Program Analysis via Graph Reachability

Thomas Reps
University of Wisconsin

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, one version of the problem of approximating the possible "shapes" that heap-allocated structures in a program can take on, and flow-insensitive points-to analysis. 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 [25]. 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 nodes or 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 forward 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: $kill_p$ represents dataflow facts "removed" by p, and gen_p represents dataflow facts "created" at 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:

An abbreviated version of this paper appeared as an invited paper in the Proceedings of the 1997 International Symposium on Logic Programming [84].

This 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), by grants from Rockwell and IBM, and by a Vilas Associate Award from the University of Wisconsin.

Author's address: Computer Sciences Department, University of Wisconsin−Madison, 1210 W. Dayton St., Madison, WI 53706.
E-mail: reps@cs.wisc.edu
WWW: http://www.cs.wisc.edu/˜reps/

- 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
- Flow-insensitive points-to analysis.

The first, second, and fourth of these applications of CFL-reachability apply to programs written in an imperative programming language, such as C. The application of CFL-reachability to shape analysis applies to both functional and imperative languages that permit the use of heap-allocated storage, but do not permit destructive updating of fields.

There are a number of benefits to be gained from using graph reachability as a vantage point for studying program-analysis problems:

- By expressing a program-analysis problem as a graph-reachability problem, we can 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.
- 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 [11,102,78,30,49,88]. For program-analysis problems that can be expressed as CFL-reachability problems, demand algorithms are easily obtained by solving single-source, single-target, multi-source, or multi-target CFL-reachability problems [49].
- Graph reachability offers insight into the “$O(n^3)$ bottleneck” that exists for certain kinds of program-analysis problems. That is, a number of program-analysis problems are known to be solvable in time $O(n^3)$, but no sub-cubic-time algorithm is known. This is sometimes erroneously attributed to the need to perform transitive closure when a problem is solved. However, because transitive closure can be performed in sub-cubic time [95], this is not the correct explanation. We have long believed that, in many cases, the real source of the $O(n^3)$ bottleneck is that a CFL-reachability problem needs to be solved. The constructions given in [66,67] show that this is indeed the case for certain set-constraint problems.

  The source of the $O(n^3)$ bottleneck has also been attributed to the need to solve a *dynamic transitive-closure problem*. The basis for this statement is that several cubic-time algorithms for solving program-analysis problems maintain the transitive closure of a relation in an on-line fashion (*i.e.*, as a sequence of insertions into the relation is performed). At the present time, no sub-cubic-time algorithm is known for this version of the dynamic transitive-closure problem. In our opinion, the statement “a dynamic transitive-closure problem needs to be solved” provides an *operational* characterization of the $O(n^3)$ bottleneck, whereas our alternative characterization—namely, “a CFL-reachability problem needs to be solved”—offers a *declarative* characterization of the source of the $O(n^3)$ bottleneck.
- 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 [82]. 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 [78,88].

The remainder of the paper is organized into six sections, as follows: Section 2 defines CFL-reachability. Section 3 discusses algorithms for solving CFL-reachability problems. Section 4 discusses how the graph-reachability approach can be used to tackle interprocedural dataflow analysis, interprocedural program slicing, shape analysis, and flow-insensitive points-to analysis. Section 5 concerns demand versions of program-analysis problems. Section 6 describes some techniques that go beyond pure graph reachability. Section 7 discusses related work.

2. Context-Free-Language Reachability Problems

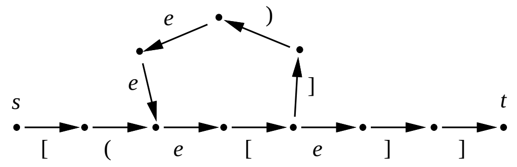
The theme of this paper is that a number of program-analysis problems can be viewed as instances of a more general problem: *CFL-reachability*. A CFL-reachability problem is not an ordinary reachability problem (*e.g.*, transitive closure), but one in which a path is considered to connect two nodes only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



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

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 ith edge with the ith letter of ω. There is an L-path from s to t iff $\omega \in L$ [100].

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 3). 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 [95]. The algorithm handles CFL-reachability problems on trees and directed acyclic graphs, as well as on chain graphs. Unfortunately, the algorithm does not seem to generalize to CFL-reachability problems on arbitrary graphs.

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.

Specific applications of CFL-reachability to program-analysis problems are discussed at length in Section 4. For the moment, we will content ourselves merely with an example to illustrate how a context-free language can be used to exclude from attention paths that clearly represent infeasible computations.

In the case of interprocedural dataflow analysis, we can characterize a superset of the feasible execution paths—and thereby eliminate from consideration a subset of the infeasible execution 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 execution paths 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—one of which represents the program's main procedure. Each flowgraph has a unique *start* node and a unique *exit* node. The other nodes of the flowgraph represent statements and predicates of the program in the usual way,[1] except that 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 of G^* 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:

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

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

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

The language $L(\mathit{realizable})$ is a language of *partially* balanced parentheses: Every right parenthesis "$)_i$" is

