Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation[1]

Mooly Sagiv,[2] *Thomas Reps, and Susan Horwitz*
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
Electronic mail: {sagiv, reps, horwitz}@cs.wisc.edu

ABSTRACT This paper concerns interprocedural dataflow-analysis problems in which the dataflow information at a program point is represented by an environment (i.e., a mapping from symbols to values), and the effect of a program operation is represented by a distributive environment transformer. We present an efficient dynamic-programming algorithm that produces precise solutions.
The method is applied to solve precisely and efficiently two (decidable) variants of the interprocedural constant-propagation problem: *copy constant propagation* and *linear constant propagation*. The former interprets program statements of the form $x := 7$ and $x := y$. The latter also interprets statements of the form $x := 5 * y + 17$.

1 Introduction

This paper concerns how to find precise solutions to a large class of interprocedural dataflow-analysis problems in polynomial time. Of the problems to which our techniques apply, several variants of the *interprocedural constant-propagation problem* stand out as being of particular importance.

In contrast with *intra*procedural dataflow analysis, where "precise" means "meet-over-all-paths" [Kil73], a precise *inter*procedural dataflow-analysis algorithm must provide the "meet-over-all-*valid*-paths" solution. (A path is *valid* if it respects the fact that when a procedure finishes it returns to the site of the most recent call [SP81, Cal88, LR91, KS92, Rep94, RSH94, RHS95].) In this paper, we show how to find the meet-over-all-valid-paths solution for a certain class of dataflow problems in which the dataflow facts are maps ("environments") from some finite set of symbols D to some (possibly infinite) set of values L (i.e., the dataflow facts are members of $Env(D, L)$), and the dataflow functions ("environment transformers" in $Env(D, L) \stackrel{d}{\rightarrow} Env(D, L)$) distribute over the meet operator of $Env(D, L)$. We call this set of dataflow problems the Interprocedural Distributive Environment problems (or IDE problems, for short).

The contributions of this paper can be summarized as follows:

- We introduce a **compact graph representation of distributive environment transformers.**
- We present a **dynamic-programming algorithm** for finding meet-over-all-valid-paths solutions. For general IDE problems the algorithm will not necessarily terminate. However, we identify a subset of IDE problems for which the algorithm does terminate and runs in time $O(ED^3)$, where E is the number of edges in the program's control-flow graph.
- We study two natural variants of the constant-propagation problem: copy-constant propagation [FL88] and linear-constant propagation, which extends copy constant propagation by interpreting statements of the form $x = a * y + b$, where a and b are

[1]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. Part of this work was done while the authors were visiting the University of Copenhagen.

[2]On leave from IBM Scientific Center, Haifa, Israel.

literals or user-defined constants. The IDE problems that correspond to both of these variants fall into the above-mentioned subset; consequently, our techniques **solve all instances of these constant-propagation problems in time** $O(E\,\text{MaxVisible}^3)$, where "MaxVisible" is the maximum number of variables visible in any procedure of the program. The algorithms obtained in this way improve on the well-known constant-propagation work from Rice [CCKT86, GT93] in two ways:

- The Rice algorithm is not precise for recursive programs. (In fact, it may fall into an infinite loop when applied to recursive programs).
- Because of limitations in the way "return jump functions" are generated, the Rice algorithm does not even yield precise answers for all non-recursive programs.

In contrast, our algorithm yields **precise results, for both recursive and non-recursive programs**.

- Our dataflow-analysis algorithm has been implemented and used to analyze C programs. Preliminary experimental results are reported in Section 6.

The remainder of the paper is organized as follows: In Section 2 we introduce the copy constant-propagation and linear constant-propagation problems. Linear constant propagation is used in subsequent sections to illustrate our ideas. In Section 3 we define the class of IDE problems. In Section 4, we define the compact graph representation of distributive environment transformers and show how to use these graphs to find the meet-over-all-valid-paths solution to a dataflow problem. Section 5 presents our dynamic-programming algorithm. In Section 5.4, we discuss the application of our approach to the copy constant-propagation and linear constant-propagation problems. Preliminary experiments in which our algorithm has been applied to perform linear constant propagation on C programs are reported in Section 6. Section 7 discusses related work. Section 8 gives an overview of how the work has been extended to perform demand-driven dataflow analysis.

2 Distributive Constant-Propagation Problems

There are (at least) two important variants of the constant-propagation problem that fit into the framework presented in this paper: copy constant propagation and linear constant propagation. In copy constant propagation, a variable x is discovered to be constant either if it is assigned a constant value (e.g., $x := 3$) or if it is assigned the value of another variable that is itself constant (e.g., $y := 3$; $x := y$). All other forms of assignment (e.g., $x := y + 1$) are (conservatively) assumed to make x non-constant.

Linear constant propagation identifies a superset of the instances of constant variables found by copy constant propagation. Variable x is discovered to be constant either if it is assigned a constant value (e.g., $x := 3$) or if it is assigned a value that is a linear function of one variable that is itself constant (e.g., $y := 3$; $x := 2 * y + 5$). All other forms of assignment are assumed to make x non-constant.

3 The IDE Framework

3.1 Program Representation

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

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

```
declare x: integer
program main
begin
        call P(7)
        print (x) /* x is a constant here */
end

procedure P (value a : integer)
begin /* a is not a constant here */
        if a > 0 then
           a := a − 2
           call P (a)
           a := a + 2
        fi
        x := −2 ∗ a + 5
        /* x is not a constant here */
end
```

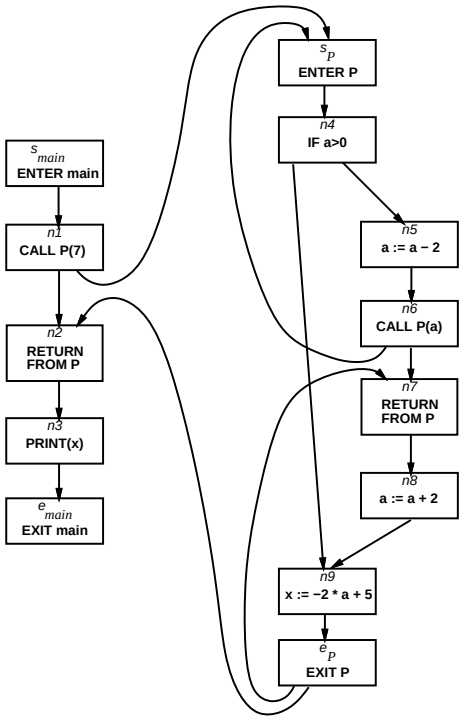

FIGURE 1. An example program and its supergraph G^*.

