Program Analysis via Graph Reachability

Thomas Reps
Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706, USA
reps@cs.wisc.edu
http://www.cs.wisc.edu/~reps/

Abstract

This paper describes how a number of program-analysis problems can be solved by transforming them to graph-reachability problems. Some of the program-analysis problems that are amenable to this treatment include program slicing, certain dataflow-analysis problems, and the problem of approximating the possible "shapes" that heap-allocated structures in a program can take on. Relationships between graph reachability and other approaches to program analysis are described. Some techniques that go beyond pure graph reachability are also discussed.

1. Introduction

The purpose of program analysis is to ascertain information about a program without actually running the program. For example, in classical dataflow analysis of imperative programs, the goal is to associate an appropriate set of "dataflow facts" with each program point (*i.e.*, with each assignment statement, call statement, I/O statement, predicate of a loop or conditional statement, *etc*.). Typically, the dataflow facts associated with a program point p describe some aspect of the execution state that holds when control reaches p, such as available expressions, live variables, reaching definitions, *etc.* Information obtained from program analysis is used in program optimizers, as well as in tools for software engineering and re-engineering.

Program-analysis frameworks abstract on the common characteristics of some class of program-analysis problems. Examples of analysis frameworks range from the gen/kill dataflow-analysis problems described in many compiler textbooks to much more elaborate frameworks [6]. Typically, there is an "analysis engine" that can find solutions to all problems that can be specified within the framework. Analyzers for different program-analysis problems are created by "plugging in" certain details that specify the program-analysis problem of interest (*e.g.*, the dataflow functions associated with the edges of a program's control-flow graph, *etc*.).

For many program-analysis frameworks, an instantiation of the framework for a particular program-analysis problem yields a set of equations. The analysis engine underlying the framework is a mechanism for solving a particular family of equation sets (*e.g.*, using chaotic iteration to find a least or greatest solution). For example, each gen/kill dataflow-analysis problem instance yields a set of equations that are solved over a domain of finite sets, where the variables in the equations correspond to program points and each equation is of the form $val_p = ((\bigcup_{q \in pred(p)} val_q) - kill_p) \cup gen_p$. The values $kill_p$ and gen_p are constants associated with program point p: gen_p represents dataflow facts "created" at p, and $kill_p$ represents dataflow facts "removed" by p.

This paper presents a program-analysis framework based on a somewhat different principle: Analysis problems are posed as graph-reachability problems. As will be discussed below, we express (or convert) program-analysis problems to *context-free-language reachability problems* ("CFL-reachability problems"), which are a generalization of ordinary graph-reachability problems. CFL-reachability is defined in Section 2. Some of the program-analysis problems that are amenable to this treatment include:

- Interprocedural program slicing.
- Interprocedural versions of a large class of dataflow-analysis problems.
- A method for approximating the possible "shapes" that heap-allocated structures can take on.

There are a number of benefits to be gained from expressing a program-analysis problem as a graph-reachability problem:

- We obtain an efficient algorithm for solving the program-analysis problem. In a case where the program-analysis problem is expressed as a single-source ordinary graph-reachability problem, the problem can be solved in time linear in the number of nodes and edges in the graph; in a case where the program-analysis problem is expressed as a CFL-reachability problem, the problem can be solved in time cubic in the number of nodes in the graph.
- The difference in asymptotic running time needed to solve ordinary reachability problems and CFL-reachability problems provides insight into possible trade-offs between accuracy and running time for certain program-analysis problems: Because a CFL-reachability problem can be solved in an approximate fashion by treating it as an ordinary reachability problem, this provides an automatic way to obtain an approximate (but safe) solution, via a method that is asymptotically faster than the method for obtaining the more accurate solution.

Invited paper, 1997 Int. Logic Prog. Symp., (Port Jefferson, NY, Oct. 12-16, 1997)

- In program optimization, most of the gains are obtained from making improvements at a program's "hot spots", such as the innermost loops, which means that dataflow information is really only needed for selected locations in the program. Similarly, software-engineering tools that use dataflow analysis often require information only at a certain set of program points (in response to user queries, for example). This suggests that applications that use dataflow analysis could be made more efficient by using a *demand* dataflow-analysis algorithm, which determines whether a given dataflow fact holds at a given point [1,39,27,7,31,14]. For program-analysis problems that can be expressed as CFL-reachability problems, demand algorithms are typically obtained by solving single-target CFL-reachability problems [14].
- The graph-reachability approach provides insight into the prospects for creating parallel program-analysis algorithms. The connection between program analysis and CFL-reachability has been used to establish a number of results that very likely imply that there are limitations on the ability to create efficient parallel algorithms for interprocedural slicing and interprocedural dataflow analysis [29]. Specifically, it was shown that
  - Interprocedural slicing is log-space complete for $\mathcal{P}$.
  - Interprocedural dataflow analysis is $\mathcal{P}$-hard.
  - Interprocedural dataflow-analysis problems that involve finite sets of dataflow facts (such as the classical "gen/kill" problems) are log-space complete for $\mathcal{P}$.

  The consequence of these results is that, unless $\mathcal{P} = \mathcal{NC}$, there do not exist algorithms for interprocedural slicing and interprocedural dataflow analysis in which (i) the number of processors is bounded by a polynomial in the input size, and (ii) the running time is bounded by a polynomial in the log of the input size.
- The graph-reachability approach offers insight into ways that more powerful machinery can be brought to bear on program-analysis problems [27,31].

The remainder of the paper is organized into five sections, as follows: Section 2 defines CFL-reachability. Section 3 discusses how the graph-reachability approach can be used to tackle interprocedural dataflow analysis, interprocedural program slicing, and shape analysis. Section 4 discusses algorithms for solving CFL-reachability problems. Section 5 concerns demand versions of program-analysis problems. Section 6 describes some techniques that go beyond pure graph reachability.

2. Context-Free-Language Reachability Problems

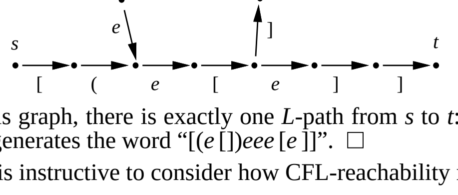
The theme of this paper is that a number of program-analysis problems can be viewed as instances of a more general problem: *CFL-reachability*. A CFL-reachability problem is not an ordinary reachability problem (*e.g.*, transitive closure), but one in which a path is considered to connect two nodes only if the concatenation of the labels on the edges of the path is a word in a particular context-free language:

Definition 2.1. Let L be a context-free language over alphabet Σ, and let G be a graph whose edges are labeled with members of Σ. Each path in G defines a word over Σ, namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in G is an *L-path* if its word is a member of L. We define four varieties of CFL-reachability problems as follows:

(i) The *all-pairs L-path problem* is to determine all pairs of nodes n_1 and n_2 such that there exists an L-path in G from n_1 to n_2.
(ii) The *single-source L-path problem* is to determine all nodes n_2 such that there exists an L-path in G from a given source node n_1 to n_2.
(iii) The *single-target L-path problem* is to determine all nodes n_1 such that there exists an L-path in G from n_1 to a given target node n_2.
(iv) The *single-source/single-target L-path problem* is to determine whether there exists an L-path in G from a given source node n_1 to a given target node n_2. □

Other variants of CFL-reachability include the multi-source L-path problem, the multi-target L-path problem, and the multi-source/multi-target L-path problem.

Example. Consider the graph shown below, and let L be the language that consists of strings of matched parentheses and square brackets, with zero or more e's interspersed:



$$\begin{aligned} L:\ \ matched\ &\rightarrow\ matched\ \ matched \\ &\mid\ (\ matched\ ) \\ &\mid\ [\ matched\ ] \\ &\mid\ e \\ &\mid\ \varepsilon \end{aligned}$$

In this graph, there is exactly one L-path from s to t: The path goes exactly once around the cycle, and generates the word "[(e [])eee [e]]". □

It is instructive to consider how CFL-reachability relates to two more familiar problems:

- An ordinary graph-reachability problem can be treated as a CFL-reachability problem by labeling each edge with the symbol e and letting L be the regular language e^*. For instance, transitive closure is the all-pairs e^*-problem. (Thus, ordinary graph reachability is an example of *regular-*

language reachability—the special case of CFL-reachability in which the language L referred to in Definition 2.1 is a regular language.)

- The *context-free-language recognition problem* (CFL-recognition) answers questions of the form "Given a string ω and a context-free language L, is $\omega \in L$?" The CFL-recognition problem for ω and L can be formulated as the following special kind of single-source/single-target CFL-reachability problem: Create a linear graph $s \rightarrow \cdots \rightarrow t$ that has $|\omega|$ edges, and label the i^{th} edge with the i^{th} letter of ω. There is an L-path from s to t iff $\omega \in L$ [37].

There is a general result that all CFL-reachability problems can be solved in time cubic in the number of nodes in the graph (see Section 4). This method provides the "analysis engine" for our program-analysis framework. Again, it is instructive to consider how the general case relates to the special cases of ordinary reachability and CFL-recognition:

- A single-source ordinary reachability problem can be solved in time linear in the size of the graph (nodes plus edges) using depth-first search.
- Valiant showed that CFL-recognition can be performed in less than cubic time [34]. Unfortunately, the algorithm does not seem to generalize to arbitrary CFL-reachability problems.