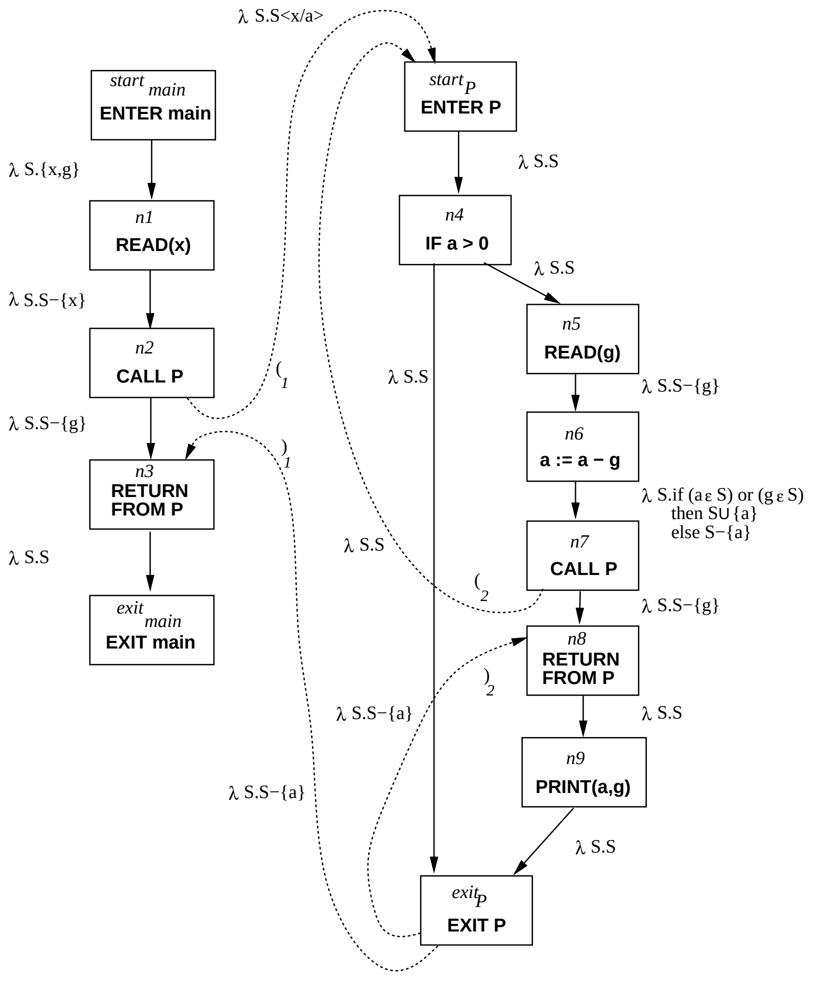
[1]The nodes of a flowgraph can represent individual statements and predicates; alternatively, they can represent basic blocks.

```
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.

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 "$start_{main} \to n1 \to n2 \to start_P \to n4 \to exit_P \to n3$", 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 "$start_{main} \to n1 \to n2 \to start_P \to n4$", which has word "$ee(_1e$", is a realizable path but not a matched path: The call-to-start edge $n2 \to start_P$ 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. The pending activation records correspond to the unmatched $(_i$'s in the path's word.
- The path "$start_{main} \to n1 \to n2 \to start_P \to n4 \to exit_P \to n8$", which has word "$ee(_1ee)_2$", is neither a matched path nor a realizable path: The exit-to-return-site edge $exit_P \to n8$ does not correspond to the preceding call-to-start edge $n2 \to start_P$. This path represents an infeasible execution path.

3. 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. 2 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 (together with suitable indexing structures), the running time of this algorithm is cubic in the number of nodes in the graph [66]. (This algorithm can be thought of as a generalization of the CYK algorithm for CFL-recognition [101].)

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, for the class of dataflow-analysis problems discussed in Section 4.1, the number of graph nodes is ND, where N is the number of nodes in the program's supergraph and D is the size of the universe of dataflow facts. However, by taking advantage of some special structural properties of the graph, the CFL-reachability problems that arise from the construction described in Section 4.1 can be solved in time $O(ED^3)$, where E is the number of edges in the program's supergraph, which is asymptotically better than the general-case time bound of $O(N^3 D^3)$ [81,49]. A similar improvement over the general-case time bound can be obtained for the interprocedural-slicing problem, discussed in Section 4.2 [77].

Another way of approaching CFL-reachability problems 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 in the graph is represented by a fact "$e(m,n)$."; each production of the form

$$p \rightarrow q_0 \;\; q_1 \;\; \cdots \;\; q_i \;\; \cdots \;\; q_k$$

in the context-free grammar (where the q_i are either terminals or nonterminals) is encoded as a chain rule, *i.e.*, a rule of the form

$$p(X,Y) :- q_0(X,Z_1),\, q_1(Z_1,Z_2),\; \cdots,\, q_i(Z_i,Z_{i+1}),\; \cdots,\, q_k(Z_k,Y).$$

A CFL-reachability problem can be solved using bottom-up semi-naive evaluation of the chain program [100]. 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 [96] and the Magic-sets transformation [86,13,17,94]. 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 [79,78,80].

Note that any Datalog problem in which all the rules are chain rules can be immediately converted to a CFL-reachability problem. The graph is constructed from the given set of base facts (which must all involve binary relations): For each fact "$e(m,n)$.", there is an edge from node m to node n, labeled e. Each Datalog rule of the form

$$p(X,Y) :- q_0(X,Z_1),\, q_1(Z_1,Z_2),\; \cdots,\, q_i(Z_i,Z_{i+1}),\; \cdots,\, q_k(Z_k,Y).$$

is converted to a context-free-grammar production of the form

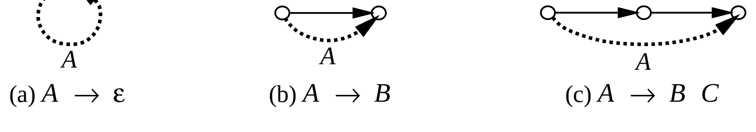


Fig. 2. Patterns for adding edges to solve a CFL-reachability problem. Symbols B and C are either terminals or nonterminals. In each case, the dotted edge labeled with nonterminal A is added to the graph.

$$p \rightarrow q_0 \ q_1 \ \cdots \ q_i \ \cdots \ q_k$$

We will make use of this transformation in Section 4.4.

4. Four Examples

In this section, we show how four program-analysis problems can be transformed into CFL-reachability problems. The first three problems discussed illustrate the use of only a limited class of context-free languages; our constructions make use of partially balanced parenthesis problems, similar to the language $L(\mathit{realizable})$ defined in Section 2 (see Sections 4.1–4.3). The application of CFL-reachability to flow-insensitive points-to analysis, presented in Section 4.4, provides an example of a program-analysis problem that can be solved by expressing it as an L-path problem, where L is a context-free language that is something other than a language of partially balanced parentheses (see also [66,67]).

4.1. 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 supergraph 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 $V \rightarrow V$) to the edges of the supergraph.