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

The call-to-return-site edges are included so that we can handle programs with local variables and parameters; the dataflow functions on call-to-return-site and exit-to-return-site edges permit the information about local variables that holds at the call site to be combined with the information about global variables that holds at the end of the called procedure.

Example 3.1 Figure 1 shows an example program and its supergraph. □

3.2 Interprocedural Paths

Definition 3.2 *A* **path** *of length j from node m to node n is a (possibly empty) sequence of j edges, which will be denoted by $[e_1, e_2, \ldots, e_j]$, s.t., for all i, $1 \leq i \leq j-1$, the target of edge e_i is the source of edge e_{i+1}. Path concatenation is denoted by $\|$.* □

The notion of an "(interprocedurally) valid path" captures the idea that some paths in G^* do not respect the fact that when a procedure finishes, control is transfered to the site of the most recent call. A "same-level valid path" is a valid path that starts and ends in the same procedure, and in which every call has a corresponding return.

Definition 3.3 *The sets of* **same-level valid paths** *and* **valid paths** *in G^* are defined inductively as follows:*

- *The empty path is a* **same-level valid path** *(and therefore a* **valid path***).*
- *Path $p \,\|\, e$ is a* **valid path** *if either e is not an exit-to-return-site edge and p is valid or e is an exit-to-return-site edge and $p = p_h \,\|\, e_c \,\|\, p_t$ where p_t is a same-level valid path, p_h is a valid path, and the source node of e_c is the call node that matches the return-site node at the target of e. Such a path is a* **same-level valid path** *if p_h is also a same-level valid path.*

We denote the set of valid paths from node m to node n by $VP(m,n)$. □

3.3 Environments and Environment Transformers

Definition 3.4 *Let D be a finite set of program symbols. Let L be a finite-height meet semi-lattice with a top element $\top$.*[3] *We denote the meet operator by $\sqcap$. The set $Env(D,L)$ of* **environments** *is the set of functions from D to L. The following operations are defined on $Env(D,L)$:*

- *The meet operator on $Env(D,L)$, denoted by $env_1 \sqcap env_2$, is $\lambda d.(env_1(d) \sqcap env_2(d))$.*
- *The top element in $Env(D,L)$, denoted by Ω, is $\lambda d.\top$.*
- *For an environment $env \in Env(D,L)$, $d \in D$, and $l \in L$, the expression $env[d \rightarrow l]$ denotes the environment in which d is mapped to l and any other symbol $d' \neq d$ is mapped according to $env(d')$.*

□

Example 3.5 In the case of integer constant propagation:

- D is the set of integer program variables.
- $L = Z_{\perp}^{\top}$ where $x \sqsubseteq y$ iff $y = \top$, $x = \perp$, or $x = y$. Thus the height of $Z_{\perp}^{\top}$ is 3.

In a constant-propagation problem, $Env(D,L)$ is used as follows: If $env(d) \in Z$ then the variable d has a known constant value in the environment env; the value $\perp$ denotes non-constant and $\top$ denotes an unknown value. □

Definition 3.6 *An environment transformer $t: Env(D,L) \rightarrow Env(D,L)$ is* **distributive** *(denoted by $t: Env(D,L) \stackrel{d}{\rightarrow} Env(D,L)$) iff for every $env_1, env_2, \ldots \in Env(D,L)$, and $d \in D$, $(t(\sqcap_i env_i))(d) = \sqcap_i (t(env_i))(d)$. Note that this equality must also hold for infinite sets of environments.* □

3.4 The Dataflow Functions

A dataflow problem is specified by annotating each edge e of G^* with an environment transformer that captures the effect of the program operation at the source of e.

Definition 3.7 *An* **instance** *of an* **interprocedural distributive environment problem** *(or* **IDE problem** *for short) is a four-tuple, $IP = (G^*, D, L, M)$, where:*

- *G^* is a supergraph.*
- *D and L are as defined in* Definition 3.4.
- *$M: E^* \rightarrow (Env(D,L) \stackrel{d}{\rightarrow} Env(D,L))$ is an assignment of distributive environment transformers to the edges of G^*.*

□

Example 3.8 In the case of linear constant propagation, the interesting environment transformers are those associated with edges whose sources are start nodes, call nodes, exit nodes, or nodes that represent assignment statements.

Whether edges out of start nodes have non-identity environment transformers depends on the semantics of the programming language. For example, these edges' environment transformers may reflect the fact that a procedure's local variables are uninitialized at the start of the procedure; that is, the transformers would be: $\lambda env.env[x_1 \rightarrow \perp][x_2 \rightarrow \perp] \ldots [x_n \rightarrow \perp]$ for all local variables x_i. The environment transformers for the edges out of the start node for the program's main procedure may also reflect the fact that global variables are uninitialized when the program is started. For example, the environment transformer associated with edge $s_{main} \rightarrow n1$ in the supergraph of Figure 1 is: $\lambda env.\lambda d.\perp$.

[3] Hence, L is also complete and has a least element, denoted by $\perp$.

The environment transformers associated with edges out of call and exit nodes reflect the assignments of actual to formal parameters, and vice versa (for call-by-value-result parameters). For example, the transformer associated with edge $n1 \rightarrow s_P$ in the supergraph of Figure 1 is $\lambda env.env[a \rightarrow 7]$. Aliasing (e.g., due to pointers or reference parameters) can be handled conservatively; if x and y might be aliased before the statement $x := 5$, then the corresponding environment transformer would be $\lambda env.env[x \rightarrow 5][y \rightarrow (5 \sqcap env(y))]$.

Linear constant propagation handles assignments of the form $x := c$ and $x := c_1 * y + c_2$ where c, c_1, and c_2 are literals or user-defined constants. The environment transformers associated with these assignment statements are of the form: $\lambda env.env[x \rightarrow c]$, and $\lambda env.env[x \rightarrow c_1 * env(y) + c_2]$, respectively.

