each  $d_3$  such that  $((succ^1(c), d_4), (e_{fg(c)}, d_3)) \in \text{PathEdge}$  do
[18]             Propagate(PathEdge,  $((c, d_5), (e_{fg(c)}, d_3)),$  WorkList)
[19]           od
[20]         fi
[21]       od
[22]     od
[23]     end case
[24]     case  $n \in (N_p - Ret_p - \{s_p\})$  :
[25]       for each  $(m, d_3)$  such that  $((m, d_3), (n, d_2)) \in E^\#$  do
[26]         Propagate(PathEdge,  $((m, d_3), (e_p, d_1)),$  WorkList)
[27]       od
[28]     end case
[29]   end switch
[30] od
end

```

Figure 9. Procedure BackwardTabulateSLRPs determines all same-level realizable paths to targets of the form (e_p, d_1) taken from WorkList. All summary edges found are accumulated in set SummaryEdge. (See also Figure 11.)

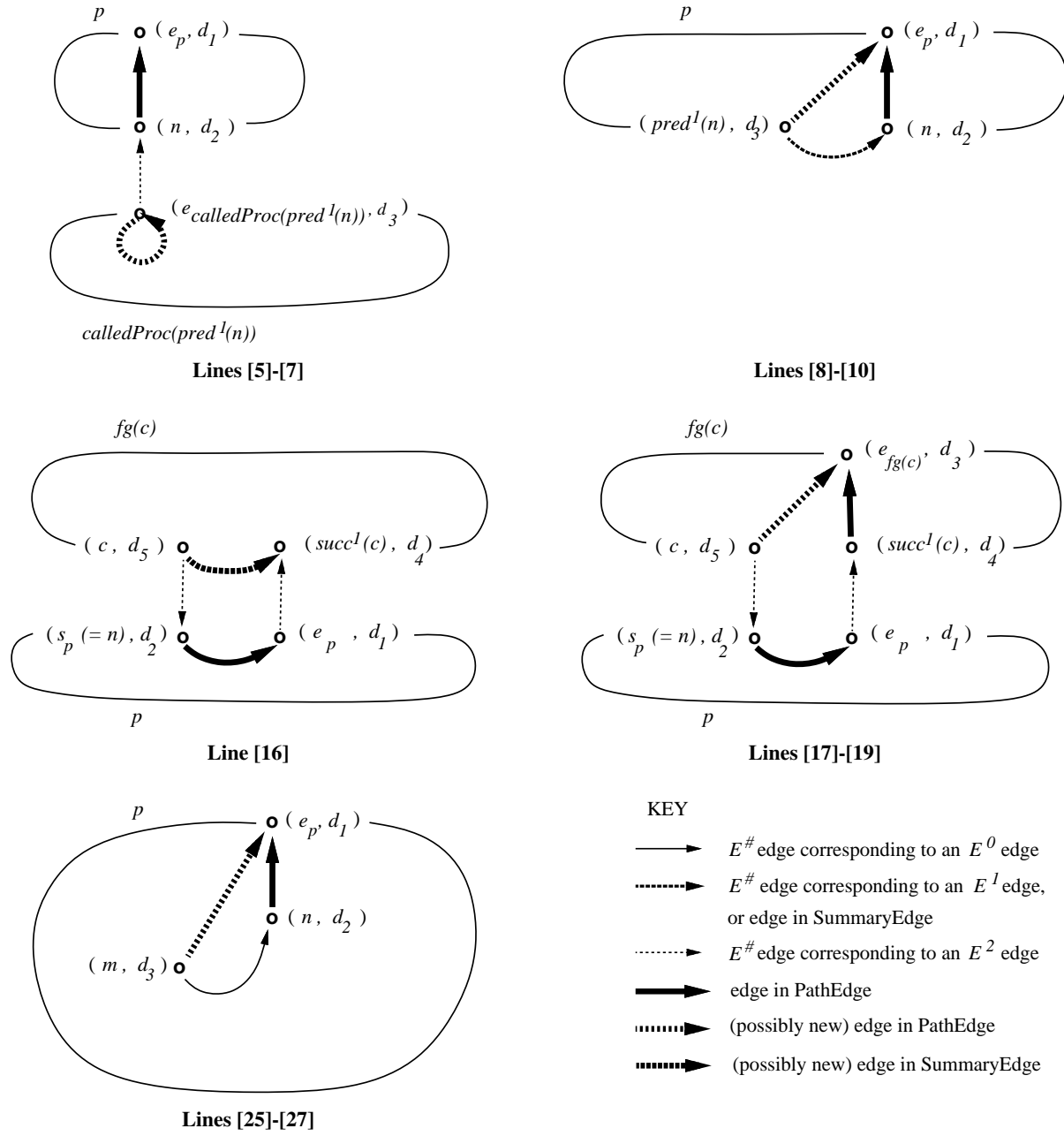


Figure 10. The above five diagrams show the situations handled in lines [5]-[7], [8]-[10], [16], [17]-[19], and [25]-[27] of Figure 9.

```

function IsMemberOfSolution( $x, d_1$ ) returns Boolean
begin
[31] RealizablePath := {  $((x, d_1), (x, d_1))$  }
[32] RPWorkList := {  $((x, d_1), (x, d_1))$  }
[33] while RPWorkList  $\neq \emptyset$  do
[34]   Select and remove an edge  $((n, d_2), (x, d_1))$  from RPWorkList
[35]   switch  $n$ 
[36]     case  $n \in Ret_p$  :
[37]       WorkList :=  $\emptyset$ 
[38]       for each  $d_3$  such that  $((e_{calledProc(pred^1(n))}, d_3), (n, d_2)) \in E^\#$  do
[39]         Propagate(PathEdge,  $((e_{calledProc(pred^1(n))}, d_3), (e_{calledProc(pred^1(n))}, d_3)),$  WorkList)
[40]       od
[41]       BackwardTabulateSLRPs(WorkList)
[42]       for each  $d_3$  such that  $((pred^1(n), d_3), (n, d_2)) \in (E^\# \cup \text{SummaryEdge})$  do
[43]         Propagate(RealizablePath,  $((pred^1(n), d_3), (x, d_1)),$  RPWorkList)
[44]       od
[45]     end case
[46]     case  $n = s_p$  :
[47]       for each  $c \in callers(p)$  do
[48]         for each  $d_3$  such that  $((c, d_3), (s_p, d_2)) \in E^\#$  do
[49]           Propagate(RealizablePath,  $((c, d_3), (x, d_1)),$  RPWorkList)
[50]         od
[51]       od
[52]     end case
[53]     case  $n \in (N_p - Ret_p - \{s_p\})$  :
[54]       for each  $(m, d_3)$  such that  $((m, d_3), (n, d_2)) \in E^\#$  do
[55]         Propagate(RealizablePath,  $((m, d_3), (x, d_1)),$  RPWorkList)
[56]       od
[57]     end case
[58]   end switch
[59] od
[60] return  $(\exists (s_{main}, c) \in C^\# \text{ such that } ((s_{main}, c), (x, d_1)) \in \text{RealizablePath})$ 
end

```

Figure 11. The Demand-Tabulation Algorithm performs a kind of “reverse tabulation” to determine whether $d_1 \in MVP_x$, where MVP_x is the value for node x in the meet-over-all-valid-paths solution to the dataflow problem. (See also Figure 9.)