From the standpoint of program analysis, the CFL-reachability constraint is a tool that can be employed to filter out paths that are irrelevant to the solution of an analysis problem. In many program-analysis problems, a graph is used as an intermediate representation of a program, but not all paths in the graph represent potential execution paths. Consequently, it is desirable that the analysis results not be polluted (or polluted as little as possible) by the presence of such paths. Although the question of whether a given path in a program representation corresponds to a possible execution path is, in general, undecidable, in many cases certain paths can be identified as being infeasible because they correspond to "execution paths" with mismatched calls and returns.

In the case of interprocedural dataflow analysis, we can characterize a superset of the feasible paths by introducing a context-free language ($L(\mathit{realizable})$, defined below) that mimics the call-return structure of a program's execution: The only paths that can possibly be feasible are those in which "returns" are matched with corresponding "calls". These paths are called *realizable paths*.

Realizable paths are defined in terms of a program's *supergraph* G^*, an example of which is shown in Fig. 1. A supergraph consists of a collection of control-flow graphs, one for each procedure in the program. Each procedure call in the program is represented in G^* 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 control-flow graphs, for each procedure call—represented, say, by call node c and return-site node r—G^* contains three edges: an intraprocedural *call-to-return-site* edge from c to r; an interprocedural *call-to-start* edge from c to the start node of the called procedure; an interprocedural *exit-to-return-site* edge from the exit node of the called procedure to r.

Let each call node in G^* be given a unique index from 1 to *CallSites*, where *CallSites* is the total number of call sites in the program. For each call site c_i, label the call-to-start edge and the exit-to-return-site edge with the symbols "$(_i$" and "$)_i$", respectively. Label all other edges with the symbol e. A path in G^* is a *matched path* iff the path's word is in the language $L(\mathit{matched})$ of balanced-parenthesis strings (interspersed with strings of zero or more e's) generated from nonterminal *realizable* according to the following context-free grammar:

$$
\begin{array}{lcll}
\mathit{matched} & \rightarrow & \mathit{matched}\ \ \mathit{matched} & \\
 & | & (_i\ \ \mathit{matched}\ \ )_i & \text{for } 1 \le i \le \mathit{CallSites} \\
 & | & e & \\
 & | & \varepsilon &
\end{array}
$$

A path is a *realizable path* iff the path's word is in the language $L(\mathit{realizable})$:

$$
\begin{array}{lcll}
\mathit{realizable} & \rightarrow & \mathit{matched}\ \ \mathit{realizable} & \\
 & | & (_i\ \ \mathit{realizable} & \text{for } 1 \le i \le \mathit{CallSites} \\
 & | & \varepsilon &
\end{array}
$$

The language $L(\mathit{realizable})$ is a language of *partially* balanced parentheses: Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse need not hold.

To understand these concepts, it helps to examine a few of the paths that occur in Fig. 1.

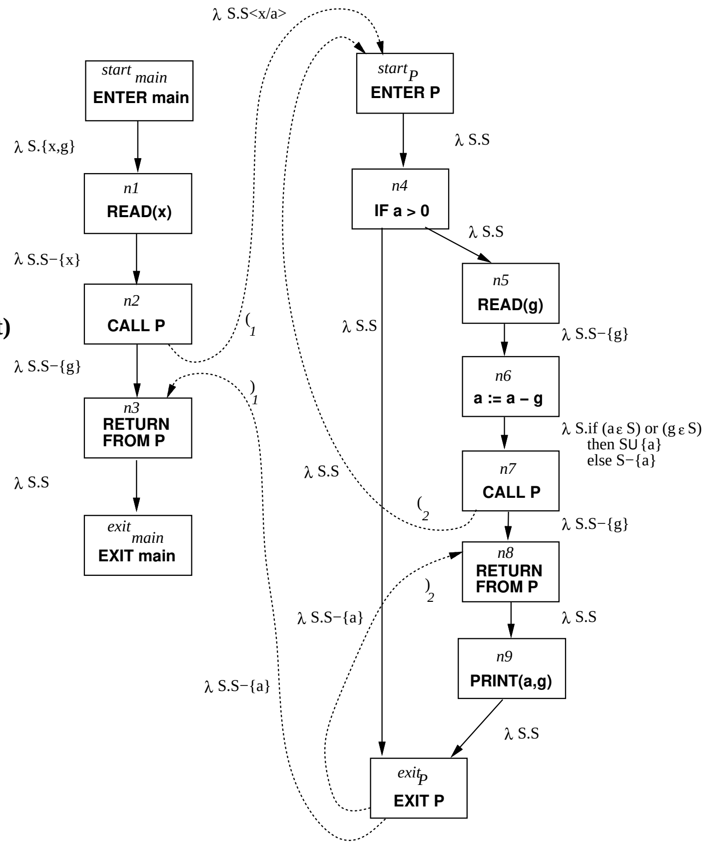
- The path "$\mathit{start}_{main} \rightarrow n1 \rightarrow n2 \rightarrow \mathit{start}_P \rightarrow n4 \rightarrow \mathit{exit}_P \rightarrow n3$", which has word "$ee(_1ee)_1$", is a matched path (and hence a realizable path, as well). In general, a matched path from m to n, where m and n are in the same procedure, represents a sequence 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 path "$\mathit{start}_{main} \rightarrow n1 \rightarrow n2 \rightarrow \mathit{start}_P \rightarrow n4$", which has word "$ee(_1e$", is a realizable path but not a matched path: The call-to-start edge $n2 \rightarrow \mathit{start}_P$ has no matching exit-to-return-site edge. A realizable path from the program's start-node s_{main} to a node n represents a sequence of execution steps that ends, in general, with some number of activation records on the call stack. These correspond to unmatched $(_i$'s in the path's word.
- The path "$\mathit{start}_{main} \rightarrow n1 \rightarrow n2 \rightarrow \mathit{start}_P \rightarrow n4 \rightarrow \mathit{exit}_P \rightarrow n8$", which has word "$ee(_1ee)_2$", is neither a matched path nor a realizable path: The exit-to-return-site edge $\mathit{exit}_P \rightarrow n8$ does not correspond to the preceding call-to-start edge $n2 \rightarrow \mathit{start}_P$. This path represents an infeasible execution path.

```
declare g: int

procedure main
begin
  declare x: int
  read(x)
  call P(x)
end

procedure P(value a : int)
begin
  if (a > 0) then
    read(g)
    a := a − g
    call P(a)
    print(a, g)
  fi
end
```

(a) Example program



(b) Supergraph G^*

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

3. Three Examples

In this section, we show how three program-analysis problems can be transformed into partially balanced parenthesis problems (using languages similar to the language $L(realizable)$ defined in Section 2). Although these examples illustrate the use of only a limited class of context-free languages, the full power of the CFL-reachability framework is also useful in some situations. That is, there are other program-analysis problems that can be solved by expressing them as L-path problems, where L is a context-free language that is something other than a language of partially balanced parentheses [22].

3.1. Interprocedural Dataflow Analysis

Dataflow analysis is concerned with determining an appropriate dataflow value to associate with each point p in a program to summarize (safely) some aspect of the execution state that holds when control reaches p. To define an instance of a dataflow problem, one needs

- The control-flow graph for the program.
- A domain V of dataflow values. Each point in the program is to be associated with some member of V.
- A meet operator $\sqcap$, used for combining information obtained along different paths.
- An assignment of dataflow functions (of type $V \to V$) to the edges of the control-flow graph.

Example. In Fig. 1, the supergraph G^* is annotated with the dataflow functions for the "possibly-uninitialized variables" problem. The possibly-uninitialized variables problem is to determine, for each node n in G^*, a set of program variables that may be uninitialized just before execution reaches n. Thus, V is the power set of the set of program variables. A variable x is possibly uninitialized at n either if there is an x-definition-free path from the start of the program to n, or if there is a path from the start of the program 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 $n6 \to n7$ shown in Fig. 1 adds a to the set of possibly-uninitialized variables after node $n6$ if either

a or g is in the set of possibly-uninitialized variables before node $n6$. □

Below we show how a large class of interprocedural dataflow-analysis problems can be handled by transforming them into realizable-path reachability problems. This is a non-standard treatment of dataflow analysis. Ordinarily, a dataflow-analysis problem is formulated as a *path-function problem*: The path function pf_q for path q is the composition of the functions that label the edges of q; the goal is to determine, for each node n, the "meet-over-*all*-paths" solution: $MOP_n = \bigsqcap_{q \in \text{Paths}(start,\, n)} pf_q(\perp)$, where Paths$(start, n)$ denotes the set of paths in the control-flow graph from the start node to n [16].[1] MOP_n represents a summary of the possible execution states that can arise at n; $\perp \in V$ is a special value that represents the execution state at the beginning of the program; $pf_q(\perp)$ represents the contribution of path q to the summarized state at n.

In *inter*procedural dataflow analysis, the goal shifts from the meet-over-*all-paths* solution to the more precise "meet-over-*all-realizable-paths*" solution: $MRP_n = \bigsqcap_{q \in \text{RPaths}(start_{main},\, n)} pf_q(\perp)$, where RPaths$(start_{main}, n)$ denotes the set of realizable paths from the main procedure's start node to n (and "realizable path" means a path whose word is in the language $L(realizable)$ defined in Section 2) [32,5,19,17,28,7]. Although some realizable paths may also be impossible execution paths, none of the non-realizable paths are possible execution paths. By restricting attention to just the realizable paths from $start_{main}$, we exclude some of the impossible execution paths. In general, therefore, MRP_n characterizes the execution state at n more precisely than MOP_n.