For other assignment statements, for example: $x := y + z$, the associated environment transformer is: $\lambda env.env[x \rightarrow \bot]$. This transformer is a safe approximation to the actual semantics of the assignment; the transformer that exactly corresponds to the semantics, $\lambda env.env[x \rightarrow env(y) + env(z)]$, cannot be used in the IDE framework because it is not distributive. □

3.5 The Meet Over All Valid Paths Solution

Definition 3.9 *Let $IP = (G^*, D, L, M)$ be an IDE problem instance. The* **meet-over-all-valid-paths** *solution of IP for a given node $n \in N^*$, denoted by MVP_n, is defined as follows: $MVP_n \stackrel{\text{def}}{=} \sqcap_{q \in VP(s_{main},n)} M(q)(\Omega)$, where M is extended to paths by composition, i.e., $M([\,]) = \lambda env.env$ and $M([e_1, e_2, \ldots, e_j]) \stackrel{\text{def}}{=} M(e_j) \circ M(e_{j-1}) \circ \cdots \circ M(e_2) \circ M(e_1)$.* □

In an IDE problem, the environment transformer associated with an intraprocedural edge e represents a safe approximation to the actual semantics of the code at the source of e. Functions on call-to-return-site edges extract (from the dataflow information valid immediately before the call) dataflow information about local variables that must be re-established after the return from the call. Functions on exit-to-return-site edges extract dataflow information that is both valid at the exit site of the called procedure and relevant to the calling procedure.

Note that call-to-return-site edges introduce some additional paths in the supergraph that do not correspond to standard program-execution paths. The intuition behind the IDE framework is that the interprocedurally valid paths of Definition 3.3 correspond to "paths of action" for particular *subsets* of the runtime entities (e.g., global variables). The path function along a particular path contributes only *part* of the dataflow information that reflects what happens during the corresponding run-time execution. The facts for other subsets of the runtime entities (e.g., local variables) are handled by different "trajectories", for example, paths that take "short-cuts" via call-to-return-site edges.

4 Using Graphs to Represent Environment Transformers

One of the keys to the efficiency of our dataflow-analysis algorithm is the use of a *pointwise* representation of environment transformers. In this section, we show that every distributive environment transformer t: $Env(D, L) \stackrel{d}{\rightarrow} Env(D, L)$ can be represented using a set of functions $F_t = \{f_{d',d} | d', d \in D \cup \{\Lambda\}\}$, each of type $L \rightarrow L$. Function $f_{\Lambda,d}$ is used to represent the effects on symbol d that are independent of the argument environment. Function $f_{d',d}$ captures the effect that the value of symbol d' in the argument environment has on the value of symbol d in the result environment; if d does not depend on d', then $f_{d',d} = \lambda l.\top$. For any symbol d, the value of $t(env)(d)$ can be determined by taking the meet of the values of $D + 1$ individual function applications: $t(env)(d) = f_{\Lambda,d}(\top) \sqcap (\sqcap_{d' \in D} f_{d',d}((env)(d')))$.

It is convenient to represent t as a graph with $2(D+1)$ nodes and at most $(D+1)^2$ edges, where each edge $d' \to d$ is annotated with the function $f_{d',d}$ as described above. (An edge function $\lambda l.\top$ does not contribute to the final value of a symbol; therefore, the edges that would normally be annotated with that function can be omitted from the graph.)

In this section we show that the meet-over-all-valid-paths solution in G^* can be found by finding the "meet-over-all-realizable-paths" solution of a related problem in a graph $G^{\#}_{IP}$ obtained by pasting together the representation graphs for every control flow edge in G^*.

4.1 A Pointwise Representation of Environment Transformers

Definition 4.1 *Let t: $Env(D,L) \xrightarrow{d} Env(D,L)$ be an environment transformer and let $\Lambda \notin D$. The* **pointwise representation** *of t, denoted by R_t: $(D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \to (L \to L)$, is defined by:*

$$R_t(d',d) \stackrel{\text{def}}{=} \begin{cases} id & d' = d = \Lambda \\ \lambda l.t(\Omega)(d) & d' = \Lambda, d \in D \\ \lambda l.\top & d', d \in D \wedge \forall l.t(\Omega[d' \to l])(d) = t(\Omega)(d) \\ id & d', d \in D \wedge \forall l.t(\Omega[d' \to l])(d) = t(\Omega)(d) \sqcap l \\ \lambda l.\begin{cases} \top & l = \top \\ t(\Omega[d' \to l])(d) & o.w. \end{cases} & o.w. \end{cases}$$

Also, for a given representation R_t: $(D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \to (L \to L)$, the **interpretation** *of R_t, $[\![R_t]\!]$: $Env(D,L) \xrightarrow{d} Env(D,L)$ is the distributive environment transformer defined by*

$$[\![R_t]\!](env)(d) \stackrel{\text{def}}{=} R_t(\Lambda,d)(\top) \sqcap (\sqcap_{d' \in D} R_t(d',d)(env(d'))) \tag{1}$$

□

Example 4.2 Figure 2 shows the pointwise representations of the environment transformers for linear constant propagation for the supergraph of Figure 1. □

The intuition behind the definition of R_t is that "macro-function" t is broken down into "micro-functions" that are basically of the form $\lambda l.t(\Omega[d' \to l])(d)$. More precisely, all micro-functions $R_t(d',d)$, where $d' \neq \Lambda$, are co-strict variants of $\lambda l.t(\Omega[d' \to l])(d)$. The micro-functions $R_t(\Lambda,d)$ are the only non-co-strict micro-functions; they play a role similar to the "gen" sets of gen-kill problems. The top function is used whenever possible, i.e., when $\lambda l.t(\Omega[d' \to l])(d)$ is equal to $R_t(\Lambda,d)$ and thus does not contribute to the right-hand side of (1). Finally, the identity function is used whenever possible, i.e., when the right-hand side of (1) will have the same value when id is substituted for $\lambda l.t(\Omega[d' \to l])(d)$.

Theorem 4.3 *For every t: $Env(D,L) \xrightarrow{d} Env(D,L)$, $t = [\![R_t]\!]$.* □

Pointwise representations are closed under composition, as captured by the following definition and theorem.