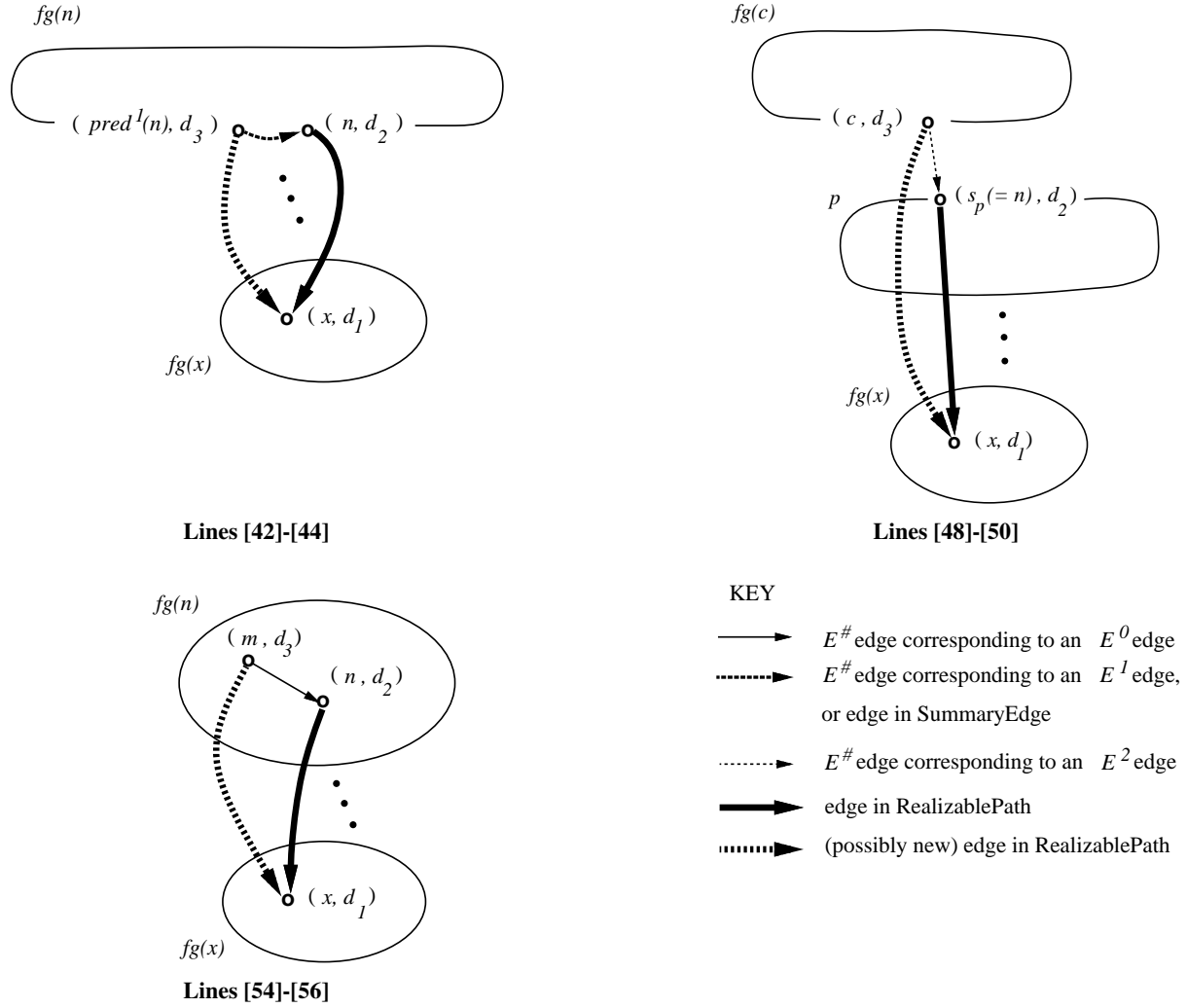


Figure 12. The above three diagrams show the situations handled in lines [42]-[44], [48]-[50], and [54]-[56] of Figure 11.

6.2. Cost of the Demand-Tabulation Algorithm

The arguments used to obtain the bounds for the Demand-Tabulation Algorithm are similar to the indegree-counting arguments given in Section 4.1 for the Tabulation Algorithm. The worst-case running time of `IsMemberOfSolution` is the same as that of `FindRealizablePaths`, whereas the worst-case running time of `BackwardTabulateSLRPs` is the same as that of `ForwardTabulateSLRPs`. Thus, the cost of `BackwardTabulateSLRPs` dominates the cost of the Demand-Tabulation Algorithm.

The following table shows how the Demand-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	$O(ED^2)$	$O(Call\ B^2D^3 + ED^3)$
h -sparse	At most $O(hD)$ edges/rep.-relation	$O(hED)$	$O(Call\ B^2D^3 + hED^2)$
(Locally) separable	Component-wise dependences	$O(E) \quad (\dagger)$	$O(ED)$

Table 6.1. Asymptotic running time of the Demand-Tabulation Algorithm (for answering a single query).

The only entry in Table 6.1 that differs from the corresponding entry in Table 4.2 is the entry for separable intraprocedural problems, marked above with (\dagger) , where the cost drops from $O(ED)$ to $O(E)$. The reason for this is that in a separable intraprocedural problem, the algorithm works backwards from a single anchor site (x, d_1) . The algorithm will only visit nodes in $G_{IP}^\#$ of the form (n, d_1) or (n, Λ) , and thus at most $O(E)$ work will be performed.

For the three classes of interprocedural problems, the worst-case performance of the Demand-Tabulation Algorithm is the same as that of the Tabulation Algorithm. However, in practice the Demand-Tabulation Algorithm should outperform the Tabulation Algorithm when there are only a small number of program elements at which dataflow information is to be reported.

As we observed earlier, the Demand-Tabulation Algorithm is a caching demand algorithm: it saves path edges and summary edges computed on previous queries in order to reduce the amount of work performed on subsequent queries. A desirable property for a demand algorithm to have is the same-total-cost property, which we define as follows:

Definition 6.2. A demand algorithm has the *same-total-cost property* with respect to an exhaustive algorithm \mathcal{A} if, for every input $G_{IP}^\#$, the total cost of any request sequence that places demands on all possible nodes of $G_{IP}^\#$ is proportional to the cost of a single run of algorithm \mathcal{A} on $G_{IP}^\#$.

□

When we compare the cost of calling `IsMemberOfSolution` ND times to satisfy a request sequence that places demands on all nodes of $G_{IP}^\#$ to the cost of calling `SolveViaTabulation` once, we find that the total time spent in `BackwardTabulateSLRPs` is proportional to the time spent in the one call on `ForwardTabulateSLRPs` made from procedure `SolveViaTabulation`. However, the version of `IsMemberOfSolution` presented in Figure 11 does not have the same-total-cost property with respect to the Tabulation Algorithm because a sequence of ND demands may perform multiple traversals of the edges of $G_{IP}^\#$ (in `IsMemberOfSolution` itself). For example, for distributive problems, the time spent in `IsMemberOfSolution` proper can be $\Omega(NED^3)$.

We now describe how to modify `IsMemberOfSolution` so that, over a request sequence that places demands on all possible nodes of $G_{IP}^\#$, each edge in $G_{IP}^\#$ is explored (by `IsMemberOfSolution` proper) exactly once. With this change, the Demand-Tabulation Algorithm has the same-total-cost property with respect to the Tabulation Algorithm. The modified version of `IsMemberOfSolution` is presented in Figure 13. The lines changed from Figure 11 are marked with “*”.

The key idea is to maintain one of three possible marks on each node (m, d) of $G_{IP}^\#$, with the following meanings:

Reachable

A realizable path from a $C^\#$ node to (m, d) definitely exists.

NotReachable

A realizable path from a $C^\#$ node to (m, d) definitely does not exist.

Unknown

It has not been established yet whether there is a realizable path from a $C^\#$ node to (m, d) .