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 $n6 \rightarrow n7$ 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: $MOP_n = \bigsqcap_{q \in \mathrm{Paths}(\mathit{start},\, n)} pf_q(\bot)$, where $\mathrm{Paths}(\mathit{start}, n)$ denotes the set of paths in the control-flow graph from the start node to n [54].[2] MOP_n represents a summary of the possible execution states that can arise at n; $\bot \in V$ is a special value that represents the execution state at the beginning of the program; $pf_q(\bot)$ 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: $MRP_n = \bigsqcap_{q \in \mathrm{RPaths}(\mathit{start}_{main},\, n)} pf_q(\bot)$, where $\mathrm{RPaths}(\mathit{start}_{main}, n)$ 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 $L(\mathit{realizable})$ defined in Section 2) [90,20,59,55,81,30]. Although some realizable paths may also be infeasible execution paths, none of the non-realizable paths are feasible execution paths. By restricting attention to just the realizable paths from start_{main}, we thereby exclude some of the infeasible execution paths. In general, therefore, MRP_n characterizes the execution state at n more precisely than MOP_n.

---

[2]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.

The *interprocedural, finite, distributive, subset problems* (*IFDS problems*) are those interprocedural dataflow-analysis problems that involve (i) a finite set of dataflow facts, and (ii) 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; it is not hard to show that IFDS problems in which the meet operator is intersection can always be handled by dualizing, *i.e.*, by transforming such a problem into a complementary problem in which the meet operator is union [76]. (Informally, if the "must-be-X" problem is an intersection IFDS problem, then the "may-not-be-X" problem is a union IFDS problem. Furthermore, for each node of G^*, the solution to the "must-be-X" problem is the complement, with respect to D, of the solution to the "may-not-be-X" problem.)

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, live variables, *etc.* In addition, the IFDS problems include many non-gen/kill problems, including possibly-uninitialized variables, truly-live variables [35], and copy-constant propagation [33, 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.

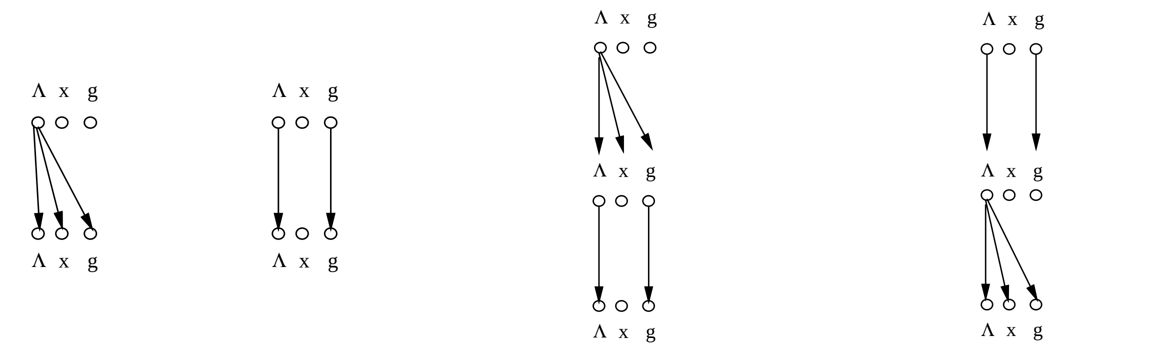
One way to solve an IFDS problem is to convert it into a realizable-path reachability problem [81,49]. For each problem instance, we build an *exploded supergraph* $G^\#$, in which each node $\langle n,d \rangle$ represents dataflow fact $d \in D$ at supergraph node n, and each edge represents a dependence 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. 3(a).

The representation relation for the function $\lambda S.S - \{x\}$ (which is associated with the edge from *n1* to *n2*) is shown in Fig. 3(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. 3(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



(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. 3. Representation relations for two functions and the two ways of composing the functions.

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. 3(c). Paths in a "pasted-together" graph represent the result of applying the composed function. For example, in Fig. 3(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 $\Lambda \rightarrow \Lambda$ 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. 3(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. 3(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 $m \rightarrow n$ in G^* is "exploded" into the representation relation of the function associated with $m \rightarrow n$. 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 $m \rightarrow n$ 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 [76].)

Example. The exploded supergraph that corresponds to the instance of the "possibly-uninitialized variables" problem shown in Fig. 1 is shown in Fig. 4. The dataflow functions are replaced by their representation relations. In Fig. 4, 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

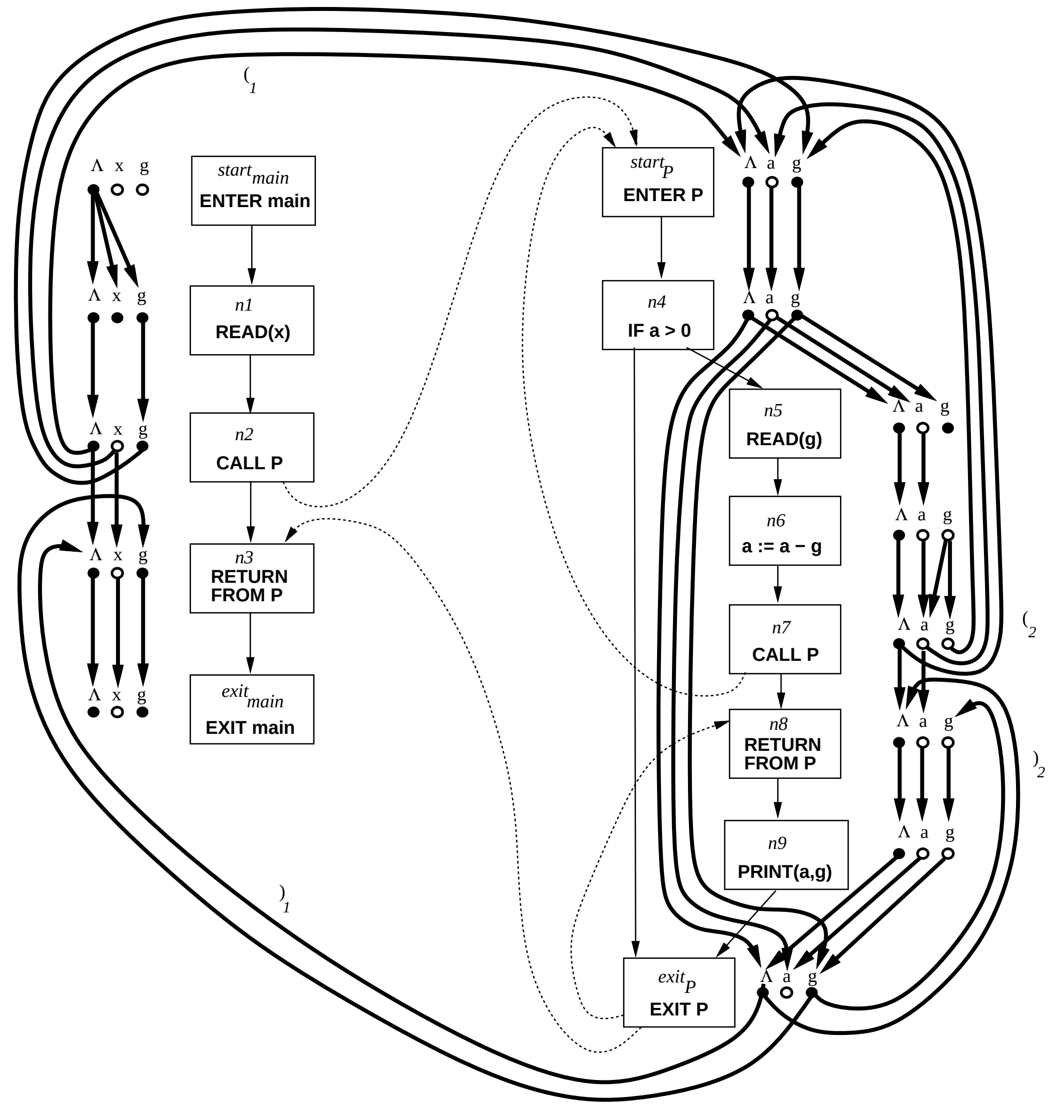


Fig. 4. The exploded supergraph that corresponds to the instance of the possibly-uninitialized variables problem shown in Fig. 1. Closed circles represent nodes of $G^{\#}$ that are reachable along realizable paths from $\langle start_{main}, \Lambda \rangle$. Open circles represent nodes not reachable along such paths.

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-uninitialized variable just before execution reaches the exit node of procedure P.) In Fig. 4, 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 $\langle exit_P, a \rangle$. □

4.2. 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 [98,72,32,46,77,92,19]. Slicing, and subsequent manipulation of slices, has applications in many software-engineering tools, including tools for program understanding, maintenance [34], debugging [61], testing [18,16], differencing [45,47], specialization [83], reuse [71], and merging [45].

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. 5.

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 [72], 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 the control and flow dependences among the procedure's statements and predicates [58,72,32]. 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 [46]. 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