Definition 4.4 *The* **composition** *$R_{t_1}; R_{t_2}$ of pointwise representations R_{t_1}, R_{t_2}: $(D \cup \{\Lambda\} \times (D \cup \{\Lambda\}) \to (L \to L)$ is defined by: $(R_{t_1}; R_{t_2})(d',d)(l) \stackrel{\text{def}}{=} \sqcap_{z \in D \cup \{\Lambda\}} R_{t_2}(z,d)(R_{t_1}(d',z)(l))$.* □

Theorem 4.5 *For all $t_1, t_2, \ldots, t_n$: $Env(D,L) \to Env(D,L)$,*
$[\![R_{t_1}; R_{t_2}; \cdots; R_{t_n}]\!] = t_n \, \boldsymbol{o} \, t_{n-1} \, \boldsymbol{o} \, \cdots \, \boldsymbol{o} \, t_1$. □

Definition 4.4 means that $R_{t_1}; R_{t_2}$ yields another representation graph. Theorem 4.5 means that the composition of several environment transformers can be represented by a single representation graph. That is, environment transformers are "compressible": there is a bound on the size of the graph needed to represent any such function **as well as the compositions of such functions**!

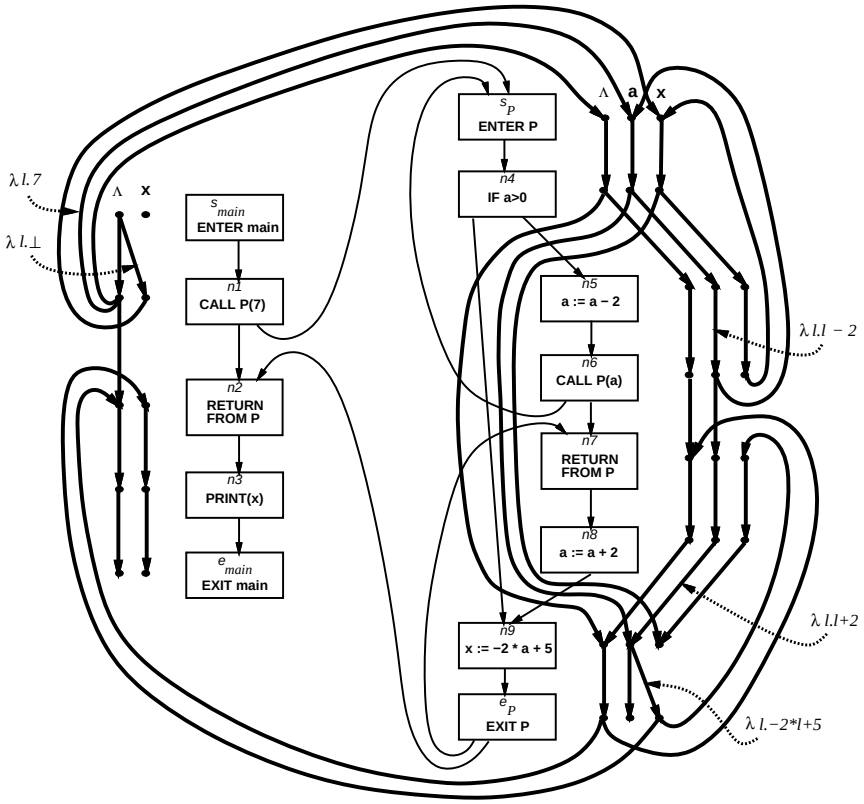

FIGURE 2. The labeled exploded supergraph for the running example program for the linear constant-propagation problem. The edge functions are all $\lambda l.l$ except where indicated.

4.2 The Labeled Exploded Supergraph

Definition 4.6 *Let $IP = (G^*, D, L, M)$ be an IDE problem instance. The* **labeled exploded supergraph** *of IP is a directed graph $G^{\#}_{IP} = (N^{\#}, E^{\#})$ where $N^{\#} \stackrel{\text{def}}{=} N^* \times (D \cup \{\Lambda\})$ and $E^{\#} \stackrel{\text{def}}{=} \{\langle m, d'\rangle \to \langle n, d\rangle \mid m \to n \in E^*, R_{M(m\to n)}(d', d) \neq \lambda l.\top\}$. Edge labels are given by a function $EdgeFn\colon E^{\#} \to (L \to L)$ defined to be: $EdgeFn(\langle m, d'\rangle \to \langle n, d\rangle) \stackrel{\text{def}}{=} R_{M(m\to n)}(d', d)$.*

A path p in $G^{\#}_{IP}$ is a **realizable path** *if the corresponding path in G^* is a valid path. We denote the set of realizable paths from an exploded-graph node em to an exploded-graph node en by $RP(em, en)$.* **Same-level realizable paths** *are defined similarly.* □

Example 4.7 Figure 2 contains the exploded supergraph for the running example program labeled with the non-identity *EdgeFn* functions. □

Definition 4.8 *Let $IP = (G^*, D, L, M)$ be an IDE problem instance. The* **meet-over-all-realizable-paths** *solution of IP for a given exploded node $en \in N^{\#}$, denoted by MRP_{en}, is defined as follows:*

$$MRP_{en} \stackrel{\text{def}}{=} \sqcap_{q \in RP(\langle s_{main}, \Lambda\rangle, en)} PathFn(q)(\top)$$

where $PathFn$ is $EdgeFn$ extended to paths by composition. □

We now state the theorem that is the basis for our algorithm for solving IDE problems:

Theorem 4.9 *For every $n \in N^*$ and $d \in D$, $MVP_n(d) = MRP_{\langle n,d\rangle}$.* □

The consequence of this theorem is that we can solve IDE problems by solving a related problem on the labeled exploded supergraph.

5 A Dynamic Programming Algorithm

In this section, we present an algorithm to compute the meet-over-all-valid-paths solution to a given dataflow problem instance IP. The input to the algorithm is the labeled exploded supergraph $G^{\#}_{IP}$; when the algorithm finishes, for every exploded-graph node $en \in N^{\#}$, $val(en) = MRP_{en}$. The algorithm operates in two phases, which are shown in Figures 3 and 4. In Phase I, the algorithm builds up *path functions* (recorded in *PathFn*) and *summary functions* (recorded in *SummaryFn*). Path functions and summary functions are defined in terms of *edge functions* (*EdgeFn*), and other path functions and summary functions. In Phase II, the path functions are used to determine the actual *values* associated with nodes of the exploded graph.