The *interprocedural, finite, distributive, subset problems* (*IFDS problems*) are those interprocedural dataflow-analysis problems that involve a finite set of dataflow facts, and dataflow functions that distribute over the confluence operator (either set union or set intersection, depending on the problem). Thus, an instance of an IFDS problem consists of the following:

- A supergraph G^*.
- A finite set D (the universe of dataflow facts). Each point in the program is to be associated with some member of the domain 2^D.
- An assignment of distributive dataflow functions (of type $2^D \rightarrow 2^D$) to the edges of G^*.

We assume that the meet operator is union; problems in which the meet operator is intersection can always be converted into an equivalent problem in which the meet operator is union.

The IFDS framework can be used for languages with a variety of features (including procedure calls, parameters, global and local variables, and pointers). The call-to-return-site edges are included in G^* so that the IFDS framework can handle programs with local variables and parameters. The dataflow functions on call-to-return-site and exit-to-return-site edges permit the information about local variables and value parameters that holds at the call site to be combined with the information about global variables and reference parameters that holds at the end of the called procedure. The IFDS problems include, but are not limited to, the classical "gen/kill" problems (also known as the "bit-vector" or "separable" problems), *e.g.*, reaching definitions, available expressions, and live variables. In addition, the IFDS problems include many non-gen/kill problems, including possibly-uninitialized variables, truly-live variables [10], and copy-constant propagation [9, pp. 660].

Expressing a problem so that it falls within the IFDS framework may, in some cases, involve a loss of precision. For example, there may be a loss of precision involved in formulating an IFDS version of a problem that must account for aliasing. However, once a problem has been cast as an IFDS problem, it is possible to find the *MRP* solution with no further loss of precision.

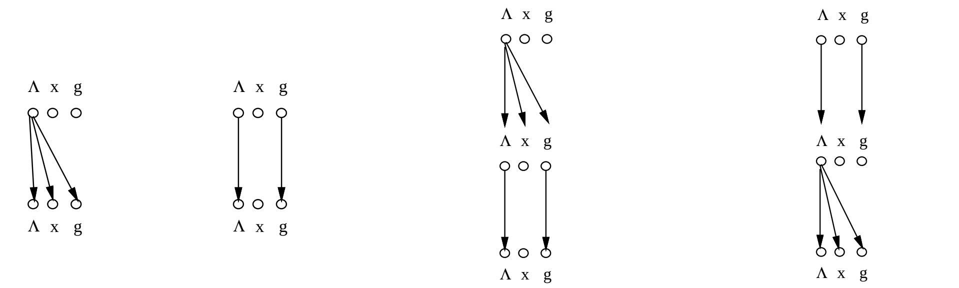
One way to solve an IFDS problem is to convert it to a realizable-path reachability problem [28,14]. For each problem instance, we build an *exploded supergraph* $G^\#$, in which each node $\langle n,d \rangle$ represents dataflow fact $d \in D$ at supergraph node n, and each edge represents a dependence between individual dataflow facts at different supergraph nodes.

The key insight behind this "explosion" is that a distributive function f in $2^D \rightarrow 2^D$ can be represented using a graph with $2D + 2$ nodes; this graph is called f's *representation relation*. Half of the nodes in this graph represent f's input; the other half represent its output. D of these nodes represent the "individual" dataflow facts that form set D, and the remaining node (which we call Λ) essentially represents the empty set. An edge $\Lambda \rightarrow d$ means that d is in $f(S)$ regardless of the value of S (in particular, d is in $f(\emptyset)$). An edge $d_1 \rightarrow d_2$ means that d_2 is not in $f(\emptyset)$, and is in $f(S)$ whenever d_1 is in S. Every graph includes the edge $\Lambda \rightarrow \Lambda$; this is so that function composition corresponds to compositions of representation relations (this is explained below).

Example. The main procedure shown in Fig. 1 has two variables, x and g. Therefore, the representation relations for the dataflow functions associated with this procedure will each have six nodes. The function associated with the edge from $start_{main}$ to $n1$ is $\lambda S.\{x,g\}$; that is, variables x and g are added to the set of possibly-uninitialized variables regardless of the value of S. The representation relation for this function is shown in Fig. 2(a).

---

[1]For some dataflow-analysis problems, such as constant propagation, the meet-over-all-paths solution is uncomputable. A sufficient condition for the solution to be computable is for each edge function f to distribute over the meet operator; that is, for all $a,b \in V$, $f(a \sqcap b) = f(a) \sqcap f(b)$. The problems amenable to the graph-reachability approach are distributive.



(a) $\lambda S.\{x,g\}$ (b) $\lambda S.S - \{x\}$ (c) $\lambda S.S - \{x\} \circ \lambda S.\{x,g\}$ (d) $\lambda S.\{x,g\} \circ \lambda S.S - \{x\}$

Fig. 2. Representation relations for two functions and the two ways of composing the functions.

The representation relation for the function $\lambda S.S - \{x\}$ (which is associated with the edge from *n1* to *n2*) is shown in Fig. 2(b). Note that x is never in the output set, and g is there iff it is in S. □

A function's representation relation captures the function's semantics in the sense that the representation relation can be used to evaluate the function. In particular, the result of applying function f to input S is the union of the values represented by the "output" nodes in f's representation relation that are the targets of edges from the "input" nodes that represent either Λ or a node in S. For example, consider applying the dataflow function $\lambda S.S - \{x\}$ to the set $\{x\}$ using the representation relation shown in Fig. 2(b). There is no edge out of the initial x node, and the only edge out of the initial Λ node is to the final Λ node, so the result of this application is $\emptyset$. The result of applying the same function to the set $\{x,g\}$ is $\{g\}$, because there is an edge from the initial g node to the final g node.

The composition of two functions is represented by "pasting together" the graphs that represent the individual functions. For example, the composition of the two functions discussed above, $\lambda S.S - \{x\} \circ \lambda S.\{x,g\}$, is represented by the graph shown in Fig. 2(c). Paths in a "pasted-together" graph represent the result of applying the composed function. For example, in Fig. 2(c) there is a path from the initial Λ node to the final g node. This means that g is in the final set regardless of the value of S to which the composed function is applied. There is *no* path from an initial node to the final x node; this means that x is not in the final set, regardless of the value of S.

To understand the need for the $\Lambda \to \Lambda$ edges in representation relations, consider the composition of the two example functions in the opposite order, $\lambda S.\{x,g\} \circ \lambda S.S - \{x\}$, which is represented by the graph shown in Fig. 2(d). Note that both x and g are in the final set regardless of the value of S to which the composed functions are applied. In Fig. 2(d), this is reflected by the paths from the initial Λ node to the final x and g nodes. However, if there were no edge from the initial Λ node to the intermediate Λ node, there would be no such paths, and the graph would not correctly represent the composition of the two functions.

Returning to the definition of the exploded supergraph $G^{\#}$: Each node n in supergraph G^* is "exploded" into $D + 1$ nodes in $G^{\#}$, and each edge $m \to n$ in G^* is "exploded" into the representation relation of the function associated with $m \to n$. In particular:

(i) For every node n in G^*, there is a node $\langle n, \Lambda\rangle$ in $G^{\#}$.
(ii) For every node n in G^*, and every dataflow fact $d \in D$, there is a node $\langle n,d\rangle$ in $G^{\#}$.

Given function f associated with edge $m \to n$ of G^*:

(iii) There is an edge in $G^{\#}$ from node $\langle m,\Lambda\rangle$ to node $\langle n,d\rangle$ for every $d \in f(\emptyset)$.
(iv) There is an edge in $G^{\#}$ from node $\langle m,d_1\rangle$ to node $\langle n,d_2\rangle$ for every d_1, d_2 such that $d_2 \in f(\{d_1\})$ and $d_2 \notin f(\emptyset)$.
(v) There is an edge in $G^{\#}$ from node $\langle m,\Lambda\rangle$ to node $\langle n,\Lambda\rangle$.

Because "pasted together" representation relations correspond to function composition, a path in the exploded supergraph from node $\langle m,d_1\rangle$ to node $\langle n,d_2\rangle$ means that if dataflow fact d_1 holds at supergraph node m, then dataflow fact d_2 holds at node n. By looking at paths that start from node $\langle start_{main},\Lambda\rangle$ (which represents the fact that no dataflow facts hold at the start of procedure *main*) we can determine which dataflow facts hold at each node. However, we are not interested in *all* paths in $G^{\#}$, only those that correspond to *realizable* paths in G^*; these are exactly the realizable paths in $G^{\#}$. (For a proof that a dataflow fact d is in MRP_n iff there is a realizable path in $G^{\#}$ from node $\langle start_{main},\Lambda\rangle$ to node $\langle n,d\rangle$, see [25].)