```
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);
}
```

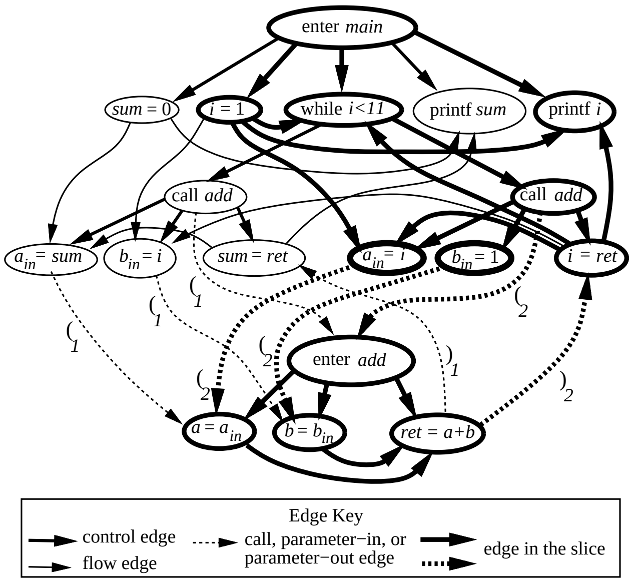


Fig. 5. A program, the slice of the program with respect to the statement *printf("i = %d\n", i)*, and the program's system dependence graph. In the slice, the starting point for the slice is shown in italics, and the empty boxes indicate where program elements have been removed from the original program. In the dependence graph, the edges shown in boldface are the edges in the slice.

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. 5, the graph shown on the right is the SDG for the program that appears on the left.

It should be noted that SDGs are really a *class* of program representations, and there is a sense in which the term "SDG" has a generic meaning: To represent programs in different programming languages one would use different kinds of PDGs, depending on the features and constructs of the given language. A large body of work exists concerning techniques for building dependence graphs for a wide variety of programming-language features and constructs. For example, previous work has addressed arrays [14,99,63,36,73,74], reference parameters [46], pointers [60,44,21,87], and non-structured control flow [12,22,2].

The issue of how to create appropriate PDGs/SDGs is mostly orthogonal to the issue of how to slice them. Once an SDG has been constructed, slicing can be formulated as a CFL-reachability problem.

One algorithm for interprocedural slicing was presented in Weiser's original paper on slicing [98]. 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. 5 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 feasible 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. 5.)

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 we are dealing with a program that does not have any unreachable procedures (*i.e.*, all procedures are transitively callable from *main*), we can change the backward-slicing 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 $L(slice)$:

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

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 $L(unbalanced\text{-}right)$-path and s is an $L(realizable)$-path. Because of the assumption that all procedures are 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. Consequently, the path $p||q||r||s$ is of the form $realizable||matched||realizable$, which can be shown to be an $L(realizable)$-path.

4.3. Shape Analysis

Shape analysis is concerned with finding approximations to the possible “shapes” that heap-allocated structures in a program can take on [85,50,68]. 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. 6. 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. □

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 [85,50]. The domain Shape of shape descrip-