5.1 Phase I

Phase I is performed by procedure ComputePathFunctions, shown in Figure 3. ComputePathFunctions is a dynamic-programming algorithm that repeatedly computes path functions, which are functions from L to L, for longer and longer paths in $G^{\#}_{IP}$. The path functions to $\langle n, d\rangle$ summarize the effects of same-level realizable paths from the start node of n's procedure p to $\langle n, d\rangle$. There may be a path function from $\langle s_p, d'\rangle$ to $\langle n, d\rangle$ for all $d' \in D \cup \{\Lambda\}$. ComputePathFunctions also computes summary functions, which summarize the effects of same-level realizable paths from nodes of the form $\langle c, d'\rangle$, where c is a call node, to $\langle r, d\rangle$, where r is the corresponding return-site node.

ComputePathFunctions is a worklist algorithm that computes successively better approximations to the path and summary functions. It starts by initializing path and summary functions to $\lambda l.\top$. The worklist is initialized to contain the path from $\langle s_{main}, \Lambda\rangle$ to $\langle s_{main}, \Lambda\rangle$, and the path function for that path is initialized to the identity function, id. On each iteration of the main loop, the algorithm determines better approximations to path and summary functions.

To reduce the amount of work performed, ComputePathFunctions uses an idea similar to the "minimal-function-graph" approach [JM86]: Only after a path function for a path from a node of the form $\langle s_p, d_1\rangle$ to a node of the form $\langle c, d_2\rangle$ has been processed, where c is a call on procedure q, will a path from $\langle s_q, d_3\rangle$ to $\langle s_q, d_3\rangle$ be put on the worklist — and then only if edge $\langle c, d_2\rangle \to \langle s_q, d_3\rangle$ is in $E^{\#}$.

5.2 Phase II

Phase II is performed by procedure ComputeValues, shown in Figure 4. In this phase, the path functions are used to determine the actual *MRP* values associated with nodes of the exploded graph. Phase II consists of two sub-phases:

(i) An iterative algorithm is used to propagate values from the start node of the main procedure to all other start nodes and all call nodes. To compute a new approximation to the value at start node $\langle s_p, d\rangle$, $EdgeFn(en, \langle s_p, d\rangle)$ is applied to the current approximation at all nodes en associated with calls to p. To compute a new approximation to the value at call node $\langle c, d\rangle$ in procedure q, $PathFn(\langle s_q, d'\rangle, \langle c, d\rangle)$ is applied to the current approximations at all nodes $\langle s_q, d'\rangle$.

(ii) Values are computed for all nodes $\langle n, d\rangle$ that are neither start nor call nodes. This is done by applying $PathFn(\langle s_p, d'\rangle, \langle n, d\rangle)$ to $MRP(\langle s_p, d'\rangle)$ for all d' (where p is the procedure that contains n), and taking the meet of the resulting values.

Note that $val(\langle s_{main}, \Lambda\rangle)$ is initialized to $\bot$ in Phase II(i). As a result $\bot$ is propagated to all nodes of the form $\langle n, \Lambda\rangle$. Because the function on an edge from one of these nodes to

```
procedure ComputePathFunctions()
begin
  for all ⟨s_p, d'⟩, ⟨m, d⟩ such that m occurs in procedure p and d', d ∈ D ∪ {Λ} do
    PathFn(⟨s_p, d'⟩, ⟨m, d⟩) = λl.⊤ od
  for all corresponding call-return pairs c, r and d', d ∈ D ∪ {Λ} do
    SummaryFn(⟨c, d'⟩, ⟨r, d⟩) = λl.⊤ od
  WorkList:= {⟨s_main, Λ⟩ → ⟨s_main, Λ⟩}
  PathFn(⟨s_main, Λ⟩ → ⟨s_main, Λ⟩) := id
  while WorkList ≠ ∅ do
    Select and remove an edge ⟨s_p, d1⟩ → ⟨n, d2⟩ from WorkList
    let f = PathFn(⟨s_p, d1⟩ → ⟨n, d2⟩)
    switch(n)
      case n is a call node in p, calling a procedure q:
        for each d3 s.t. ⟨n, d2⟩ → ⟨s_q, d3⟩ ∈ E# do
          Propagate (⟨s_q, d3⟩ → ⟨s_q, d3⟩, id) od
        let r be the return-site node that corresponds to n
        for each d3 s.t. e = ⟨n, d2⟩ → ⟨r, d3⟩ ∈ E# do
          Propagate(⟨s_p, d1⟩ → ⟨r, d3⟩, EdgeFn(e) o f) od
        for each d3 s.t. f3 = SummaryFn(⟨n, d2⟩ → ⟨r, d3⟩) ≠ λl.⊤ do
          Propagate(⟨s_p, d1⟩ → ⟨r, d3⟩, f3 o f) od endcase
      case n is the exit node of p:
        for each call node c that calls p with corresponding return-site node r do
          for each d4, d5 s.t. ⟨c, d4⟩ → ⟨s_p, d1⟩ ∈ E# and ⟨e_p, d2⟩ → ⟨r, d5⟩ ∈ E# do
            let f4 = EdgeFn(⟨c, d4⟩ → ⟨s_p, d1⟩) and
                f5 = EdgeFn(⟨e_p, d2⟩ → ⟨r, d5⟩) and
                f' = (f5 o f o f4)⊓SummaryFn(⟨c, d4⟩ → ⟨r, d5⟩)
            if f' ≠ SummaryFn(⟨c, d4⟩ → ⟨r, d5⟩) then
              SummaryFn(⟨c, d4⟩ → ⟨r, d5⟩) := f'
              let s_q be the start node of c's procedure
              for each d3 s.t. f3 = PathFn(⟨s_q, d3⟩ → ⟨c, d4⟩) ≠ λl.⊤ do
                Propagate(⟨s_q, d3⟩ → ⟨r, d5⟩, f' o f3) od fi od od endcase
      default:
        for each ⟨m, d3⟩ s.t. ⟨n, d2⟩ → ⟨m, d3⟩ ∈ E# do
          Propagate(⟨s_p, d1⟩ → ⟨m, d3⟩, EdgeFn(⟨n, d2⟩ → ⟨m, d3⟩) o f) od endcase
    end switch od
end
procedure Propagate(e, f)
begin
  let f' = f⊓PathFn(e)
  if f' ≠ PathFn(e) then
    PathFn(e) := f'
    Insert e into WorkList fi
end
```