At the end of a given invocation of `IsMemberOfSolution`, all nodes visited during that invocation are marked either “Reachable” or “NotReachable”. When any of these nodes are encountered on a subsequent invocation of `IsMemberOfSolution`, their reachability status is already known, and therefore it is unnecessary to explore their predecessors.

Before the first call on `IsMemberOfSolution` is performed, the $C^\#$ nodes are marked “Reachable”, all nodes with no predecessors are marked “NotReachable”, and all other nodes are marked “Unknown”.

```

function IsMemberOfSolution( $x, d_1$ ) returns Boolean
begin
[1]   RealizablePath := { ( $(x, d_1), (x, d_1)$ ) }
[2]   RPWorkList := { ( $(x, d_1), (x, d_1)$ ) }
[3]*  ReachableNodes :=  $\emptyset$ ; UnreachableNodes :=  $\emptyset$ 
[4]*  VisitedNodes :=  $\emptyset$ ; VisitedEdges :=  $\emptyset$ 
[5]   while RPWorkList  $\neq \emptyset$  do
[6]     Select and remove an edge  $((n, d_2), (x, d_1))$  from RPWorkList
[7]*  Insert  $(n, d_2)$  into VisitedNodes
[8]*  if  $(n, d_2)$  is marked “Reachable” then Insert  $(n, d_2)$  into ReachableNodes
[9]*  else if  $(n, d_2)$  is marked “NotReachable” then Insert  $(n, d_2)$  into UnreachableNodes
[10]* else /*  $(n, d_2)$  is marked “Unknown” */
[11]   switch  $n$ 
[12]     case  $n \in Ret_p$  :
[13]       WorkList :=  $\emptyset$ 
[14]       for each  $d_3$  such that  $((e_{calledProc(pred^1(n))}, d_3), (n, d_2)) \in E^\#$  do
[15]         Propagate(PathEdge,  $((e_{calledProc(pred^1(n))}, d_3), (e_{calledProc(pred^1(n))}, d_3)),$  WorkList)
[16]       od
[17]       BackwardTabulateSLRPs(WorkList)
[18]       for each  $d_3$  such that  $((pred^1(n), d_3), (n, d_2)) \in (E^\# \cup \text{SummaryEdge})$  do
[19]*      Insert  $((pred^1(n), d_3), (n, d_2))$  into VisitedEdges
[20]      Propagate(RealizablePath,  $((pred^1(n), d_3), (x, d_1)),$  RPWorkList)
[21]     od
[22]   end case
[23]   case  $n = s_p$  :
[24]     for each  $c \in callers(p)$  do
[25]       for each  $d_3$  such that  $((c, d_3), (s_p, d_2)) \in E^\#$  do
[26]*      Insert  $((c, d_3), (s_p, d_2))$  into VisitedEdges
[27]      Propagate(RealizablePath,  $((c, d_3), (x, d_1)),$  RPWorkList)
[28]     od
[29]   od
[30]   end case
[31]   case  $n \in (N_p - Ret_p - \{s_p\})$  :
[32]     for each  $(m, d_3)$  such that  $((m, d_3), (n, d_2)) \in E^\#$  do
[33]*      Insert  $((m, d_3), (n, d_2))$  into VisitedEdges
[34]      Propagate(RealizablePath,  $((m, d_3), (x, d_1)),$  RPWorkList)
[35]     od
[36]   end case
[37]   end switch
[38]*  fi
[39]  od
[40]* /* Find all reachable nodes that are reachable along visited edges */
[41]*  WorkList := ReachableNodes
[42]*  while WorkList  $\neq \emptyset$  do
[43]*    Select and remove a node  $(m, d)$  from WorkList
[44]*    for each successor  $(n, d')$  of  $(m, d)$  such that  $((m, d), (n, d')) \in \text{VisitedEdges}$  do
[45]*      if  $(n, d')$  is marked “Unknown” then
[46]*        Mark  $(n, d')$  “Reachable”; Insert  $(n, d')$  into WorkList
[47]*      Insert  $(n, d')$  into ReachableNodes
[48]*    fi
[49]*  od
[50]*  od
[51]* for each node  $(m, d) \in (\text{VisitedNodes} - \text{ReachableNodes})$  do Mark  $(m, d)$  “NotReachable” od
[52]* return(ReachableNodes  $\neq \emptyset$ )
end

```

Figure 13. A version of IsMemberOfSolution that has the same-total-cost property with respect to the Tabulation Algorithm. The lines changed from Figure 11 are marked with “*”.

During a given invocation of `IsMemberOfSolution`, the algorithm uses the sets `VisitedNodes` and `VisitedEdges` to keep track of all nodes and edges visited by `IsMemberOfSolution` itself (see lines [4], [7], [19], [26], and [33]). (Nodes and edges visited by `BackwardTabulateSLRPs` are not recorded in these sets; as pointed out above, it is not the time spent in `BackwardTabulateSLRPs` that prevents the Demand-Tabulation Algorithm from having the same-total-cost property with respect to the Tabulation Algorithm.) `IsMemberOfSolution` also collects the set of visited nodes that are already marked “Reachable” and “NotReachable” in sets `ReachableNodes` and `UnreachableNodes`. New code at the beginning of the loop (lines [8]-[9]) tests whether the mark on node (n, d_2) is either “Reachable” or “NotReachable”; if so, the node’s predecessors are not explored.

Before `IsMemberOfSolution` returns, the sets `VisitedNodes`, `VisitedEdges`, `ReachableNodes`, and `UnreachableNodes` are used in a new phase of the algorithm—a forward traversal of $G_{IP}^\#$, during which the marks on visited nodes are changed from “Unknown” to either “Reachable” or “NotReachable” (see lines [40]-[51]). Any node that has a predecessor with the mark “Reachable” is marked “Reachable”; any node for which all predecessors have the mark “NotReachable” is marked “NotReachable”. (Because this phase of the algorithm only traverses edges visited in the earlier phase (lines [1]-[39]), this causes no increase in the asymptotic cost of the algorithm when it is used for answering a single query.)

Finally, the return condition is changed to:

[52] **return**(`ReachableNodes` $\neq \emptyset$)

In this version of the Demand-Tabulation Algorithm, for a request sequence that places demands on all possible nodes of $G_{IP}^\#$, a given edge in $G_{IP}^\#$ is traversed (in `IsMemberOfSolution` proper) during exactly one of the invocations of `IsMemberOfSolution`. Thus, the total amount of work performed (in `IsMemberOfSolution` proper) is proportional to the total number of edges in $G_{IP}^\#$. This is $O(ED^2)$, $O(hED)$, and $O(ED)$, for distributive, h -sparse, and locally separable problems, respectively.

7. Applications to Other Interprocedural Analysis Problems

In this section, we describe adaptations of our techniques that allow them to be used for solving two other kinds of interprocedural analysis problems: interprocedural flow-sensitive side-effect analysis and interprocedural program slicing. In both cases, the algorithms we give are asymptotically faster than ones given previously in the literature. We also describe how to obtain demand algorithms for the two problems, which have no analog in previous papers on these problems.

7.1. Flow-Sensitive Side-Effect Analysis

Callahan gave algorithms for solving two flow-sensitive side-effect problems: must-modify and may-use [6]. 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, each of which represents a subset of the intraprocedural reaching-definitions information, together with interprocedural linkage information.

Although the problems examined by Callahan are not IFDS problems as defined in Definition 2.4, they are closely related to them. The basic difference between IFDS problems and Callahan’s problems is that the former summarize what must be true at a program point in all calling contexts, while the latter 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 has a further difference: it is 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.