Example. The exploded supergraph that corresponds to the instance of the "possibly-uninitialized variables" problem shown in Fig. 1 is shown in Fig. 3. The dataflow functions are replaced by their representation relations. In Fig. 3, closed circles represent nodes that are reachable along realizable paths from $\langle start_{main},\Lambda\rangle$. Open circles represent nodes not reachable along realizable paths. (For example, note that nodes $\langle n8,g\rangle$ and $\langle n9,g\rangle$ are reachable only along non-realizable paths from $\langle start_{main},\Lambda\rangle$.) This information indicates the nodes' values in the meet-over-all-realizable-paths solution to the dataflow-analysis problem. For instance, the meet-over-all-realizable-paths solution at node $exit_P$ is the set $\{g\}$. (That is, variable g is the only possibly-

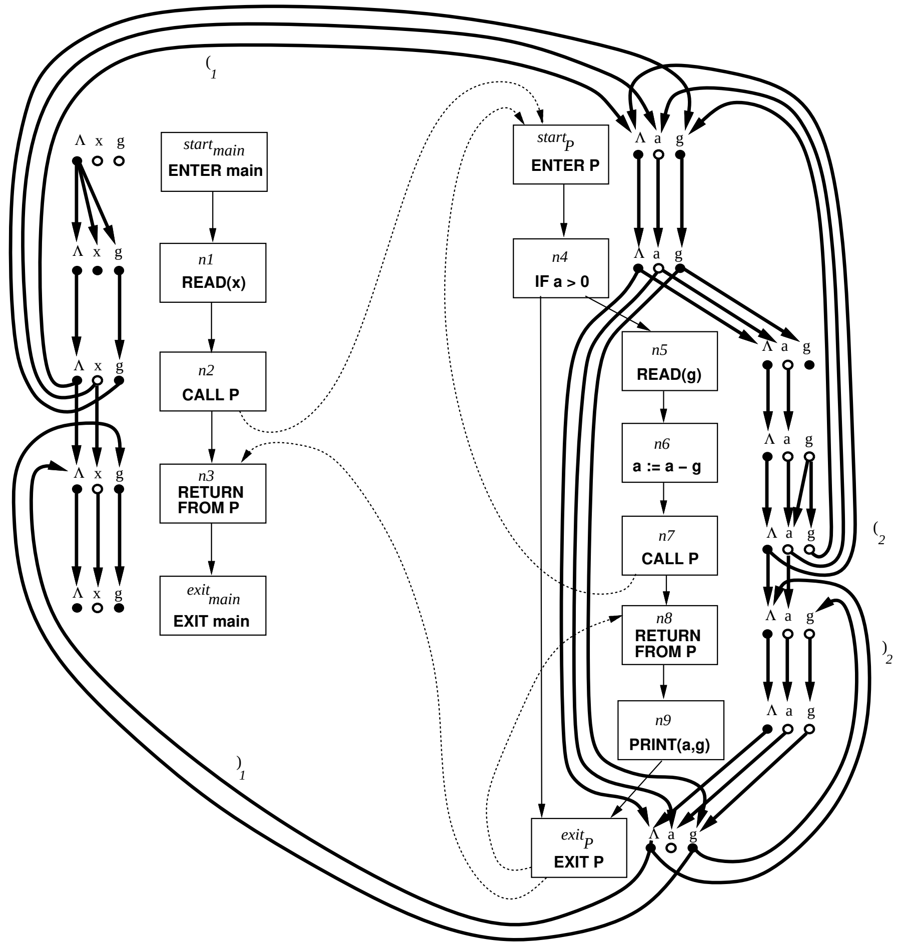


Fig. 3. The exploded supergraph that corresponds to the instance of the possibly-uninitialized variables problem shown in Fig. 1. Closed circles represent nodes of $G^{\#}$ that are reachable along realizable paths from $\langle start_{main}, \Lambda \rangle$. Open circles represent nodes not reachable along such paths.

uninitialized variable just before execution reaches the exit node of procedure P.) In Fig. 3, this information can be obtained by determining that there is a realizable path from $\langle start_{main}, \Lambda \rangle$ to $\langle exit_P, g \rangle$, but not from $\langle start_{main}, \Lambda \rangle$ to $\langle exit_P, a \rangle$. $\square$

3.2. Interprocedural Program Slicing

Slicing is an operation that identifies semantically meaningful decompositions of programs, where the decompositions consist of elements that are not necessarily textually contiguous [36,24,8,12,26,33]. Slicing, and subsequent manipulation of slices, has applications in many software-engineering tools, including tools for program understanding, maintenance, debugging, testing, differencing, specialization, reuse, and merging. (See [33] for references to the literature.)

There are two kinds of slices: a *backward slice* of a program with respect to a set of program elements S is the set of all program elements that might affect (either directly or transitively) the values of the variables used at members of S; a *forward slice* with respect to S is the set of all program elements that might be affected by the computations performed at members of S. A program and one of its backward slices is shown in Fig. 4.

The value of a variable x defined at p is directly affected by the values of the variables used at p and by the predicates that control how many times p is executed; the value of a variable y used at p is directly affected by assignments to y that reach p and by the predicates that control how many times p is executed. Consequently, a slice can be obtained by following chains of dependences in the directly-affects relation. This observation is due to Ottenstein and Ottenstein [24], who noted that *program dependence graphs* (PDGs), which were originally devised for use in parallelizing and vectorizing compilers, are a convenient data structure for slicing. The PDG for a program is a directed graph whose nodes are connected by several kinds of edges. The nodes in the PDG represent the individual statements and predicates of the program. The edges of a PDG represent

```
int add(a, b)
int a, b;
{
  return(a + b);
}

void main()
{
  int sum, i;
  sum = 0;
  i = 1;
  while (i < 11) {
    sum = add(sum,i);
    i = add(i,1);
  }
  printf("sum=%d\n",sum);
  printf("i=%d\n",i);
}
```

```
int add(a, b)
int a, b;
{
  return(a + b);
}

void main()
{
  int [ ] i;
  [ ]
  i = 1;
  while (i < 11) {
    [ ]
    i = add(i,1);
  }
  [ ]
  printf("i=%d\n",i);
}
```

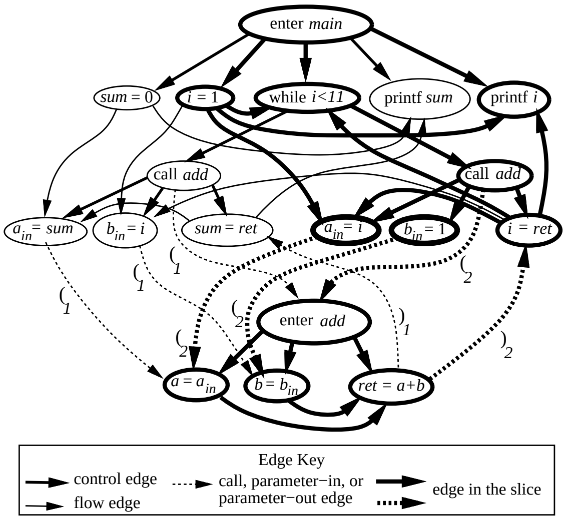


Fig. 4. A program, the slice of the program with respect to the statement *printf("i = %d\n", i)*, and the program's system dependence graph. In the slice, the starting point for the slice is shown in italics, and the empty boxes indicate where program elements have been removed from the original program. In the dependence graph, the edges shown in boldface are the edges in the slice.

the control and flow dependences among the procedure's statements and predicates [18,24,8]. Once a program is represented by its PDG, slices can be obtained in time linear in the size of the PDG by solving an ordinary reachability problem on the PDG. For example, to compute the backward slice with respect to PDG node v, find all PDG nodes from which there is a path to v along control and/or flow edges.

The problem of *inter*procedural slicing concerns how to determine a slice of an entire program, where the slice crosses the boundaries of procedure calls. For this purpose, it is convenient to use *system dependence graphs* (SDGs), which are a variant of PDGs extended to handle multiple procedures [12]. An SDG consists of a collection of procedure dependence graphs (which we will refer to as PDGs)—one for each procedure, including the main procedure. In addition to nodes that represent the assignment statements, I/O statements, and predicates of a procedure, each call statement is represented in the procedure's PDG by a call node and by a collection of actual-in and actual-out nodes: There is an actual-in node for each actual parameter; there is an actual-out node for the return value (if any) and for each value-result parameter that might be modified during the call. Similarly, procedure entry is represented by an entry node and a collection of formal-in and formal-out nodes. (Global variables are treated as "extra" value-result parameters, and thus give rise to additional actual-in, actual-out, formal-in, and formal-out nodes.) The edges of a PDG represent the control and flow dependences in the usual way. The PDGs are connected together to form the SDG by *call* edges (which represent procedure calls, and run from a call node to an entry node) and by *parameter-in* and *parameter-out* edges (which represent parameter passing, and which run from an actual-in node to the corresponding formal-in node, and from a formal-out node to all corresponding actual-out nodes, respectively). In Fig. 4, the graph shown on the right is the SDG for the program that appears on the left.

One algorithm for interprocedural slicing was presented in Weiser's original paper on slicing [36]. This algorithm is equivalent to solving an ordinary reachability problem on the SDG. However, Weiser's algorithm is imprecise in the sense that it may report effects that are transmitted through paths that have mismatched calls and returns (and hence do not represent feasible execution paths). The slices obtained in this way may include unwanted components. For example, there is a path in the SDG shown in Fig. 4 from the node of procedure *main* labeled "$sum = 0$" to the node of *main* labeled "printf *i*." However, this path corresponds to an "execution" in which procedure *add* is called from the first call site in *main*, but returns to the second call site in *main*. This could never happen, and so the node labeled "$sum = 0$" should not be included in the slice with respect to the node labeled "printf *i*".

Although it is undecidable whether a path in the SDG actually corresponds to a possible execution path, we can again use a language of partially balanced parentheses to exclude from consideration paths in which calls and returns are mismatched. The parentheses are defined as follows: Let each call node in SDG G be given a unique index from 1 to *CallSites*, where *CallSites* is the total number of call sites in the program. For each call site c_i, label the outgoing parameter-in edges and the incoming parameter-out edges with the symbols "$(_i$" and "$)_i$", respectively; label the outgoing call edge with "$(_i$". Label all other edges in G with the symbol e. (See Fig. 4.)