FIGURE 3. The algorithm for Phase I.

```
procedure ComputeValues()
begin
  /* Phase II(i) */
  for each en ∈ N# do val(en) := ⊤ od
  val(⟨s_main, Λ⟩) := ⊥
  WorkList:= {⟨s_main, Λ⟩}
  while WorkList ≠ ∅ do
    Select and remove an exploded-graph node ⟨n, d⟩ from WorkList
    switch(n)
      case n is the start node of p:
        for each c that is a call node inside p do
          for each d' s.t. f' = PathFn(⟨n, d⟩ → ⟨c, d'⟩) ≠ λl.⊤ do
            PropagateValue(⟨c, d'⟩,f'(val(⟨s_p, d⟩))) od od endcase
      case n is a call node in p, calling a procedure q:
        for each d' s.t. ⟨n, d⟩ → ⟨s_q, d'⟩ ∈ E# do
          PropagateValue(⟨s_q, d'⟩, EdgeFn(⟨n, d⟩ → ⟨s_q, d'⟩)(val(⟨n, d⟩))) od endcase
    end switch od
  /* Phase II(ii) */
  for each node n, in a procedure p, that is not a call or a start node do
    for each d', d s.t. f' = PathFn(⟨s_p, d'⟩ → ⟨n, d⟩) ≠ λl.⊤ do
      val(⟨n, d⟩) := val(⟨n, d⟩) ⊓ f'(val(⟨s_p, d'⟩)) od od
end
procedure PropagateValue(en, v)
begin
      let v' = v ⊓ val(en)
      if v' ≠ val(en) then
        val(en) := v'
        Insert en into WorkList fi
end
```

FIGURE 4. The algorithm for Phase II.

a non-Λ node em is always a constant function (see Definition 4.1), the $\bot$ value at $\langle n, \Lambda\rangle$ cannot affect the value at em.

Example 5.1 When applied to the exploded graph of Figure 2, our algorithm discovers that x has the constant value -9 at node $n3$ (the print statement in the main procedure), and that a does *not* have a constant value at node s_P (the start node of procedure P). During Phase I, the algorithm computes the following relevant path and summary functions: $PathFn(\langle s_P, a\rangle \rightarrow \langle n6, a\rangle) = \lambda l.l - 2$, $PathFn(\langle s_P, a\rangle \rightarrow \langle e_P, x\rangle) = \lambda l. -2 * l + 5$, $SummaryFn(\langle n1, \Lambda\rangle \rightarrow \langle n2, x\rangle) = \lambda l. -9$, $PathFn(\langle s_{main}, \Lambda\rangle \rightarrow \langle n2, x\rangle) = \lambda l. -9$, and $PathFn(\langle s_{main}, \Lambda\rangle \rightarrow \langle n3, x\rangle) = \lambda l. -9$.

During Phase II(i), values are propagated as follows to discover that a is not constant at node s_P: $val(\langle s_{main}, \Lambda\rangle) := \bot$, $val(\langle n1, \Lambda\rangle) := \bot$, $val(\langle s_P, a\rangle) := 7$, $val(\langle n6, a\rangle) := 5$, $val(\langle s_P, a\rangle) := 5 \sqcap 7 = \bot$.

During Phase II(ii), $PathFn(\langle s_{main}, \Lambda\rangle \rightarrow \langle n3, x\rangle)$ is applied to $val(\langle s_{main}, \Lambda\rangle)$, producing the value -9. □

5.3 Termination and Cost Issues

The algorithm of the previous section does not terminate for all IDE problems; however, it does terminate for all copy constant-propagation problems, all linear constant-propagation problems, and, in general, for all problems for which the space F of value-transformer functions contains no infinite decreasing chains. (Note that it is possible to construct infinite decreasing chains even in certain distributive variants of constant propagation [SP81, page 206].)

The cost of the algorithm is dominated by the cost of Phase I. This phase can be carried out particularly efficiently if there exists a way of representing the functions such that the functional operations can be computed in unit-time.

These termination and cost issues motivate the following definition:

Definition 5.2 *A class of value-transformer functions $F \subseteq L \to L$ has an* **efficient representation** *if*

- *$id \in F$ and F is closed under functional meet and composition.*
- *F has a finite height (under the pointwise ordering).*
- *There is a representation scheme for F with the following properties:*

  Apply: *Given a representation for a function $f \in F$, for every $l \in L$, $f(l)$ can be computed in constant time.*[4]

  Composition: *Given the representations for any two functions $f_1, f_2 \in F$, a representation for the function $f_1 \circ f_2 \in F$ can be computed in constant time.*

  Meet: *Given the representations for any two functions $f_1, f_2 \in F$, a representation for the function $f_1 \sqcap f_2 \in F$ can be computed in constant time.*

  EQU: *Given the representations for any two functions $f_1, f_2 \in F$, it is possible to test in constant time whether $f_1 = f_2$.*

  Storage: *There is a constant bound on the storage needed for the representation of any function $f \in F$.*

An IDE problem instance $IP = (G^*, D, L, M)$ is **efficiently representable** *if for every $e \in E^*$, and $d', d \in D$, $R_{M(e)}(d', d) \in F$ for some class of functions F that has an efficient representation.* □

Note that in the above definition we do not impose any restrictions on $R_{M(e)}(d', d)$ when either d' or d is Λ. This is based on the assumption that the constant functions and the identity function can always be represented in an efficient manner. (Similarly, we assume that $\lambda l.\top$ can always be represented in an efficient manner.)

In describing the cost of the algorithm it is convenient to introduce the notions of *path edge* and *summary edge*. A path edge is a pair of exploded-graph nodes whose path function is not equal to $\lambda l.\top$; likewise, a summary edge is a pair of exploded-graph nodes whose summary function is not equal to $\lambda l.\top$.

The source of a path edge is a node of the form $\langle s, d \rangle$, where s is the start node of some procedure; thus, there can be at most $D + 1$ path-edge sources in each procedure. Each iteration of Phase I extends a known path edge by composing it with (the function of) either an $E^\#$ edge or a summary edge. There are at most $O(ED^2)$ such edges. Because each edge e can be used in the operation "extend a path along edge e" once for every path-edge source, there are at most $O(ED^3)$ such composition steps.