When looked at in these terms, Callahan’s two problems can be thought of as specific examples of problems in two general *classes* of problems: one class can be posed as realizable-path problems in an exploded super-graph; the other can be posed as same-level realizable-path problems. As we now show, slight modifications to the algorithms of Sections 4, 5, and 6 can be used to solve the problems in these classes. Furthermore, the algorithms we give can solve distributive and h -sparse problem instances, not just locally separable ones such as must-modify and may-use.

Same-Level Realizable-Path Problems

To solve same-level realizable-path problems, which include (the complement of) the must-modify problem, only two modifications to the Tabulation Algorithm are needed:

- (i) Initialize PathEdge by inserting all edges of the form $((s_p, d), (s_p, d))$ for all $p \in \{0, \dots, k\}$ and $d \in (D_p \cup \{\Lambda\})$.
- (ii) Determine the answer from the set of edges of the form $((s_q, d_1), (e_q, d_2))$ that are in the final value of PathEdge.

Procedure ForwardTabulateSLRPs is unchanged. The modified version of the Tabulation Algorithm, called SolveViaSLRPTabulation, is presented in Figure 14.

Realizable-Path Problems

In the realizable-path problems, which include the may-use problem, we are interested in realizable paths from each node of the form (s_p, d) to some distinguished set $X^\#$ of nodes of $G_{IP}^\#$. For example, in the may-use problem $X^\#$ is the set of all nodes of the form (n, v) where flow-graph node n uses program-variable v .

To solve the realizable-path problems, we first apply a transformation to the exploded super-graph similar to the one described at the end of Section 4.1: We add a new node *dummy* to $G_{IP}^\#$, and add edges from all of the nodes in $X^\#$ to *dummy*. We then make use of a function, called ReachedFromViaRealizablePath, which is identical to function IsMemberOfSolution of Figure 11 except that it initializes PathEdge and SummaryEdge to \emptyset , and it returns a set of exploded super-graph nodes rather than a Boolean:

[60] **return**($\{(n, d_2) \mid ((n, d_2), (x, d_1)) \in \text{RealizablePath}\}$)

Finally, for each procedure p , the solution X_p is obtained as follows:

$X_p := \{d \in D_p \mid (s_p, d) \in \text{ReachedFromViaRealizablePath}(\text{dummy})\}.$

Costs of the Realizable-Path Algorithms

Callahan gave two algorithms—one for the must-modify problem and one for the may-use problem. Surprisingly, in each case, when the *total* cost is considered—the cost of constructing the program summary graph, plus the cost of solving the problem on the program summary graph—Callahan’s algorithms are *less efficient* than the

```

procedure SolveViaSLRPTabulation( $G_{IP}^\#$ )
begin
[1]  ForwardTabulateAllSLRPs( $G_{IP}^\#$ )
[2]  for each  $p \in \{0, \dots, k\}$  do
[3]     $X_p := \{(d_1, d_2) \in ((D_p \cup \{\Lambda\}) \times (D_p \cup \{\Lambda\})) \mid ((s_p, d_1), (e_p, d_2)) \in \text{PathEdge}\}$ 
[4]  od
end

procedure ForwardTabulateAllSLRPs( $G_{IP}^\#$ )
begin
[5]  Let  $(N^\#, E^\#, C^\#) = G_{IP}^\#$ 
[6]  PathEdge :=  $\emptyset$ 
[7]  SummaryEdge :=  $\emptyset$ 
[8]  WorkList :=  $\emptyset$ 
[9]  for each  $p \in \{0, \dots, k\}$  and  $d \in (D_p \cup \{\Lambda\})$  do
[10]   Insert  $((s_p, d), (s_p, d))$  into PathEdge
[11]   Insert  $((s_p, d), (s_p, d))$  into WorkList
[12] od
[13] ForwardTabulateSLRPs(WorkList)
end

```

Figure 14. Procedure SolveViaSLRPTabulation determines the solution to flow-sensitive side-effect analysis problems like must-modify by determining whether certain same-level realizable paths exist in $G_{IP}^\#$.

algorithms given above!⁷ The cost of Callahan’s approach has two parts:

- (i) The cost of doing the dataflow analysis (computing “reaches information”) needed to construct the program summary graph, which in the worst case is $O(D \sum_p Call_p E_p)$.
- (ii) The cost of solving the problem on the program summary graph, which in the worst case is $O(D \sum_p Call_p^2)$.

In contrast, `SolveViaSLRPTabulation` and `ReachedFromViaRealizablePath` have the same asymptotic running time as the Tabulation Algorithm. In particular, for locally separable problems the running time is $O(ED)$.

Table 7.1 compares the costs of three strategies for solving flow-sensitive side-effect problems.⁸

Class of functions	Phase of algorithm	Asymptotic running time		
		Variants of Tabulation (<code>SolveViaSLRPTabulation</code> and <code>ReachedFromViaRealizablePath</code>)	Variants of Compressed-Tabulation (not given in the paper)	Callahan’s Algorithm
Distributive	Compress		$O(D^3 \sum_p Call_p E_p)$	Not applicable
	Solve	$O(Call B^2 D^2 + ED^3)$	$O(Call B^2 D^2 + D^3 \sum_p Call_p^2)$	
h -sparse	Compress		$O(hD^2 \sum_p Call_p E_p)$	Not applicable
	Solve	$O(Call B^2 D^2 + Call D^3 + hED^2)$	$O(Call B^2 D^2 + D^3 \sum_p Call_p^2)$	
Locally separable	Compress		$O(D \sum_p Call_p E_p)$	$O(D \sum_p Call_p E_p)$
	Solve	$O(ED)$	$O(D \sum_p Call_p^2)$	$O(D \sum_p Call_p^2)$

Table 7.1. Comparison of the asymptotic running times of three strategies for solving flow-sensitive side-effect analysis problems.

With minor changes, procedures `FindRealizablePaths` of Figure 7 (modified to call `ForwardTabulateSLRPs` similarly to the way that `IsMemberOfSolution` calls `BackwardTabulateSLRPs`) and `BackwardTabulateSLRPs` of Figure 9 yield algorithms that have no analog in Callahan’s work—*demand* algorithms for, respectively, realizable-path and same-level realizable-path flow-sensitive side-effect problems.

7.2. Interprocedural Program Slicing

The realizable-path reachability problem is also the heart of the problem of interprocedural program slicing. The fastest previously known algorithm for the problem is the one given by Horwitz, Reps, and Binkley; in this section, we assume familiarity with the terminology used in the paper that describes that algorithm [18].

⁷On reflection, this is less surprising: The step of constructing the program summary graph corresponds to the step of compression in the algorithm presented in Section 5. We have already seen in Section 5 how compression leads to algorithms that, in the worst case, are less efficient than algorithms that do not use compression.

⁸From a historical standpoint, it is interesting to consider a fourth strategy for solving these problems: one based on the Horwitz-Reps-Binkley interprocedural-slicing algorithm. By coincidence, the original paper describing the Horwitz-Reps-Binkley slicing algorithm [17], and the paper by Callahan describing how the “program summary graph” could be used to solve flow-sensitive side-effect analysis problems [6] were published back-to-back in the Proceedings of the 1988 Conference on Programming Language Design and Implementation. While it was thought at that time that the techniques presented in the two papers must be related, it was not entirely clear what the relationship was. The framework of the present paper allows us to clarify the relationship between the two papers:

- One of the subroutines of the Horwitz-Reps-Binkley algorithm is an algorithm for the same-level realizable-path reachability problem, and this can now be seen as an algorithm for solving the same-level-valid-path flow-sensitive side-effect problems. The slicing algorithm itself can be used as an algorithm for solving the valid-path flow-sensitive side-effect problems. Unlike Callahan’s algorithm, the algorithms based on the Horwitz-Reps-Binkley algorithm can handle distributive and h -sparse, as well as locally separable problems. However, they are not as efficient as the algorithms described in the present paper. Furthermore, on the locally separable problems they are not as efficient as Callahan’s algorithm.