Slicing is slightly different from the CFL-reachability problems defined in Definition 2.1. For instance, a backward slice with respect to a given target node t consists of the set of nodes that *lie* *on* a realizable path from the entry node of *main* to t (*cf.* Definition 2.1). However, as long as t is located within a procedure that is transitively callable from *main*, we can change this problem into a single-target CFL-reachability problem (in the sense of Definition 2.1(iii)). We say that a path in an SDG is a *slice path* iff the path's word is in the language $L(slice)$:

$$\begin{array}{lcll} \mathit{unbalanced\text{-}right} & \rightarrow & \mathit{unbalanced\text{-}right}\ \ \mathit{matched} & \\ & | & \mathit{unbalanced\text{-}right}\ \ )_i & \text{for } 1 \le i \le \mathit{CallSites} \\ & | & \varepsilon & \\ \mathit{slice} & \rightarrow & \mathit{unbalanced\text{-}right}\ \ \mathit{realizable} & \end{array}$$

The nodes in the backward slice with respect to t are all nodes n such that there exists an $L(slice)$-path between n and t. That is, the nodes in the backward slice are the solution to the single-target $L(slice)$-path problem for target node t.

To see this, suppose that $r \| s$ is an $L(slice)$-path that connects n and t, where r is an $L(\mathit{unbalanced\text{-}right})$-path and s is an $L(realizable)$-path. As long as t is located within a procedure that is transitively callable from *main*, there exists a path $p \| q$ (of control and call edges) that connects the entry node of *main* to n, where p is an $L(realizable)$-path and q "balances" r; that is, the path $q \| r$ is an $L(matched)$-path. It can be shown that the path $p \| q \| r \| s$ is an $L(realizable)$-path.

3.3. Shape Analysis

Shape analysis is concerned with finding approximations to the possible "shapes" that heap-allocated structures in a program can take on [30,15,23]. This section addresses shape analysis for imperative languages that support non-destructive manipulation of heap-allocated objects. Similar techniques apply to shape analysis for pure functional languages.

We assume we are working with an imperative language that has assignment statements, conditional statements, loops, I/O statements, goto statements, and procedure calls; the parameter-passing mechanism is either by value or value-result; recursion (direct and indirect) is permitted; the language provides atomic data (*e.g.*, integer, real, boolean, identifiers, *etc.*) and Lisp-like constructor and selector operations (nil, cons, car, and cdr), together with appropriate predicates (equal, atom, and null), but not rplaca and rplacd operations. Because of the latter restriction, circular structures cannot be created; however, dag structures (as well as trees) can be created. We assume that a read statement reads just an atom and not an entire tree or dag. For convenience, we also assume that only one constructor or selector is performed per statement (*e.g.*, "y := cons(car(x), y)" must be broken into two statements: "*temp* := car(x); y := cons(*temp*, y)"). (The latter assumption is not essential, but simplifies the presentation.)

Example. An example program is shown in Fig. 5. The program first reads atoms and forms a list x; it then traverses x to assign y the reversal of x. This example will be used throughout the remainder of this section to illustrate our techniques. □

x := nil
read(z)
while $z \neq 0$ **do**
  x := cons(z, x)
  read(z)
od
y := nil
while $x \neq$ nil **do**
  temp := car(x)
  y := cons(*temp*, y)
  x := cdr(x)
od

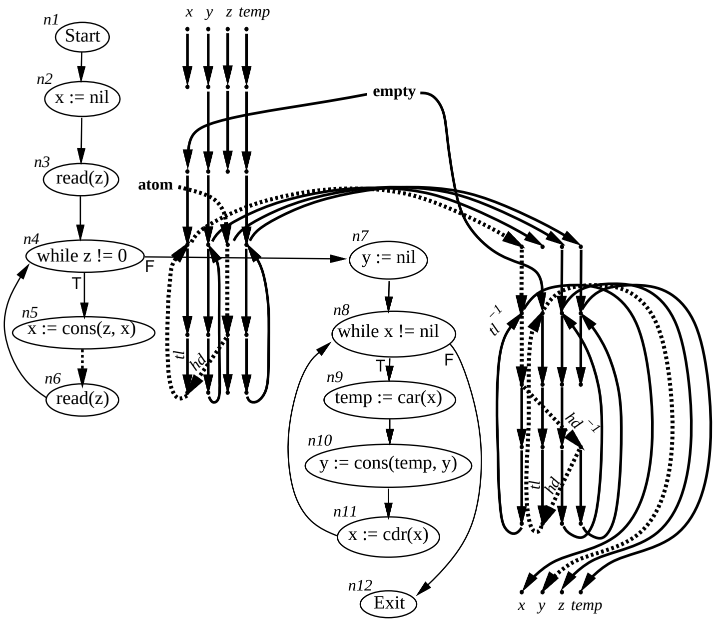


Fig. 5. A program, its control-flow graph, and its equation dependence graph. All edges of the equation dependence graph shown without labels have the label ***id***. The path shown by the dotted lines is a *hd_path* from **atom** to node $\langle n12, y\rangle$.

A collection of dataflow equations can be used to capture an approximation to the shapes of a superset of the terms that can arise at the various points in the program [30,15]. The domain Shape of shape descriptors is the set of selector sequences terminated by **at** or **nil**: Shape $= 2^{L((\mathbf{hd}+\mathbf{tl})^*(\mathbf{at}+\mathbf{nil}))}$. Each sequence in $L((\mathbf{hd}+\mathbf{tl})^*(\mathbf{at}+\mathbf{nil}))$ represents a possible root-to-leaf path. Note that a single shape descriptor in Shape may contain both the selector sequences **hd**.**tl**.**at** and **hd**.**tl**.**hd**.**at**, even though the two paths cannot occur together in a single term.

Dataflow variables correspond to ⟨*program-point*,*program-variable*⟩ pairs. For example, if x is a program variable and p is a point in the program, then $v_{\langle p,x \rangle}$ is a dataflow variable. The dataflow equations are associated with the control-flow graph's edges; there are *several* dataflow equations associated with each edge, one per program variable. The equations on an edge $p \rightarrow q$ reflect the execution actions performed at node p. Thus, the value of a dataflow variable $v_{\langle q,x \rangle}$ approximates the shape of x just *before* q executes. The dataflow-equation schemas are shown in Fig. 6.

Procedure calls with value parameters are handled by introducing equations between dataflow variables associated with actual parameters and dataflow variables associated with formal parameters to reflect the binding changes that occur when a procedure is called. (By introducing equations between dataflow variables associated with formal out-parameters and dataflow variables associated with the corresponding actuals at the return site, call-by-value-result can also be handled.)

When solved over a suitable domain, the equations define an abstract interpretation of the program. The question, however, is: "Over what domain are they to be solved?" One approach is to let the value of each dataflow variable be a *set* of shapes (*i.e.*, a set of sets of root-to-leaf paths) and the join operation be union [30,15]. Functions **cons**, **car**, and **cdr** are appropriate functions from shape sets to shape sets. For example, **cons** is defined as:

$$\mathbf{cons} =_{df} \lambda S_1.\lambda S_2.\{\, \{\, \mathbf{hd}.p_1 \mid p_1 \in s_1 \,\} \cup \{\, \mathbf{tl}.p_2 \mid p_2 \in s_2 \,\} \mid s_1 \in S_1, s_2 \in S_2 \,\}.$$

In our work, however, we use an alternative approach: The value of each dataflow variable is a *single* Shape (*i.e.*, a single set of root-to-leaf paths), and the join operation is union [23]. Functions **cons**, **car**, and **cdr** are functions from Shape to Shape. For example, **cons** is defined as:

$$\mathbf{cons} =_{df} \lambda S_1.\lambda S_2.\{\, \mathbf{hd}.p_1 \mid p_1 \in S_1 \,\} \cup \{\, \mathbf{tl}.p_2 \mid p_2 \in S_2 \,\}.$$

With both approaches, solutions to shape-analysis equations are, in general, infinite. Thus, in practice, there must be a way to report the "shape information" that characterizes the possible values of a program variable at a given program point *indirectly*—*i.e.*, in terms of the values of other program variables at other program points. This indirect information can be viewed as a *simplified set of equations* [30], or, equivalently, as a *regular-tree grammar* [15,23].

The use of domain Shape in place of 2^{Shape} does involve some loss of precision. A feeling for the kind of information that is lost can be obtained by considering the following program fragment:

if $\cdots$ **then** p: $A :=$ cons(B, C)
else q: $A :=$ cons(D, E)
fi
r: $\cdots$

The information available about the value of A at program point r in the two approaches can be represented with the following two tree grammars:

(i) $v_{\langle r,A \rangle} \rightarrow \mathbf{cons}(v_{\langle p,B \rangle}, v_{\langle p,C \rangle}) \mid \mathbf{cons}(v_{\langle q,D \rangle}, v_{\langle q,E \rangle})$ (ii) $v_{\langle r,A \rangle} \rightarrow \mathbf{cons}(v_{\langle p,B \rangle} \mid v_{\langle q,D \rangle}, v_{\langle p,C \rangle} \mid v_{\langle q,E \rangle})$

Grammar (i) uses multiple **cons** right-hand sides for a given nonterminal [15]. In grammar (ii), the link between branches in different **cons** alternatives is broken, and a single **cons** right-hand side is formed with a collection of alternative nonterminals in each arm [23]. The shape descriptions are sharper with grammars of type (i): With grammar (i), nonterminals $v_{\langle p,B \rangle}$ and $v_{\langle q,E \rangle}$ can never occur simultaneously as children of $v_{\langle r,A \rangle}$, whereas grammar (ii) associates nonterminal $v_{\langle r,A \rangle}$ with trees of the form $\mathbf{cons}(v_{\langle p,B \rangle}, v_{\langle q,E \rangle})$.