For each path edge and summary edge from an exploded node $\langle n, \Lambda \rangle$, the path-function value can change at most height of L times. Similarly, path edges and summary edges emanating from other exploded nodes $\langle n, d \rangle$, $d \in D$, can change at most height of F times. Consequently, the total cost of Phase I, and thus of the entire algorithm, is bounded by $O(ED^3)$ (where the constant of proportionality depends on the heights of L and F.)

5.4 Some Efficiently Representable IDE Problems

Copy Constant Propagation

In copy constant propagation, all of the constant functions $\lambda l.c$ are associated with edges of the form $\langle m, \Lambda \rangle \to \langle n, d \rangle$. The only functions on "non-Λ" edges are identity functions. Since

[4] We assume a uniform-cost measure, rather than a logarithmic-cost measure; e.g., operations on integers can be performed in constant time.

$id \boldsymbol{o} id = id$ and $id \sqcap id = id$, the class $F = \{id\}$ is trivially a class of functions that has an efficient representation.

Linear Constant Propagation

Linear constant propagation can be handled using the set of functions $F_{lc} = \{\lambda l.(a*l+b) \sqcap c \mid a \in Z - \{0\}, b \in Z, \text{and } c \in Z_\perp^\top\}$. (The functions where $a = 0$ are the constant functions, and, as in copy constant propagation, these are all associated with "Λ" edges.) Every function $f \in F_{lc}$ can be represented by a triple (a, b, c) where $a \in Z - \{0\}$, $b \in Z$, and $c \in Z_\perp^\top$ where:

$$f = \lambda l. \begin{cases} \top & l = \top \\ (a * l + b) \sqcap c & \text{otherwise} \end{cases}$$

F_{lc} has an efficient representation because:

- $id \in F_{lc}$
- Longest chains in F_{lc} have the form: $\lambda l.(a * l + b) \sqsupset \lambda l.(a * l + b) \sqcap c \sqsupset \lambda l.\perp$, for some $a, b, c \in Z$.
- The four representation requirements are met:

  Apply: Trivial.
  Meet:

$$(a_1, b_1, c_1) \sqcap (a_2, b_2, c_2) = \begin{cases} (a_1, b_1, c_1 \sqcap c_2) & a_1 = a_2, b_1 = b_2 \\ (a_1, b_1, c) & c = (a_1 * l_0 + b_1) \sqcap c_1 \sqcap c_2, \text{ where} \\ & l_0 = (b_1 - b_2)/(a_2 - a_1) \in Z \\ (1, 0, \perp) & \text{o.w.} \end{cases}$$

  Composition: $(a_1, b_1, c_1) \boldsymbol{o} (a_2, b_2, c_2) = ((a_1 a_2), (a_1 b_2 + b_1), ((a_1 c_2 + b_1) \sqcap c_1))$. Here it is assumed that $x * \top = \top * x = x + \top = \top + x = \top$ for $x \in Z$.

  EQU: All representations except that of $\lambda l.\perp$ are unique. Any two triples in which $c = \perp$ represents $\lambda l.\perp$. However, equality can still be tested in unit time.

The third component c is needed so that the meet of two functions can be represented. For example, consider the code fragment **if** $\cdots$ **then** $y := 5 * x - 7$ **else** $y := 3 * x + 1$ **fi**. Variable y is only constant after the **if** when the initial value of x is 4, and in this case y's value is 13. Therefore, the meet of the functions in the then- and else-branches for y in terms of x is represented by $(5, -7, 13)$.

Linear constant propagation can be also performed on real numbers $R_\perp^\top$. In this case, the meet operation is slightly simpler because there no need to test whether $a_2 - a_1$ divides $b_1 - b_2$ evenly — only that $a_2 \neq a_1$ if $b_2 \neq b_1$.

6 Preliminary Experiments

We have carried out a preliminary study to determine the feasibility of the dynamic-programming algorithm and to compare its accuracy and time requirements with those of the naive algorithm that considers *all* paths rather than just the realizable paths. (The latter approach is still safe, but may be less accurate than the algorithm that considers only realizable paths. For example, for the program in Figure 1, variable x would not be identified as a constant at the print statement in procedure *main*.) The two algorithms were implemented in C and used with a front end that analyzes a C program and generates the corresponding exploded supergraph for the integer linear constant-propagation problem. (Pointers were handled conservatively; every assignment through a pointer was considered to kill all variables to which the "&" operator is applied somewhere in the program; all uses through pointers were considered non-constant.) The study used five C programs taken from the SPEC integer benchmark suite [SPE92]. Tests were carried out on a Sun SPARCstation 20 Model 61 with 64 MB of RAM.

Example	Lines of code		Procedures	Calls	N	E	D
	Source	Preprocessed					
compress	1503	657	15	28	1329	1464	77
eqntott	3454	2570	61	211	4015	4266	57
gcc.cpp	7037	4079	71	306	6492	7344	91
li	7741	6054	356	1707	13286	12648	56
sc	8515	7378	151	682	12366	13157	150

TABLE 1. Sizes of the input programs.

Example	Dynamic Programming			Naive Algorithm		
	Time	Constants	Lines/sec.	Time	Constants	Lines/sec.
compress	2.3 + .57	82	524	.91 + .40	50	1147
eqntott	4.53 + 1.59	9	564	2.42 + .14	9	1349
gcc.cpp	14.2 + 6.58	37	339	8.1 + .28	29	840
li	51.69 + 43.23	2	81	9.96 + .34	2	752
sc	47.91 + 43.12	78	94	20.4 + 1.13	72	395

TABLE 2. CPU times and number of constants discovered.

Table 1 gives information about code size (lines of source code and lines of preprocessed source code) and the parameters that characterize the size of the control-flow graphs. Table 2 compares the cost and accuracy of the dynamic-programming algorithm and the naive algorithm. The running times are "user cpu-time" + "system cpu-time" (in seconds) for the algorithms once the exploded supergraph is constructed. The columns labeled "Constants" indicate the number of right-hand-side variable uses that were found to be constant. "Lines/sec." indicates the number of lines of source code processed per second.