```
x := nil
read(z)
while z ≠ 0 do
  x := cons(z, x)
  read(z)
od
y := nil
while x ≠ nil do
  temp := car(x)
  y := cons(temp, y)
  x := cdr(x)
od
```

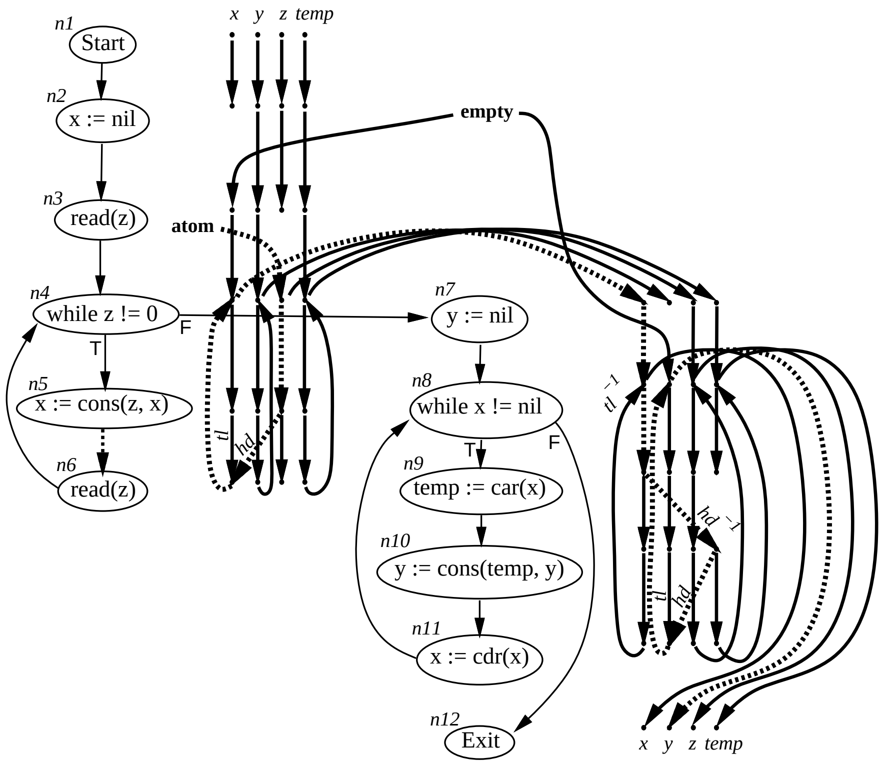


Fig. 6. A program, its control-flow graph, and its equation dependence graph. All edges of the equation dependence graph shown without labels have the label ***id***. The path shown by the dotted lines is an L (*hd_path*)-path from **atom** to node ⟨*n12*,*y*⟩.

tors 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. 7.

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 [85,50]. 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 [68]. 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* [85], or, equivalently, as a *regular-tree grammar* [50,68].

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 ··· then p: A := cons(B, C)
else q: A := cons(D, E)
fi
r: ···
```