We now show how shape-analysis information can be obtained by solving CFL-reachability problems on a graph obtained from the program's dataflow equations.

Definition 3.1. Let Eqn_G be the set of equations for the shape-analysis problem on control-flow-graph G. The associated *equation dependence graph* has two special nodes **atom** and **empty**,

Form of source-node p	Equations associated with edge $p \rightarrow q$	
$x := a$, where a is an atom	$v_{\langle q,x \rangle} = \{\, \mathbf{at} \,\}$	$v_{\langle q,z \rangle} = v_{\langle p,z \rangle}$, for all $z \neq x$
read(x)	$v_{\langle q,x \rangle} = \{\, \mathbf{at} \,\}$	"
$x :=$ nil	$v_{\langle q,x \rangle} = \{\, \mathbf{nil} \,\}$	"
$x := y$	$v_{\langle q,x \rangle} = v_{\langle p,y \rangle}$	"
$x :=$ car(y)	$v_{\langle q,x \rangle} = \mathbf{car}(v_{\langle p,y \rangle})$	"
$x :=$ cdr(y)	$v_{\langle q,x \rangle} = \mathbf{cdr}(v_{\langle p,y \rangle})$	"
$x :=$ cons(y, z)	$v_{\langle q,x \rangle} = \mathbf{cons}(v_{\langle p,y \rangle}, v_{\langle p,z \rangle})$	"

Fig. 6. Dataflow-equation schemas for shape analysis.

together with a node $\langle p,z \rangle$ for each variable $v_{\langle p,z \rangle}$ in Eqn_G. The edges of the graph, each of which is labeled with one of { $\boldsymbol{id}$, $\boldsymbol{hd}$, $\boldsymbol{tl}$, $\boldsymbol{hd}^{-1}$, $\boldsymbol{tl}^{-1}$ }, are defined as shown in the following table:

Form of equation	Edge(s) in the equation dependence graph	Label
$v_{\langle q,x \rangle} = \{ \textbf{at} \}$	$\textbf{atom} \to \langle q,x \rangle$	$\boldsymbol{id}$
$v_{\langle q,x \rangle} = \{ \textbf{nil} \}$	$\textbf{empty} \to \langle q,x \rangle$	$\boldsymbol{id}$
$v_{\langle q,x \rangle} = v_{\langle p,y \rangle}$	$\langle p,y \rangle \to \langle q,x \rangle$	$\boldsymbol{id}$
$v_{\langle q,x \rangle} = \textbf{cons}(v_{\langle p,y \rangle}, v_{\langle p,z \rangle})$	$\langle p,y \rangle \to \langle q,x \rangle$ $\langle p,z \rangle \to \langle q,x \rangle$	$\boldsymbol{hd}$ $\boldsymbol{tl}$
$v_{\langle q,x \rangle} = \textbf{car}(v_{\langle p,y \rangle})$	$\langle p,y \rangle \to \langle q,x \rangle$	$\boldsymbol{hd}^{-1}$
$v_{\langle q,x \rangle} = \textbf{cdr}(v_{\langle p,y \rangle})$	$\langle p,y \rangle \to \langle q,x \rangle$	$\boldsymbol{tl}^{-1}$

□

The equation dependence graph for this section's example is shown in Fig. 5.

Shape-analysis information can be obtained by solving *three* CFL-reachability problems on the equation dependence graph, using the following context-free grammars:

$$
\begin{array}{lll}
L_1: & id\_path & \to id\_path \;\; id\_path \\
 & & \;|\;\; \boldsymbol{hd} \;\; id\_path \;\; \boldsymbol{hd}^{-1} \\
 & & \;|\;\; \boldsymbol{tl} \;\; id\_path \;\; \boldsymbol{tl}^{-1} \\
 & & \;|\;\; \boldsymbol{id} \\
 & & \;|\;\; \varepsilon \\
L_2: & hd\_path & \to id\_path \;\; \boldsymbol{hd} \;\; id\_path \\
L_3: & tl\_path & \to id\_path \;\; \boldsymbol{tl} \;\; id\_path
\end{array}
$$

The language L_1 represents paths in which each $\boldsymbol{hd}$ ($\boldsymbol{tl}$) is balanced by a matching $\boldsymbol{hd}^{-1}$ ($\boldsymbol{tl}^{-1}$); these paths correspond to values transmitted along execution paths in which each cons operation (which gives rise to a $\boldsymbol{hd}$ or $\boldsymbol{tl}$ label on an edge in the path) is eventually "taken apart" by a matching car ($\boldsymbol{hd}^{-1}$) or cdr ($\boldsymbol{tl}^{-1}$) operation. Thus, the second and third rules of the L_1 grammar are the grammar-theoretic analogs of McCarthy's rules: "car(cons(x, y)) = x" and "cdr(cons(x, y)) = y" [21].

The language L_2 represents paths that are slightly unbalanced—those with one unmatched $\boldsymbol{hd}$; these paths correspond to the possible values that could be accessed by performing one additional **car** operation (which would extend the path with an additional $\boldsymbol{hd}^{-1}$). The language L_3 also represents paths that are slightly unbalanced—in this case, those with one unmatched $\boldsymbol{tl}$; these paths correspond to the possible values that could be accessed by performing one additional **cdr** operation (extending the path with $\boldsymbol{tl}^{-1}$).

Example. Suppose we are interested in characterizing the shape of program variable y just before the **exit** statement of the program shown in Fig. 5. We can determine information about the possible origin of the root constituent of y at *n12* by solving the single-target L_1-path problem for $\langle n12,y \rangle$. This yields the set { $\langle n12,y \rangle$, $\langle n8,y \rangle$, $\langle n11,y \rangle$, **empty** }. This indicates that y is either nil or was allocated at *n10* during an execution of the second while loop. Similarly, the solution to the single-target L_2-path problem for $\langle n12,y \rangle$ is the set { $\langle n10,temp \rangle$, $\langle n5,z \rangle$, $\langle n4,z \rangle$, **atom** }. This indicates that the atom in car(y) is one originally read in as the value of z. (See Fig. 5, which shows an L_2-path from **atom** to $\langle n12,y \rangle$.) Finally, the solution to the single-target L_3-path problem for $\langle n12,y \rangle$ is the set { $\langle n10,y \rangle$, $\langle n9,y \rangle$, $\langle n8,y \rangle$, $\langle n11,y \rangle$, **empty** }. This indicates that the tail of y is either nil or was allocated at *n10* during an execution of the second while loop.

This information can be interpreted as the following regular-tree grammar:

$$
\begin{array}{ll}
\langle n12,y \rangle \to & \langle n12,y \rangle \mid \langle n8,y \rangle \mid \langle n11,y \rangle \mid \textbf{empty} \\
 & \mid\; \textbf{cons}(\langle n10,temp \rangle \mid \langle n5,z \rangle \mid \langle n4,z \rangle \mid \textbf{atom},\; \langle n10,y \rangle \mid \langle n9,y \rangle \mid \langle n8,y \rangle \mid \langle n11,y \rangle \mid \textbf{empty})
\end{array}
$$

□

4. Algorithms for Solving CFL-Reachability Problems

CFL-reachability problems can be solved via a simple dynamic-programming algorithm. The grammar is first normalized by introducing new nonterminals wherever necessary so that the right-hand side of each production has at most two symbols (either terminals or nonterminals). Then, additional edges are added to the graph according to the patterns shown in Fig. 7 until no more edges can be added. The solution is obtained from the edges labeled with the grammar's root symbol. When an appropriate worklist algorithm is used, the running time of this algorithm is cubic in the number of nodes in the graph [22]. (This algorithm can be thought of as a generalization of the

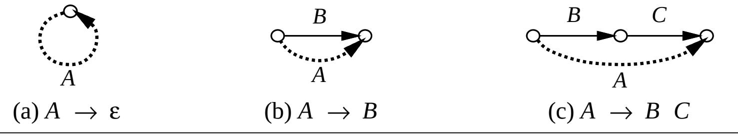


Fig. 7. Patterns for adding edges to solve a CFL-reachability problem. In each case, the dotted edge is added to the graph.

CYK algorithm for CFL-recognition [38].)

Although all CFL-reachability problems can be solved in time cubic in the number of graph nodes, one can sometimes do asymptotically better than this by taking advantage of the structure of the graph that arises in a program-analysis problem. For instance, with IFDS problems, the number of nodes in the exploded supergraph is *ND*, where *N* is the number of nodes in the supergraph and *D* is the size of the universe of dataflow facts. However, by taking advantage of the structure of the exploded supergraph, IFDS problems can be solved in time $O(ED^3)$, which is asymptotically better than the general-case time bound of $O(N^3D^3)$ [28,14]. A similar improvement over the general-case time bound can be obtained for interprocedural slicing, as well [26].