- Comparing in the other direction, Callahan did not give a fully general algorithm for either the realizable-path reachability problem or the same-level realizable-path reachability problem, and so his techniques do not directly solve the interprocedural-slicing problem.

The methods that have been described in earlier sections of the present paper carry over to the interprocedural-slicing problem by using them not as operations on the exploded super-graph of an IFDS problem, but as operations on the program’s “system dependence graph”, or SDG, (but before the “characteristic-graph edges” have been added to the SDG). Given an SDG (without characteristic-graph edges), interprocedural slicing can be performed by the following method:

- (i) Run procedure ForwardTabulateAllSLRPs of Figure 14.⁹
- (ii) When ForwardTabulateAllSLRPs terminates, SummaryEdge contains all the characteristic-graph edges needed to complete the construction of the SDG. Add these edges to the SDG.
- (iii) Perform the actual slicing operations as usual (*i.e.*, using the backward-slicing algorithm of Figure 9 of reference [18]).

A bound on the cost of the Horwitz-Reps-Binkley algorithm for interprocedural slicing can be expressed in terms of k (the number of procedures in the program), $Call_p$, $Call$, and the following additional parameters:

E_{\max}	the largest number of edges in any procedure’s dependence graph
$Params$	the largest number of formal parameters in any procedure
$Globals$	the number of global variables in the program
X	$Globals + Params$
$Call_{\max}$	$\max_p Call_p$

In [18], it is shown that the cost of the Horwitz-Reps-Binkley algorithm is bounded by $O(Call X E_{\max} + Call Call_{\max}^2 X^4)$. In contrast, it can be shown that when the algorithm outlined above is used for interprocedural slicing, the cost is bounded by $O(k X E_{\max} + Call X^3)$. Note that it is reasonable to assume that k , the number of procedures, is strictly less than $Call$, the total number of call sites. Because there is a family of examples on which the Horwitz-Reps-Binkley algorithm actually performs $\Omega(Call X E_{\max} + Call Call_{\max}^2 X^4)$ steps, the new method is an asymptotically faster algorithm.

As discussed in Section 6, neither the Horwitz-Reps-Binkley backward-slicing algorithm nor the slicing algorithm presented above is a true *demand* algorithm, since they both compute *all* characteristic graph edges before computing a slice. A demand algorithm can be obtained from function ReachedFromViaRealizablePath, described in Section 7.1. When ReachedFromViaRealizablePath is called with an SDG node n as its argument, it returns the set of SDG nodes that are identified by Pass 1 of the Horwitz-Reps-Binkley backward-slicing algorithm (Figure 9 of reference [18]). The full slice set can be obtained as follows:

$$\text{ReachedFromViaRealizablePath}(n) \cup \{ m \mid \exists x \text{ such that } (m, x) \in \text{PathEdge} \}$$

When used for program slicing, the worst-case running time of ReachedFromViaRealizablePath is also bounded by $O(k X E_{\max} + Call X^3)$, so in the worst case its running time is the same as that of the algorithm based on ForwardTabulateAllSLRPs. However, because the modified ReachedFromViaRealizablePath only computes characteristic-graph edges as needed, it is reasonable to expect it to outperform the algorithm based on ForwardTabulateAllSLRPs.

Because ReachedFromViaRealizablePath re-initializes PathEdge and SummaryEdge to \emptyset , the algorithm described above is not a caching demand algorithm for interprocedural slicing. Because the answer slice set is determined from the edges in PathEdge, it is important to re-initialize PathEdge to \emptyset . However, the SummaryEdge set can be reused, and by extending the notion of summary edge to include path edges of the form $((s_p, d_1), (e_p, d_2))$, it is possible to create a caching demand algorithm that eliminates the cost of installing a summary edge that has already been computed during a previous demand.

Algorithms for forward slicing are obtained merely by applying the algorithms described above to the SDG with all edges reversed in direction.

8. Related Work

This paper has defined a natural and quite general framework for interprocedural dataflow analysis, which we have termed the IFDS framework. It has shown how problems in this framework can be solved by finding the solution to a realizable-path reachability problem. Three polynomial-time algorithms for realizable-path reacha-

⁹For the purposes of this section, we assume that in the algorithms of this paper all references to nodes of exploded super-graph $G_{IP}^{\#}$ —that is, pairs of the form (n, d) —are changed to refer to nodes of the SDG.

bility problems have been presented: the Tabulation Algorithm, the Compressed-Tabulation Algorithm, and the Demand-Tabulation Algorithm. The Tabulation and Demand-Tabulation Algorithms are asymptotically faster than the best previously known algorithm for the realizable-path reachability problem.

In the remainder of this section, we summarize how these ideas relate to previous work.

Previous Interprocedural Dataflow-Analysis Frameworks

The IFDS framework is based on earlier interprocedural dataflow-analysis frameworks defined by Sharir and Pnueli [31] and Knoop and Steffen [21]. 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) There are allowed to be dataflow functions on the E^1 edges.

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).

It can be shown that for distributive problems, condition (iii) is a generalization of a previous extension—by Knoop and Steffen—to the Sharir-Pnueli framework in order to permit it to cover programming languages in which recursive procedures have local variables and call-by-value parameters [21]. (Knoop and Steffen observed that the summary function for a return-site node must combine the dataflow information that holds at the corresponding call node with the dataflow information that holds at the exit node of the called procedure. Intuitively, the dataflow information that holds at the return-site node is a combination of the information about local variables that holds at the corresponding call—those variables are not visible to the called procedure and therefore cannot be affected by the call—with the information about global variables that holds at the end of the called procedure.)

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 [31]. 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 [21]. 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 [31] and [21]. 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 algorithms given in Sections 4 and 6.

The Tabulation Algorithm can be thought of as a specialization for IFDS problems of the Sharir-Pnueli iterative algorithm. The main differences introduced in the Tabulation Algorithm are:

- The use of representation relations as a compact way of representing functions.
- “Point-wise” tabulation, rather than tabulation on the basis of set-valued arguments. That is, PathEdge is a subset of $N^\# \times N^\#$; the Sharir-Pnueli iterative algorithm tabulates a subset of $N^* \times 2^D \times 2^D$.
- Summary edges are introduced to avoid redundant work at call sites when path edges of the form $((s_p, d_1), (e_p, d_2))$ are discovered.

When compared with the algorithms of Callahan and Horwitz-Reps-Binkley, the key feature that causes our algorithms to have better worst-case running times is the idea (in “forward algorithms”, such as the Tabulation Algorithm) that the only anchor sites are nodes in $G_{lp}^\#$ of the form (s_p, d) . This idea falls out naturally from the algorithms of Sharir and Pnueli.

Logic-Programming Implementations of Interprocedural Dataflow-Analysis Algorithms

The work described in the present paper was originally motivated by the desire to generalize to a broader class of dataflow-analysis problems the ideas described by Reps in [28]. Reps’s paper shows how locally separable problems in the Sharir-Pnueli framework can be encoded so that they can be solved by evaluating a program with a bottom-up logic-programming system. (With a slight extension, locally separable problems in the Knoop-Steffen framework can be handled, as well.) By making use of the “Magic-sets” transformation [29,2,3], demand dataflow-analysis algorithms are obtained automatically.