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. 7. Dataflow-equation schemas for shape analysis.

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:

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

Grammar (i) uses multiple **cons** right-hand sides for a given nonterminal [50]. 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 [68]. 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 4.1. 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**, 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
$v_{\langle q,x\rangle} = \{ \mathbf{at} \}$	$\mathbf{atom} \rightarrow \langle q,x\rangle$	$\boldsymbol{id}$
$v_{\langle q,x\rangle} = \{ \mathbf{nil} \}$	$\mathbf{empty} \rightarrow \langle q,x\rangle$	$\boldsymbol{id}$
$v_{\langle q,x\rangle} = v_{\langle p,y\rangle}$	$\langle p,y\rangle \rightarrow \langle q,x\rangle$	$\boldsymbol{id}$
$v_{\langle q,x\rangle} = \mathbf{cons}(v_{\langle p,y\rangle}, v_{\langle p,z\rangle})$	$\langle p,y\rangle \rightarrow \langle q,x\rangle$ $\langle p,z\rangle \rightarrow \langle q,x\rangle$	$\boldsymbol{hd}$ $\boldsymbol{tl}$
$v_{\langle q,x\rangle} = \mathbf{car}(v_{\langle p,y\rangle})$	$\langle p,y\rangle \rightarrow \langle q,x\rangle$	$\boldsymbol{hd}^{-1}$
$v_{\langle q,x\rangle} = \mathbf{cdr}(v_{\langle p,y\rangle})$	$\langle p,y\rangle \rightarrow \langle q,x\rangle$	$\boldsymbol{tl}^{-1}$

□

The equation dependence graph for this section's example is shown in Fig. 6.

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

$$\begin{array}{llcl}
L_1: & id\_path & \rightarrow & id\_path\ \ id\_path \\
 & & \mid & \boldsymbol{hd}\ \ id\_path\ \ \boldsymbol{hd}^{-1} \\
 & & \mid & \boldsymbol{tl}\ \ id\_path\ \ \boldsymbol{tl}^{-1} \\
 & & \mid & \boldsymbol{id} \\
 & & \mid & \varepsilon \\
L_2: & hd\_path & \rightarrow & id\_path\ \ \boldsymbol{hd}\ \ id\_path \\
L_3: & tl\_path & \rightarrow & id\_path\ \ \boldsymbol{tl}\ \ id\_path
\end{array}$$

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" [64].

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. 6. 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 $\{ \langle n12,y\rangle, \langle n8,y\rangle, \langle n11,y\rangle, \mathbf{empty} \}$. 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 $\{ \langle n10,temp\rangle, \langle n5,z\rangle, \langle n4,z\rangle, \mathbf{atom} \}$. This indicates that the atom in car(y) is one originally read in as the value of z. (See Fig. 6, 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 $\{ \langle n10,y\rangle, \langle n9,y\rangle, \langle n8,y\rangle, \langle n11,y\rangle, \mathbf{empty} \}$. 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:

$$\begin{aligned}\langle n12,y\rangle \rightarrow\ & \langle n12,y\rangle \mid \langle n8,y\rangle \mid \langle n11,y\rangle \mid \textbf{empty}\\ \mid\ & \textbf{cons}(\langle n10,temp\rangle \mid \langle n5,z\rangle \mid \langle n4,z\rangle \mid \textbf{atom},\ \langle n10,y\rangle \mid \langle n9,y\rangle \mid \langle n8,y\rangle \mid \langle n11,y\rangle \mid \textbf{empty})\end{aligned}\qquad \square$$

4.4. Flow-Insensitive Points-To Analysis

In languages like C, analysis of the ways pointers are used in a program is crucial for obtaining good results from static analysis. Direct use of the results from pointer analysis can allow optimizations to be performed that would otherwise not be possible. Pointer analysis is also useful as a preprocessing step for other static analyses, with the payback being that it permits more precise information to be obtained: In many dataflow problems, the dataflow function at a particular point depends on the set of memory locations that a pointer variable may point to; when a better estimate can be provided for this set, a more precise dataflow function can be employed.

"Points-to analysis" is one approach to pointer analysis. It is concerned with determining, for each point in the program, (a superset of) the set of variables that a pointer variable might point to during execution. Even though points-to analysis concentrates on variables (*i.e.*, on stack-allocated storage), heap-allocated storage need not be completely ignored in points-to analysis. A common approach to handling heap-allocated storage is to fold together into a single, allocation-site-specific, pseudo-variable all of the memory locations that are allocated at an individual allocation site. For example, the C statement

```
s1: p = malloc(...);
```

would, for purposes of analysis, be treated as the statement

```
s1: p = &malloc_s1;
```

where malloc_s1 is a newly created variable.

A flow-insensitive analysis of a program is one that ignores the actual structure of the program's control-flow graph, and instead assumes that any statement can be executed immediately after any other statement. (Many type-inference algorithms are examples of flow-insensitive analyses.) Flow-insensitive points-to analyses have been developed by Andersen [9], Steensgaard [91], and Shapiro and Horwitz [89].