5. Solving Demand Versions of Program-Analysis Problems

An exhaustive dataflow-analysis algorithm associates with each point in a program a set of "dataflow facts" that are guaranteed to hold whenever that point is reached during program execution. By contrast, a *demand* dataflow-analysis algorithm determines whether a single given dataflow fact holds at a single given point [1,27,7,14]. Demand analysis can sometimes be preferable to exhaustive analysis for the following reasons:

- *Narrowing the focus to specific points of interest*. Software-engineering tools that use dataflow analysis often require information only at a certain set of program points. Similarly, in program optimization, most of the gains are obtained from making improvements at a program's "hot spots"—in particular, its innermost loops. The use of a demand algorithm has, in some cases, the potential to reduce greatly the amount of extraneous information computed.
- *Narrowing the focus to specific dataflow facts of interest*. Even when dataflow information is desired for every program point *p*, the full set of dataflow facts at *p* may not be required. For example, for the uninitialized-variables problem we are ordinarily interested in determining only whether the variables *used* at *p* might be uninitialized, rather than determining that information at *p* for *all* variables.
- *Reducing work in preliminary phases*. In problems that can be decomposed into separate phases, not all of the information from one phase may be required by subsequent phases. For example, the MayMod problem determines, for each call site, which variables may be modified during the call. This problem can be decomposed into two phases: computing side effects disregarding aliases (the so-called DMod problem), and computing alias information [3]. Given a demand (*e.g.*, "What is the MayMod set for a given call site *c*?"), a demand algorithm has the potential to reduce drastically the amount of work spent in earlier phases by propagating only relevant demands (*e.g.*, "What are the alias pairs (x, y) such that *x* is in DMod(*c*)"?).
- *Sidestepping incremental-updating problems*. A transformation performed at one point in the program can affect previously computed dataflow information at other points in the program. In many cases, the old information at such points is no longer safe; the dataflow information needs to be updated before it is possible to perform further transformations at such points. Incremental dataflow analysis could be used to maintain complete information at all program points; however, updating all invalidated information can be expensive. An alternative is to demand only the dataflow information needed to validate a proposed transformation; each demand would be solved using the current program, so the answer would be up-to-date.
- *Demand analysis as a user-level operation*. It is desirable to have program-development tools in which the user can ask questions interactively about various aspects of a program [20]. 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 it is unlikely that a programmer will ask questions about all program points, solving just the user's sequence of demands is likely to be significantly less costly than performing an exhaustive analysis.

Of course, determining whether a given fact holds at a given point may require determining whether other, related facts hold at other points (and those other facts may not be "facts of interest" in the sense of the second bullet-point above). It is desirable, therefore, for a demand-driven program-analysis algorithm to minimize the amount of such auxiliary information computed.

For program-analysis problems that have been transformed into graph-reachability problems, demand algorithms are obtained for free, by solving a single-target or multi-target graph-reachability problem. For instance, the problem transformation described in Section 3.1 has been used to devise demand algorithms for interprocedural dataflow analysis [25,14,13]. Because an algorithm for solving single-target (or multi-target) reachability problems focuses on the nodes that reach the specific target(s), it minimizes the amount of extraneous information computed.

In the case of IFDS problems, to answer a single demand we need to solve a single-source/single-target $L(\textit{realizable})$-path problem: "Is there a realizable path in $G^{\#}$ from node $\langle \textit{start}_{main}, \Lambda \rangle$ to node $\langle n,d \rangle$?" For an exhaustive algorithm, we need to solve a single-source $L(\textit{realizable})$-path problem: "What is the set of nodes $\langle n,d \rangle$ such that there is a realizable path in $G^{\#}$ from $\langle \textit{start}_{main}, \Lambda \rangle$ to $\langle n,d \rangle$?" In general, however, it is not known how to solve single-source/single-target (or single-source/multi-target) CFL-reachability problems any faster than single-source CFL-reachability problems. Experimental results showed that in situations when only a small number of demands are made, or when most demands are answered *yes*, a demand algorithm for IDFS problems (*i.e.*, for the single-source/single-target or single-source/multi-target $L(\textit{realizable})$-path problems) runs faster than an exhaustive algorithm (*i.e.*, for the single-source

$L(realizable)$-path problem) [14,13].

In the case of partially balanced parenthesis problems, it is possible to use a hybrid scheme; that is, one in between a pure exhaustive and a pure demand-driven approach. The hybrid approach takes advantage of the fact that there is a natural way to divide partially balanced parenthesis problems into two stages. The first stage is carried out in an exhaustive fashion, after which individual queries are answered on a demand-driven basis. In the description that follows, we explain the hybrid technique for backward interprocedural slicing [12,26]. A similar technique also applies in the case of IFDS problems.

The preprocessing step adds *summary edges* to the SDG. Each summary edge represents a matched path between an actual-in and an actual-out node (where the two nodes are associated with the same a call site). Let P be the number of procedures in the program; let E be the maximum number of control and flow edges in any procedure's PDG; and let *Params* be the the maximum number of actual-in vertices in any procedure's PDG. There are no more than $CallSites\ Params^2$ summary edges, and the task of identifying all summary edges can be performed in time $O((P \times E \times Params) + (CallSites \times Params^3))$ [26]. By the *augmented SDG*, we mean the SDG after all appropriate summary edges have been added to it.

The second, demand-driven, stage involves only *regular-reachability* problems on the augmented SDG. In the second stage, we use the following two linear grammars:

$$\begin{array}{rll} \textit{unbalanced-right}' \rightarrow & \textit{unbalanced-right}'\ \textit{summary} & \\ | & \textit{unbalanced-right}'\ e & \\ | & \textit{unbalanced-right}'\ )_i & 1 \le i \le \textit{CallSites} \\ | & \varepsilon & \end{array} \qquad \begin{array}{rll} \textit{realizable}' \rightarrow & \textit{summary realizable}' & \\ | & e\ \textit{realizable}' & \\ | & (_i\ \textit{realizable}' & 1 \le i \le \textit{CallSites} \\ | & \varepsilon & \end{array}$$

Suppose we wish to find the backward slice with respect to SDG node n. First, we solve the single-target $L(realizable')$-path problem for node n, which yields a set of nodes S. Let S' be the subset of actual-out nodes in S. The set of nodes in the slice is S together with the solution to the multi-target $L(unbalanced\text{-}right')$-path problem with respect to S'.

An advantage of this approach is that each regular-reachability problem—and hence each slice—can be solved in time linear in the number of nodes and edges in the augmented SDG; *i.e.*, in time $O((P \times E) + (CallSites \times Params^2))$.

This approach is used in the Wisconsin Program-Slicing Tool, a slicing system that supports essentially the full C language. (The system is available under license from the University of Wisconsin. It has been successfully applied to slice programs as large as 51,000 lines.)

6. Program Analysis Using More Than Graph Reachability

The graph-reachability approach offers insight into ways that machinery more powerful than the graph-reachability techniques described above can be brought to bear on program-analysis problems [27,31].

One way to generalize the CFL-reachability approach stems from the observation that CFL-reachability problems correspond to a restricted class of Datalog programs, so-called "chain programs": Each edge $m \rightarrow n$ labeled e is represented by a fact "$e(m,n)$."; each production $A \rightarrow B\ C$ is encoded as a chain rule "$a(X,Z) :- b(X,Y),\ c(Y,Z)$." A CFL-reachability problem can be solved using bottom-up semi-naive evaluation of the chain program [37]. This observation provides a way for program-analysis tools to take advantage of the methods developed in the logic-programming and deductive-database communities for the efficient evaluation of recursive queries in deductive databases, such as tabulation [35] and the Magic-sets transformation [2,4]. For instance, algorithms for demand versions of program-analysis problems can be obtained from their exhaustive counterparts essentially for free by specifying the problem with Horn clauses and then applying the "Magic-sets" transformation [27]. The fact that CFL-reachability problems are related to chain programs, together with the fact that chain programs are just a special case of the logic programs to which tabulation and transformation techniques apply, suggests that more powerful program-analysis algorithms can be obtained by going outside the class of pure chain programs [27].

A different way to generalize the CFL-reachability approach so as to bring more powerful techniques to bear on interprocedural dataflow analysis was presented in [31]. This method applies to problems in which the dataflow information at a program point is represented by a finite environment (*i.e.*, a mapping from a finite set of *symbols* to a finite-height domain of *values*), and the effect of a program operation is captured by a distributive "environment-transformer" function. Two of the dataflow-analysis problems that this framework handles are (decidable) variants of the constant-propagation problem: *copy-constant propagation* and *linear-constant propagation*. The former interprets assignment statements of the form $x = 7$ and $y = x$. The latter also interprets statements of the form $y = -2*x+5$.

By means of an "explosion transformation" similar to the one utilized in Section 3.1, an interprocedural distributive-environment-transformer problem can be transformed from a meet-over-all-realizable-paths problem on a program's supergraph to a meet-over-all-realizable-paths problem on a graph that is *larger*, but in which every edge is labeled with a much *simpler* edge function (a so-called "micro-function") [31]. Each micro-function on an edge $d_1 \rightarrow d_2$ captures the effect that the value of symbol d_1 in the argument environment has on the value of symbol d_2 in the result environment. Fig. 8 shows the exploded representations of four environment-transformer functions used in constant propagation. Fig. 8(a) shows how the identity function $\lambda env.env$ is represented. Figs. 8(b)–(d) show the representations of the functions $\lambda env.env[x \mapsto 7]$, $\lambda env.env[y \mapsto env(x)]$, and $\lambda env.env[y \mapsto -2*env(x)+5]$, which are the functions for the statements $x = 7$, $y = x$, and $y = -2*x+5$, respectively. (Λ is used to represent the effects of a function that are independent of the argument environment. Each graph includes an edge of the form $\Lambda \rightarrow \Lambda$, labeled with $\lambda v.v$; as in Section 3.1, these edges are needed to capture function composition properly.)