In the present paper we have not made use of logic-programming terminology, nor have we stated explicitly how the techniques developed in the present paper generalize the approach taken in Reps’s paper. Suffice it to say that, using the transformation of IFDS problems to realizable-path reachability problems described in Section

3.2, the algorithms from Sections 4, 5, 6, and 7 have implementations as logic programs similar to the algorithms given in [28]. Thus, the work reported in this paper shows how to extend the logic-programming-based approach to implementing dataflow-analysis algorithms to all IFDS problems.

One advantage of the approach adopted in the present paper is that it makes our results more accessible to people who implement in imperative programming languages. All of the algorithms in the paper are given in a style that permits straightforward implementation in an imperative language. In particular, our description of the Demand-Tabulation Algorithm is far simpler than the demand algorithm that results from applying the Magic-sets transformation to a logic program that implements the Tabulation Algorithm.

Dataflow Analysis and Graph-Reachability Problems

Many dataflow-analysis algorithms can be classified as either iterative algorithms [32,20], elimination algorithms [7,13], or reachability algorithms [8,9,6]. Sharir and Pnueli presented two iterative algorithms for the problems in their framework [31]. Our work shows that a large subclass of the problems in the Sharir-Pnueli and Knoop-Steffen frameworks can also be posed as graph-reachability problems.

Other work on solving dataflow-analysis problems by reducing them to reachability problems has been done by Kou [23] and Cooper and Kennedy [8,9]. 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 the intraprocedural live-variable problem [23], although his ideas immediately carry over to all the intraprocedural separable problems.

Cooper and Kennedy showed how certain flow-insensitive interprocedural dataflow-analysis problems could be converted to reachability problems [9,8]. 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.

Demand Dataflow Analysis

With the exception of the paper by Reps discussed above [28], previous work on demand-driven dataflow analysis has dealt only with the *intraprocedural* case [1,35].

The work reported in the present paper complements previous work on the intraprocedural case in the sense that our approach to obtaining algorithms for demand-driven dataflow-analysis problems applies equally well to intraprocedural dataflow analysis. However, in intraprocedural dataflow analysis all paths in the flow graph are (statically) valid paths; for this reason, previous work on demand-driven *intraprocedural* dataflow analysis does not extend well to the *interprocedural* case, where the notion of valid paths is important.

A recent paper by Duesterwald, Gupta, and Soffa discusses a very different approach to (intraprocedural) demand dataflow analysis [10]. For each query of the form “Is fact d in the solution set at node x ?”, a set of dataflow equations is set up on the flow graph (but as if all edges were reversed). The flow functions on the reverse graph are the (approximate) inverses of the original forward functions. (A special function—derived from the query—is used for the reversed flow function of node x .) These equations are then solved using a demand-driven fixed-point finding procedure to obtain a value for the start node. The answer to the query (**true** or **false**) is determined from the value so obtained.

In one way, the Duesterwald-Gupta-Soffa approach is more general than ours because they handle distributive functions on arbitrary domains, while we handle only finite power-set domains. However, Duesterwald, Gupta, and Soffa only briefly discuss how their technique might be extended to handle interprocedural dataflow-analysis problems, and it is not really clear from the extensions they outline for the interprocedural case whether the algorithm obtained will properly account for valid paths.

Although the formal framework used by Duesterwald, Gupta, and Soffa and the one developed in the present paper are very different (Galois connections versus realizable-path reachability problems), there is a natural correspondence between the reversal of representation relations and the approximate inverses used in their work. This connection is beyond the scope of this paper. However, we believe that the Demand-Tabulation Algorithm provides (in graph-reachability terms) the appropriate extension of their work to the interprocedural case in a way that properly takes into account valid paths.

Another point of contrast between their work and ours is that we have given a caching demand algorithm, while their (intraprocedural) method does not appear to permit information to be accumulated over successive queries. The reason is that the equations for a given query are tailored to that particular query and are slightly different from the equations for all other queries. As a consequence, answers and intermediate values previously

computed for other queries cannot be reused.

A final point of contrast has to do with the “early-cut-off” optimization proposed by Duesterwald, Gupta, and Soffa. Their algorithm stops—and returns **false**—whenever a certain distinguished value (“double- \top ”) arises. Similarly, it would be possible to halt the Demand-Tabulation Algorithm and return **true** if the source of an edge in `RealizablePath` or `PathEdge` is ever of the form (m, Λ) , because if this happens there must be a realizable path from (s_{main}, Λ) to query-node (x, d_1) . However, this optimization would preclude reuse of information because it leaves the `PathEdge` and `SummaryEdge` sets in an inconsistent state; reusing the sets could cause the wrong answer to be reported on a subsequent query.

Nevertheless, the modifications to the Demand-Tabulation Algorithm described in Section 6.2 do introduce a form of early cut-off that is compatible with the reuse of information: The algorithm does not explore the predecessors of a node already marked `Reachable` or `NotReachable`. This feature of the algorithm is, in fact, vitally important; it is the feature that enables the Demand-Tabulation Algorithm to have the same-total-cost property with respect to the Tabulation Algorithm.

Constant Propagation

Several different algorithms for interprocedural constant propagation were presented by Callahan, Cooper, Kennedy, and Torczon in [5]. It is interesting to compare the accuracy and costs of their methods with the constant-propagation problems that fit into the IFDS framework.¹⁰

The most powerful method considered in [5], *symbolic interpretation*, does not fit into the IFDS framework, both because the functions are not distributive, and because the domain is infinite. Their next best method is *pass-through constant propagation* enhanced with “return jump functions”[14]. This method is of particular interest because experimental evidence indicates that pass-through constant propagation may be as good as symbolic interpretation in practice [14]. A similar—but slightly more accurate—problem can be expressed as a locally separable IFDS problem. (The IFDS version is also more general because it handles recursion. Because [5,14] were working with Fortran programs, they were able to assume that the programs were non-recursive, and their method for computing return jump functions relied on that fact.) The cost of performing pass-through constant propagation using the method proposed in [5,14]—including the cost of building jump functions—is $O(ED)$. Because the IFDS version is locally separable, the cost of our method is also $O(ED)$.

Another kind of constant propagation, *copy constant propagation* (discussed in Appendix A) falls in between pass-through constant propagation and symbolic interpretation in accuracy. Although copy constant propagation is not locally separable, it is 1-sparse, and parameter B equals 1. Therefore the cost is $O(Call D^3 + ED^2)$. The cost can be specified more precisely by observing that the propagation of one literal is independent of the propagation of all other literals. Thus, the Tabulation Algorithm acts as if it were solving *Literal* independent problems, where *Literal* is the number of distinct literals that appear in the program text. Letting $Vars_{max}$ be the maximum number of variables visible in any procedure, the total cost for copy constant propagation is bounded by $O(Literals (Call Vars_{max}^3 + E Vars_{max}^2))$.

Extensions and Other Contexts

Finally, we note two other contexts in which the ideas developed in this paper should prove useful:

- (i) In elimination-based dataflow techniques it is necessary to have an efficient representation for the compositions of functions [7,13]. Thus, the representation described in Section 3.1 also has applications for representing distributive functions in conjunction with standard elimination algorithms.
- (ii) Recently, Khedker and Dhamdhere [19] have put forward a theory of bidirectional dataflow-analysis problems. In bidirectional problems, information is permitted to flow both forwards and backwards along flow-graph edges. Khedker and Dhamdhere confine their attention to intraprocedural separable bidirectional problems. Although Khedker and Dhamdhere do not use graph-reachability terminology, their worklist algorithm can be considered as such (*i.e.*, their value TOP corresponds to “not reachable”; BOT corresponds to “reachable”).