In this section, we show how a variant of Andersen's analysis, as reformulated by Shapiro and Horwitz [89], can be expressed as a CFL-reachability problem. Throughout the section, we assume that the assignment statements in the program have been normalized, by suitable introduction of temporary variables, so that all assignment statements have one of the following four forms:

```
p = &q;     p = q;     p = *q;     *p = q;
```

We also assume that malloc statements are handled via the transformation discussed above.

Shapiro and Horwitz reformulate Andersen's algorithm as one that builds up a graph that represents points-to relationships among the program's variables [89].[3] The nodes of the points-to graph represent the program's variables; an edge from the node for variable p to the node for variable q means "p might point to q". As the algorithm processes the assignment statements of the program, additional edges may be added to the points-to graph. Figure 8(b) illustrates the rules that are applied for each of the different kinds of assignment statements. In general, statements are considered multiple times; the rules are applied iteratively until a graph is obtained in which the application of the appropriate rule for each statement in the program fails to add any additional edges to the points-to graph.

Example 4.2. Suppose that the program contains the following assignment statements:

[3]Reference [89] does not actually give a full description of the method; only an example of the final points-to graph is given, and the steps by which the graph is created are not explained.

Statement		Fact generated
p = &q;	⇒	assignAddr(p,q).
p = q;	⇒	assign(p,q).
p = *q;	⇒	assignStar(p,q).
*p = q;	⇒	starAssign(p,q).
	(a)	

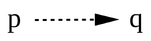
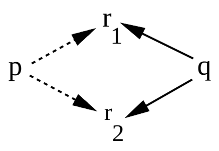
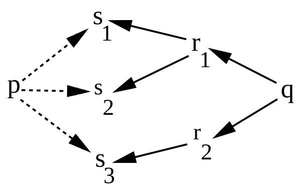
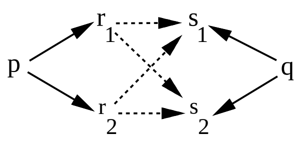
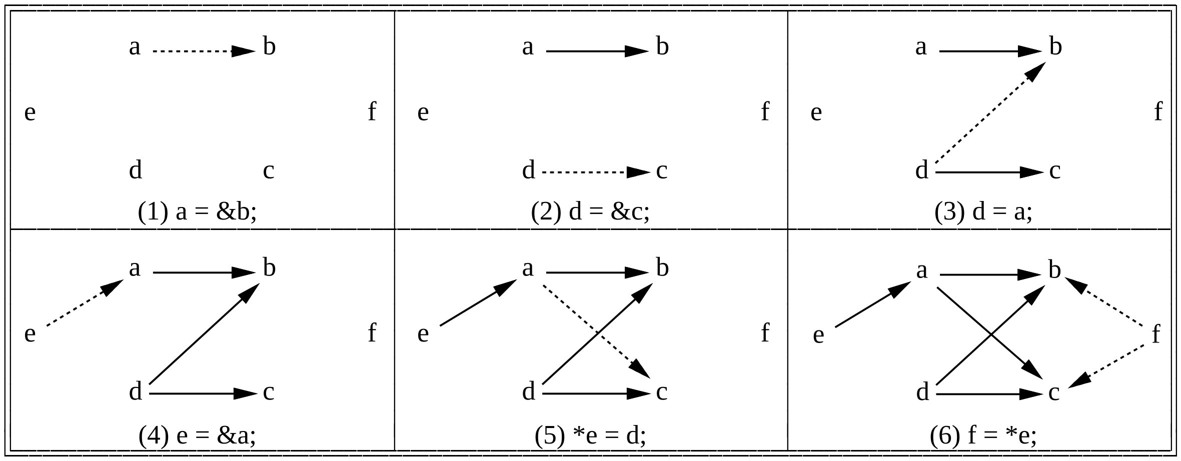
Statement	Effect on points-to graph	Corresponding Horn-clause rule
p = &q;		pointsTo(P,Q) :– assignAddr(P,Q).
p = q;		pointsTo(P,R) :– assign(P,Q), pointsTo(Q,R).
p = *q;		pointsTo(P,S) :– assignStar(P,Q), pointsTo(Q,R), pointsTo(R,S).
*p = q;		pointsTo(R,S) :– starAssign(P,Q), pointsTo(P,R), pointsTo(Q,S).
(b)		(c)

Fig. 8. Points-to analysis expressed as a Datalog program: (a) Examples of base facts generated for each of the four statement kinds. (b) Examples illustrating how each kind of statement causes the points-to graph to be modified in the Shapiro-Horwitz formulation of Andersen's algorithm [89]. In each case, the dotted edges are added to the points-to graph. (c) Horn-clause rules that express, in the form of a Datalog program, an algorithm equivalent to the Shapiro-Horwitz formulation of Andersen's algorithm.

```
a = &b;
d = &c;
d = a;
e = &a;
*e = d;
f = *e;
```

In this example, it happens that when the statements are processed in the order listed above, they only need to be considered once for points-to analysis to reach quiescence. When the statements are processed in this order, the points-to graph is built up as follows:



In each case, the dotted edge (or edges) indicates what is added to the points-to graph when we process the statement shown at the bottom of the box. Graph (6) is the final points-to graph; for each statement in the program, reapplying the corresponding rule fails to add any additional edges to the points-to graph. □

Our goal is now to show that this formulation of the problem can be reformulated as a CFL-reachability problem. To do this, we will show that the problem can be reformulated in yet two additional ways—first as a Datalog program, and then as a chain program.