Dynamic programming on the exploded supergraph can be used to find the meet-over-all-realizable-paths solution to the original problem: An exhaustive algorithm can be used to find the values for all symbols at all program points; a demand algorithm can be used to find the value for an individual symbol at a particular program point [31]. An experiment was carried out in which the exhaustive and demand algorithms were used to perform constant propagation on 38 C programs, which ranged in size from 300 lines to 6,000 lines. The experiment found that

- In contrast to previous results for numeric Fortran programs [11], linear-constant propagation found more constants than copy-constant propagation in 6 of the 38 programs.
- The demand algorithm, when used to demand values for all uses of scalar integer variables, was faster than the exhaustive algorithm by a factor ranging from 1.14 to about 6.

Acknowledgements

This paper is based in part on joint work with S. Horwitz, M. Sagiv, G. Rosay, and D. Melski. The work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by NSF under grants DCR-8552602, CCR-9100424, and CCR-9625667, by DARPA (monitored by ONR under contracts N00014-88-K-0590 and N00014-92-J-1937), and by the Univ. of Wisconsin through a Vilas Associate Award.

References

1. Babich, W.A. and Jazayeri, M., "The method of attributes for data flow analysis: Part II. Demand analysis," *Acta Inf.* **10**(3) pp. 265-272 (Oct. 1978).
2. Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., "Magic sets and other strange ways to implement logic programs," pp. 1-15 in *Proc. of the Fifth ACM Symp. on Princ. of Database Syst.,* (Cambridge, MA, Mar. 1986), (1986).
3. Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," pp. 29-41 in *Conf. Rec. of the Sixth ACM Symp. on Princ. of Prog. Lang.,* (San Antonio, TX, Jan. 29-31, 1979), ACM, New York, NY (1979).
4. Beeri, C. and Ramakrishnan, R., "On the power of magic," pp. 269-293 in *Proc. of the Sixth ACM Symp. on Princ. of Database Syst.,* (San Diego, CA, Mar. 1987), (1987).
5. Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proc. of the ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *SIGPLAN Not.* **23**(7) pp. 47-56 (July 1988).
6. Cousot, P. and Cousot, R., "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," pp. 238-252 in *Conf. Rec. of the Fourth ACM Symp. on Princ. of Prog. Lang.,* (Los Angeles, CA, Jan. 17-19, 1977), ACM, New York, NY (1977).
7. Duesterwald, E., Gupta, R., and Soffa, M.L., "Demand-driven computation of interprocedural data flow," pp. 37-48 in *Conf. Rec. of the Twenty-Second ACM Symp. on Princ. of Prog. Lang.,* (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).
8. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.* **9**(3) pp. 319-349 (July 1987).
9. Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler,* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).
10. Giegerich, R., Möncke, U., and Wilhelm, R., "Invariance of approximative semantics with respect to program transformation," pp. 1-10 in *GI 81: 11th GI Conf., Inf.-Fach. 50,* Springer-Verlag, New York, NY (1981).
11. Grove, D. and Torczon, L., "Interprocedural constant propagation: A study of jump function implementation," pp. 90-99 in *Proc. of the ACM SIGPLAN 93 Conf. on Prog. Lang. Design and Implementation,* (Albuquerque, NM, June 23-25, 1993), ACM, New York, NY (June 1993).

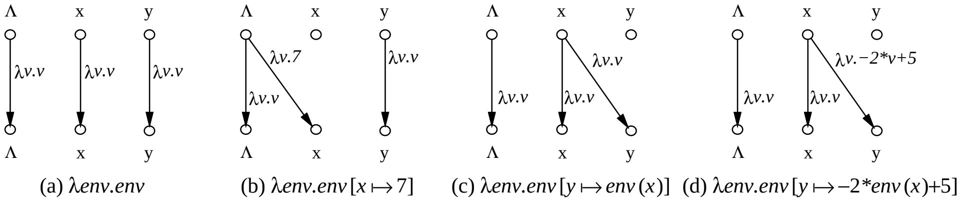


Fig. 8. The exploded representations of four environment-transformer functions used in constant propagation.

12. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (Jan. 1990).
13. Horwitz, S., Reps, T., and Sagiv, M., "Demand interprocedural dataflow analysis," TR-1283, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI (Aug. 1995).
14. Horwitz, S., Reps, T., and Sagiv, M., "Demand interprocedural dataflow analysis," *SIGSOFT 95: Proc. of the Third ACM SIGSOFT Symp. on the Found. of Softw. Eng.,* (Wash., DC, Oct. 10-13, 1995), *ACM SIGSOFT Softw. Eng. Notes* **20**(4) pp. 104-115 (1995).
15. Jones, N.D. and Muchnick, S.S., "Flow analysis and optimization of Lisp-like structures," pp. 102-131 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).
16. Kildall, G., "A unified approach to global program optimization," pp. 194-206 in *Conf. Rec. of the First ACM Symp. on Princ. of Prog. Lang.,* ACM, New York, NY (1973).
17. Knoop, J. and Steffen, B., "The interprocedural coincidence theorem," pp. 125-140 in *Proc. of the Fourth Int. Conf. on Comp. Construct.,* (Paderborn, FRG, Oct. 5-7, 1992), *Lec. Notes in Comp. Sci.,* Vol. 641, ed. U. Kastens and P. Pfahler,Springer-Verlag, New York, NY (1992).
18. Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Dependence graphs and compiler optimizations," pp. 207-218 in *Conf. Rec. of the Eighth ACM Symp. on Princ. of Prog. Lang.,* (Williamsburg, VA, Jan. 26-28, 1981), ACM, New York, NY (1981).
19. Landi, W. and Ryder, B.G., "Pointer-induced aliasing: A problem classification," pp. 93-103 in *Conf. Rec. of the Eighteenth ACM Symp. on Princ. of Prog. Lang.,* (Orlando, FL, Jan. 1991), ACM, New York, NY (1991).
20. Masinter, L.M., "Global program analysis in an interactive environment," Tech. Rep. SSL-80-1, Xerox Palo Alto Res. Cent., Palo Alto, CA (Jan. 1980).
21. McCarthy, J., "A basis for a mathematical theory of computation," pp. 33-70 in *Computer Programming and Formal Systems*, ed. Braffort and Hershberg,North-Holland, Amsterdam (1963).
22. Melski, D. and Reps, T., "Interconvertibility of set constraints and context-free language reachability," pp. 74-89 in *Proc. of the ACM SIGPLAN Symp. on Part. Eval. and Sem.-Based Prog. Manip. (PEPM 97),* (Amsterdam, The Netherlands, June 12-13, 1997), ACM, New York, NY (1997).
23. Mogensen, T., "Separating binding times in language specifications," pp. 12-25 in *Fourth Int. Conf. on Func. Prog. and Comp. Arch.,* (London, UK, Sept. 11-13, 1989), ACM, New York, NY (1989).
24. Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proc. of the ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. on Practical Softw. Develop. Env.,* (Pittsburgh, PA, Apr. 23-25, 1984), *SIGPLAN Not.* **19**(5) pp. 177-184 (May 1984).
25. Reps, T., Sagiv, M., and Horwitz, S., "Interprocedural dataflow analysis via graph reachability," TR 94-14, Datalogisk Institut, Univ. of Copenhagen, Copenhagen, Denmark (Apr. 1994).
26. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," *SIGSOFT 94: Proc. of the Second ACM SIGSOFT Symp. on the Found. of Softw. Eng.,* (New Orleans, LA, Dec. 7-9, 1994), *ACM SIGSOFT Softw. Eng. Notes* **19**(5) pp. 11-20 (Dec. 1994).
27. Reps, T., "Demand interprocedural program analysis using logic databases," pp. 163-196 in *Applications of Logic Databases*, ed. R. Ramakrishnan,Kluwer Academic Publishers, Boston, MA (1994).
28. Reps, T., Horwitz, S., and Sagiv, M., "Precise interprocedural dataflow analysis via graph reachability," pp. 49-61 in *Conf. Rec. of the Twenty-Second ACM Symp. on Princ. of Prog. Lang.,* (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).
29. Reps, T., "On the sequential nature of interprocedural program-analysis problems," *Acta Inf.* **33** pp. 739-757 (1996).
30. Reynolds, J.C., "Automatic computation of data set definitions," pp. 456-461 in *Information Processing 68: Proc. of the IFIP Congress 68*, North-Holland, New York, NY (1968).
31. Sagiv, M., Reps, T., and Horwitz, S., "Precise interprocedural dataflow analysis with applications to constant propagation," *Theor. Comp. Sci.* **167** pp. 131-170 (1996).
32. Sharir, M. and Pnueli, A., "Two approaches to interprocedural data flow analysis," pp. 189-233 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).
33. Tip, F., "A survey of program slicing techniques," *J. Program. Lang.* **3** pp. 121-181 (1995).
34. Valiant, L.G., "General context-free recognition in less than cubic time," *J. Comp. Syst. Sci.* **10**(2) pp. 308-315 (Apr. 1975).
35. Warren, D.S., "Memoing for logic programs," *Commun. ACM* **35**(3) pp. 93-111 (Mar. 1992).
36. Weiser, M., "Program slicing," *IEEE Trans. on Softw. Eng.* **SE-10**(4) pp. 352-357 (July 1984).
37. Yannakakis, M., "Graph-theoretic methods in database theory," pp. 230-242 in *Proc. of the Ninth ACM Symp. on Princ. of Database Syst.,* (1990).
38. Younger, D.H., "Recognition and parsing of context-free languages in time *n***3," *Inf. and Cont.* **10** pp. 189-208 (1967).
39. Zadeck, F.K., "Incremental data flow analysis in a structured program editor," *Proc. of the SIGPLAN 84 Symp. on Comp. Construct.,* (Montreal, Can., June 20-22, 1984), *SIGPLAN Not.* **19**(6) pp. 132-143 (June 1984).