7 Related Work

The IDE framework is based on earlier interprocedural dataflow-analysis frameworks defined by Sharir and Pnueli [SP81] and Knoop and Steffen [KS92], as well as the *interprocedural, finite, distributive, subset framework* (or *IFDS framework*, for short) that we proposed earlier [RSH94, RHS95]. The IDE framework is basically the Sharir-Pnueli framework with three modifications:

(i) The dataflow domain is restricted to be a domain of environments.
(ii) The dataflow functions are restricted to be distributive environment transformers.
(iii) The edge from a call node to the corresponding return-site node can have an associated dataflow function.

Conditions (i) and (ii) are restrictions that make the IDE framework less general than the full Sharir-Pnueli framework. Condition (iii), however, generalizes the Sharir-Pnueli framework and permits it to cover programming languages in which recursive procedures have local variables and parameters (which the Sharir-Pnueli framework does not). A different generalization to handle recursive procedures with local variables and parameters was proposed by Knoop and Steffen [KS92].

The IDE framework is a strict generalization of the IFDS framework proposed in [RSH94, RHS95]. In IFDS problems, the set of dataflow facts D is a finite set and the dataflow functions (which are in $2^D \to 2^D$) distribute over the meet operator (either union or intersection, depending on the problem). Some IDE problems can be encoded as IFDS problems; however, an IDE problem in which L is infinite — such as the linear constant-propagation problem — cannot be translated into an IFDS problem. Consequently, this paper strictly extends the class of interprocedural dataflow-analysis problems known to be solvable in polynomial time. However, even when L is finite, the algorithm presented in this paper will

perform much better than the algorithm for IFDS problems for many kinds of problems. For example, in the case of copy constant propagation, in any given problem instance the size of L is no larger than the number of constant literals in the program. The IDE version of copy constant propagation involves environments of size D, where D is the set of program variables; by contrast, the IFDS version involves subsets of $D \times L$. For some C programs that we investigated (of around 1,300 lines), the IFDS version ran out of virtual memory, whereas the IDE version finished in a few seconds.

The algorithm for solving IDE problems yields an efficient polynomial algorithm for determining precise (i.e., meet-over-all-valid-paths) solutions. For both copy constant propagation and linear constant propagation, there are several antecedents. A version of interprocedural copy constant propagation was developed at Rice and has been in use for many years. The algorithm is described in [CCKT86], and studies of how the algorithm performs in practice on Fortran programs were carried out by Grove and Torczon [GT93]. However, the Rice algorithm has two potential drawbacks that our algorithm does not have:

- The Rice algorithm is not precise for recursive programs. (In fact, it may fall into an infinite loop when applied to recursive programs.)
- Because of limitations in the way "return jump functions" are generated, the Rice algorithm does not yield precise answers for all non-recursive programs.

We have also shown in this paper how to solve linear constant-propagation problems, which in general find a superset of the instances of constant variables found by copy constant propagation. Several others have also examined classes of constant-propagation problems more general than copy constant propagation [Kar76, SK91, GT93, MS93].

8 Demand Dataflow Analysis

We have developed and implemented a demand algorithm for the IDE framework. The demand algorithm finds the value of a given symbol $d \in D$ at a given control flow graph node $n \in N^*$. Because of space limitations we confine ourselves to a brief summary of this work.

The demand algorithm is similar to the dynamic-programming algorithm of Section 5; however, in the demand algorithm, path functions are computed during a backwards traversal of $G^{\#}_{IP}$ (i.e., edges are traversed from target to source). The relationship between the demand algorithm and the algorithm of Section 5 is similar to the relationship that holds for IFDS problems between the demand algorithm of [RSH94, HRS95] and the exhaustive algorithm of [RSH94, RHS95].

A different approach to obtaining demand versions of interprocedural dataflow-analysis algorithms has been investigated by Duesterwald, Gupta, and Soffa [DGS95]. In their approach, for each query 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 forward flow functions. These equations are then solved using a demand-driven fixed-point-finding procedure.

Our demand algorithm has the following advantages over the algorithm given by Duesterwald, Gupta, and Soffa:

(1) Their algorithm only applies when L has a *finite number of elements*, whereas we require only that L and F be of *finite height*. For example, linear constant propagation, where L has an infinite number of elements, is outside the class of problems handled by their algorithm.

(2) Instead of computing the value of d at n, their algorithm answers queries of the form "Is the value of d at $n \sqsupseteq l$?" for a given value $l \in L$. In linear constant propagation, there is no way to use queries of this form to find the constant value of a given variable.

(3) When restricted to IFDS problems, the worst-case cost of the Duesterwald-Gupta-Soffa

technique is $O(E\,D\,2^D)$. In contrast, the worst-case cost of our demand algorithm is $O(E\,D^3)$.

Duesterwald, Gupta, and Soffa also give a specialized algorithm that, for copy constant propagation, remedies problems (2) and (3).

References

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 47–56, 1988.

[CCKT86] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *SIGPLAN Symposium on Compiler Construction*, pages 152–161, 1986.

[DGS95] E. Duesterwald, R. Gupta, and M.L. Soffa. Demand-driven computation of interprocedural data flow. In *ACM Symposium on Principles of Programming Languages*, pages 37–48, 1995.

[FL88] C.N. Fischer and R.J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1988.

[GT93] D. Grove and L. Torczon. A study of jump function implementations. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 90–99, 1993.

[HRS95] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. Unpublished manuscript, 1995.

[JM86] N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *ACM Symposium on Principles of Programming Languages*, pages 296–306, 1986.

[Kar76] M. Karr. Affine relationship among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[Kil73] G.A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *International Conference on Compiler Construction*, pages 125–140, 1992.

[LR91] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[MS93] R. Metzger and S. Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, 2, 1993.

[Rep94] T. Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction*, pages 389–403, 1994.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[RSH94] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report TR 94-14, Datalogisk Institut, University of Copenhagen, 1994.

[SK91] B. Steffen and J. Knoop. Finite constants: Characterizations of a new decidable set of constants. *Theoretical Computer Science*, 80(2):303–318, 1991.

[SP81] M. Sharir and A. Pnueli. Two approaches for interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[SPE92] SPEC Component CPU Integer Release 2/1992 (Cint92). Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1992.