Although (in our terminology) Khedker and Dhamdhere generate their graph-reachability problem by, in a sense, treating the underlying flow graph as an undirected graph, this is probably an inessential detail. Our algorithms can be applied whenever a correspondence between solutions to bidirectional dataflow-analysis problems and realizable-path reachability problems can be established. Once a realizable-path reachability

¹⁰There is no discussion of aliasing in [5], and in this comparison, we will assume that programs are alias-free.

problem is in hand, the Tabulation, Compressed-Tabulation, and Demand-Tabulation algorithms (or perhaps slight notational variants of them) can be used to solve it. Consequently, we expect that the ideas developed in this paper will permit defining a natural class of interprocedural bidirectional problems that can be solved efficiently using our techniques.

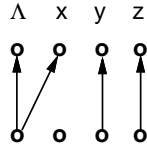
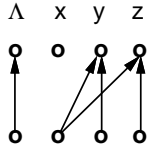
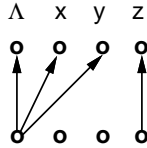
Appendix A: Distributive but Non-Separable Dataflow-Analysis Problems

In this appendix we define several dataflow problems that are \cup -distributive (and h -sparse) but not separable. In each case we give a brief description of the problem, we specify the problem domain D , and the problem direction (forward or backward). We also describe the dataflow functions that would be associated with E^1 edges (edges from a call to the corresponding return-site), the transfer functions T (the functions that map dataflow values across call and return edges), and say what the constants B and h are for each problem instance. For simplicity, our definitions are formulated for call-by-value or call-by-value-result parameter passing. Finally, we give the dataflow functions that would be associated with some example program statements, as well as the graph representations of the functions. In each case, the functions could be associated with either the incoming or the outgoing edges of the nodes that represent the statements; in the former case, the meet-over-all-valid-paths solution for node n would be the information true after n executes, while in the latter case it would be the information true before n executes. (For backward problems, we assume that the edges of the program's flowgraph are reversed, and that the roles of the procedures' start and exit nodes, as well as the roles of the call and return-site nodes, are also reversed. Thus, the outgoing edges of node n for a backward problem are the incoming edges for a forward problem.)

Truly-Live Variables

The *truly-live variables* problem is a variation on the standard *live-variables* problem; every truly-live variable is also live, but not vice versa [12]. Variable x is considered to be truly-live at flowgraph node n if there is a path from n to the end of the program on which x is used before being defined, and the use is either in a predicate, in an output statement, or in an assignment statement that defines a truly-live variable.

For each procedure p , domain D_p is the set of variables visible in p ; the direction is backward. An E^1 edge function is the identity function restricted to the variables that are *not* visible in the called procedure (because those variables' liveness status cannot be affected by the call). The transfer functions for both calls and returns map global variables to themselves, and map actuals to formals and vice versa depending on their parameter-passing modes (for example, for call-by-value parameters, actuals will be mapped to formals but not vice versa). B is the maximum number of times the same actual parameter appears in a single call (for example, for the call $P(x, y, x)$, $B = 2$). Because the indegree of every node in an edge function's representation relation is at most 2, every problem instance is 2-sparse.

statement	output(x)	$x := y + z$	if ($x < y$)
function	$\lambda S. S \cup \{x\}$	$\lambda S. \text{if } x \in S \text{ then}$ $(S - \{x\}) \cup \{y, z\}$ else S	$\lambda S. S \cup \{x, y\}$
graph representation			

Copy Constant Propagation

Copy constant propagation is a simple kind of constant propagation in which no expressions are evaluated. Therefore, a variable is only discovered to be constant if it is assigned a literal value, or it is assigned the value of another variable that is known to be constant [11, pp. 660].

The natural way to specify the problem is to use intersection as the meet operator (the “must-be-constant” problem). To solve the problem using the algorithms described in this paper, it must be transformed into the corresponding \cup -distributive problem: the may-not-be-constant problem. The information given below is for the transformed version of the problem.

For each procedure p , domain D_p is the set of pairs of the form: $\langle x, v \rangle$, for all visible variables x and all literals v that occur in the program. The direction is forward. The E^1 edge functions and the transfer functions are similar to those described above for the truly-live variable problem: information about variables that are not visible to the called procedure is “copied” across the E^1 edge; information about globals is “passed” across call and return edges, as is appropriate information about parameters. Again, B is the maximum number of times the

same actual parameter appears in a single call. Because the indegree of every node in an edge function’s representation relation is at most 1, every problem instance is 1-sparse.

In the functions given below, we use the notation “ $\langle x, * \rangle$ ” to mean all pairs with x as the first component.

statement	$x := 5$	$x := y$	$x := y + 1$
function	$\lambda S. (S \cup \{ \langle x, * \rangle \})$ $\quad - \{ \langle x, 5 \rangle \}$	$\lambda S. (S - \{ \langle x, * \rangle \})$ $\quad \cup \{ \langle x, v \rangle \mid \langle y, v \rangle \in S \}$	$\lambda S. S \cup \{ \langle x, * \rangle \}$
graph representation	$\Lambda \ x, 1 \ x, 5 \ y, 1 \ y, 5$ 	$\Lambda \ x, 1 \ x, 5 \ y, 1 \ y, 5$ 	$\Lambda \ x, 1 \ x, 5 \ y, 1 \ y, 5$

May-Alias Pairs

A pair of expressions $\langle e_1, e_2 \rangle$ (interpreted as r-values) may be aliased at flowgraph node n if there is some path to n that causes e_1 and e_2 to be aliases.

We only know how to express the may-alias-pairs problem in the IFDS framework for a language limited to one-level pointers and procedures with no local variables or parameters.¹¹ In this case, the (single) domain D is the set of pairs of the form $\langle a, b \rangle$, for all pointer variables a and b of the same type, or of the form $\langle a, \&x \rangle$, for all pointer variables a and scalar variables x of the type pointed to by a . (For simplicity, we assume that the pairs are unordered, so if e_1 and e_2 are aliased, this is denoted either by $\langle e_1, e_2 \rangle$ or by $\langle e_2, e_1 \rangle$.) The direction is forward. Every E^1 edge function is the constant function $\lambda S. \emptyset$ (since there are only global variables, all alias pairs can be affected by the call), and every transfer function is the identity function $\lambda S. S$. For every problem instance, $B = 1$ and $h = 1$.

In the functions given below, variables a and b are pointers to some type t , and variable x is a scalar of type t . We use the notation “ $\langle e, * \rangle$ ” to mean all pairs that include e as one component.

statement	$a := b$	$a := \&x$
function	$\lambda S. ((S - \{ \langle a, * \rangle \})$ $\quad \cup \{ \langle a, b \rangle \})$ $\quad \cup \{ \langle a, v \rangle \mid \langle b, v \rangle \in S \text{ and } v \neq a \}$	$\lambda S. ((S - \{ \langle a, * \rangle \})$ $\quad \cup \{ \langle a, \&x \rangle \})$ $\quad \cup \{ \langle a, v \rangle \mid \langle \&x, v \rangle \in S \text{ and } v \neq a \}$
graph representation	$\Lambda \ a, b \ a, \&x \ b, \&x$ 	$\Lambda \ a, b \ a, \&x \ b, \&x$

Must-Alias Pairs

The must-alias-pairs problem is the same as the may-alias-pairs problem except that an alias must hold on *all* paths. The table given below is for the \cup -distributive version of the problem (the may-not-be-aliased problem). The E^1 edge functions, the transfer functions, and the constants B and h are the same as for the may-alias-pairs problem.

¹¹Landi and Ryder [24] gave an algorithm to find the meet-over-all-valid-paths solution to the may-alias-pairs problem in the presence of local variables and parameters; however, their solution involves non-distributive functions, and thus does not fit into the IFDS framework.

statement	$a := b$	$a := \&x$
function	$\lambda S. (S - \{ \langle a, * \rangle \})$ $\cup \{ \langle a, v \rangle \mid \langle b, v \rangle \in S \text{ and } v \neq a \}$	$\lambda S. (S - \{ \langle a, * \rangle \})$ $\cup \{ \langle a, v \rangle \mid \langle \&x, v \rangle \in S \text{ and } v \neq a \}$
graph representation	$\Lambda \quad a, b \quad a, \&x \quad b, \&x$ 	$\Lambda \quad a, b \quad a, \&x \quad b, \&x$

Appendix B: Index of Terms and Notation

$\llbracket R \rrbracket$	Definition 3.2	interprocedurally valid path	Definition 2.3
anchor site	Section 4.1	IP	Definition 2.4
B	Definition 2.4	IsMemberOfSolution	Figure 11
BackwardTabulateSLRPs	Figure 9	IVP	Definition 2.3
C	Definition 2.4	k	Definition 2.1
$C^\#$	Definition 3.8	locally separable problem	Definition 3.12
caching demand algorithm	Section 6	M	Definition 2.4
<i>Call</i>	Definition 2.1	meet-over-all-valid-paths solution	Definition 2.6
call edge	Definition 2.1	MVP_n	Definition 2.6
call node	Definition 2.1	N	Section 4.1
<i>calledProc</i>	Definition 2.1	N_p	Definition 2.1
<i>callers</i>	Definition 2.1	$N^\#$	Definition 3.8
composition, of two relations	Definition 3.4	N^*	Definition 2.1
Compress	Figure 6	node set	Definition 2.1
CompressedEdges	Figure 6	path	Definition 2.2
Compressed-Tabulation Algorithm	Figure 6	path edge	Section 4
corresponding (call and return edges)	Definition 2.1	PathEdge	Figures 4 and 9
demand algorithm	Section 6	path function	Definition 2.5
Demand-Tabulation Algorithm	Figures 9 and 11	$pred^{0,1,2}$	Definition 2.1
D	Definition 2.4	Propagate	Figure 4
D_p	Definition 2.4	R_f	Definition 3.1
E	Section 4.1	ReachedFromViaRealizablePath	Section 7.1
E^0	Definition 2.1	realizable path	Definition 3.9
E_p^0	Definition 2.1	RealizablePath	Figures 7 and 11
E_p^1	Definition 2.1	representation relation	Definition 3.1
e_p	Definition 2.1	<i>Ret</i>	Definition 2.1
E_p	Definition 2.1	return edge	Definition 2.1
$E^\#$	Definition 3.8	return-site node	Definition 2.1
E^*	Definition 2.1	s_p	Definition 2.1
E^0	Definition 2.1	same-level valid path	Definition 2.3
E^1	Definition 2.1	same-level realizable path	Definition 3.9
E^2	Definition 2.1	same-total-cost property	Definition 6.2
edge set	Definition 2.1	SLIVP	Definition 2.3
empty path	Definition 2.2	SolveViaCompression	Figure 6
endpoints, of path	Definition 2.2	SolveViaTabulation	Figure 4
<i>Exit</i>	Definition 2.1	<i>source</i>	Definition 2.1
exit node	Definition 2.1	sparse	Definition 3.11
exploded super-graph	Definition 3.8	<i>Start</i>	Definition 2.1
F	Definition 2.4	start node	Definition 2.1
fg	Definition 2.1	$succ^{0,1,2}$	Definition 2.1
flow graph	Definition 2.1	summary edge	Section 4
ForwardTabulateAllSLRPs	Figure 14	SummaryEdge	Figures 4 and 9
ForwardTabulateSLRPs	Figure 4	super-graph	Definition 2.1
G_p	Definition 2.1	T	Definition 2.4
$G^\#$	Definition 3.8	Tabulation Algorithm	Figure 4
G^*	Definition 2.1	<i>target</i>	Definition 2.1
h -sparse problem	Definition 3.11	<i>Tr</i>	Definition 2.4
IFDS problem	Definition 2.4	valid path	Definition 2.3
interpretation, of a relation	Definition 3.2		
interprocedural, finite, distributive, subset problem	Definition 2.4		

References

1. Babich, W.A. and Jazayeri, M., “The method of attributes for data flow analysis: Part II. Demand analysis,” *Acta Informatica* **10**(3) pp. 265-272 (October 1978).
2. Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., “Magic sets and other strange ways to implement logic programs,” in *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, (1986).
3. Beeri, C. and Ramakrishnan, R., “On the power of magic,” pp. 269-293 in *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, (San Diego, CA, March 1987), (1987).
4. Cai, J. and Paige, R., “Program derivation by fixed point computation,” *Science of Computer Programming* **11** pp. 197-261 (1988/89).
5. 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).
6. 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).
7. Cocke, J., “Global common subexpression elimination,” *ACM SIGPLAN Notices* **5**(7) pp. 20-24 (1970).
8. 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).
9. 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).
10. Duesterwald, E., Gupta, R., and Soffa, M.L., “Demand-driven program analysis,” Technical Report TR-93-15, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA (October 1993).
11. Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).
12. 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).
13. Graham, S.L. and Wegman, M., “A fast and usually linear algorithm for global data flow analysis,” *J. ACM* **23**(1) pp. 172-202 (1976).
14. 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 (1989).
15. Hecht, M.S., *Flow Analysis of Computer Programs*, North-Holland, New York, NY (1977).
16. Horwitz, S. and Teitelbaum, T., “Generating editing environments based on relations and attributes,” *ACM Trans. Program. Lang. Syst.* **8**(4) pp. 577-608 (October 1986).
17. 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).
18. Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).
19. Khedker, U.P. and Dhamdhere, D.M., “A generalized theory of data flow analysis,” *ACM Trans. Program. Lang. Syst.*, (). (To appear.)
20. 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).
21. 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).
22. 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).
23. Kou, L.T., “On live-dead analysis for global data flow problems,” *Journal of the ACM* **24**(3) pp. 473-483 (July 1977).
24. 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).
25. Linton, M.A., “Implementing relational views of programs,” *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. 132-140 (May 1984).
26. Masinter, L.M., “Global program analysis in an interactive environment,” Tech. Rep. SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, CA (January 1980).
27. Reps, T., “Solving demand versions of interprocedural analysis problems,” Unpublished report, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark (October 1993).
28. Reps, T., “Solving demand versions of interprocedural analysis problems,” pp. 389-403 in *Proceedings of the Fifth International Conference on Compiler Construction*, (Edinburgh, Scotland, April 7-9, 1994), *Lecture Notes in Computer Science*, Vol. 786, ed. P. Fritzson, Springer-Verlag, New York, NY (1994).
29. Rohmer, R., Lescoeur, R., and Kersit, J.-M., “The Alexander method, a technique for the processing of recursive axioms in deductive databases,” *New Generation Computing* **4**(3) pp. 273-285 (1986).
30. Rosay, G., *Personal communication*. October 1993.
31. 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).
32. Vyssotsky, V. and Wegner, P., “A graph theoretical Fortran source language analyzer,” Unpublished report, AT&T Bell Laboratories, Murray Hill, NJ (1963). (As cited in Aho, Sethi, and Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading,

MA, 1986.)

33. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).
34. Yellin, D.M., "Speeding up dynamic transitive closure for bounded degree graphs," *Acta Informatica* **30** pp. 369-384 (1993).
35. Zadeck, F.K., "Incremental data flow analysis in a structured program editor," *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, (Montreal, Can., June 20-22, 1984), *ACM SIGPLAN Notices* **19**(6) pp. 132-143 (June 1984).