The first step is easy: Figures 8(a) and 8(c) show how the Shapiro-Horwitz formulation of Andersen's algorithm can be expressed as a Datalog program. Figure 8(a) shows examples of base facts generated for each of the four statement kinds. One such fact would be generated for each assignment statement in the program. Figure 8(c) shows how the rules used for adding edges to the points-to graph in the Shapiro-Horwitz formulation can be expressed as Horn clauses.

The second step—to show that this can be expressed as a chain program—is not quite so easy.[4] As we see from Figure 8(c), the first three Horn clauses are all chain rules. Unfortunately, the fourth rule does not have the required form:

$$\text{pointsTo}(R,S) :\!-\ \text{starAssign}(P,Q),\ \text{pointsTo}(P,R),\ \text{pointsTo}(Q,S). \qquad (4.3)$$

To recast the program as a chain program, we start by rewriting the fourth rule as follows:

$$\text{pointsTo}(R,S) :\!-\ \text{pointsTo}(P,R),\ \text{starAssign}(P,Q),\ \text{pointsTo}(Q,S). \qquad (4.4)$$

This has nearly the form required, except that the order of the variables in the first literal are reversed: R follows P, whereas to have a chain rule R has to precede P.

The way to finesse this issue is to introduce a new relation, "$\overline{\text{pointsTo}}$", in which we maintain the reversal of the pointsTo relation. (This corresponds to maintaining every edge of the Shapiro-Horwitz points-to graph in both directions, one with the label "pointsTo" and one with the label "$\overline{\text{pointsTo}}$".) This allows us to rewrite rule (4.4) as the following chain rule:

$$\text{pointsTo}(R,S) :\!-\ \overline{\text{pointsTo}}(R,P),\ \text{starAssign}(P,Q),\ \text{pointsTo}(Q,S). \qquad (4.5)$$

The question is now how to build up the appropriate set of $\overline{\text{pointsTo}}$ tuples. In Datalog, this can be done by introducing the following rule:

$$\overline{\text{pointsTo}}(Q,P) :\!-\ \text{pointsTo}(P,Q). \qquad (4.6)$$

However, this is unsatisfactory for our purposes because rule (4.6) is not itself a chain rule. The way to overcome this difficulty is shown in Figure 9: Figure 9(a) shows the revised set of base facts generated for

[4]The construction we give for this step is due to David Melski [65].

Statement		Facts generated
p = &q;	⇒	assignAddr(p,q). $\overline{\text{assignAddr}}$(q,p).
p = q;	⇒	assign(p,q). $\overline{\text{assign}}$(q,p).
p = *q;	⇒	assignStar(p,q). $\overline{\text{assignStar}}$(q,p).
*p = q;	⇒	starAssign(p,q). $\overline{\text{starAssign}}$(q,p).
	(a)	

Form of statement	Chain rules for inferring pointsTo tuples and $\overline{\text{pointsTo}}$ tuples
P = &Q;	pointsTo(P,Q) :– assignAddr(P,Q). $\overline{\text{pointsTo}}$(Q,P) :– $\overline{\text{assignAddr}}$(Q,P).
P = Q;	pointsTo(P,R) :– assign(P,Q), pointsTo(Q,R). $\overline{\text{pointsTo}}$(R,P) :– $\overline{\text{pointsTo}}$(R,Q), $\overline{\text{assign}}$(Q,P).
P = *Q;	pointsTo(P,S) :– assignStar(P,Q), pointsTo(Q,R), pointsTo(R,S). $\overline{\text{pointsTo}}$(S,P) :– $\overline{\text{pointsTo}}$(S,R), $\overline{\text{pointsTo}}$(R,Q), $\overline{\text{assignStar}}$(Q,P).
*P = Q;	pointsTo(R,S) :– $\overline{\text{pointsTo}}$(R,P), starAssign(P,Q), pointsTo(Q,S). $\overline{\text{pointsTo}}$(S,R) :– $\overline{\text{pointsTo}}$(S,Q), $\overline{\text{starAssign}}$(Q,P), pointsTo(P,R).
	(b)

Fig. 9. Points-to analysis expressed as a chain program: (a) Revised set of base facts generated for each of the four statement kinds. (b) Chain rules that express an algorithm equivalent to the Shapiro-Horwitz formulation of Andersen's algorithm.

each of the four statement kinds; here we have introduced four new "reversed" base relations: $\overline{\text{assignAddr}}$, $\overline{\text{assign}}$, $\overline{\text{assignStar}}$, and $\overline{\text{starAssign}}$. Figure 9(b) shows the eight chain rules used for inferring pointsTo and $\overline{\text{pointsTo}}$ tuples. The first six rules in Figure 9(b) are the first three rules from Figure 8(c) and their respective reversals:

- In addition to the rule

  pointsTo(P,Q) :– assignAddr(P,Q).

  we have the rule

  $\overline{\text{pointsTo}}$(Q,P) :– $\overline{\text{assignAddr}}$(Q,P).

- In addition to the rule

pointsTo(P,R) :– assign(P,Q), pointsTo(Q,R).

we have the rule

$\overline{\text{pointsTo}}$(R,P) :– $\overline{\text{pointsTo}}$(R,Q), $\overline{\text{assign}}$(Q,P).

Notice that in this rule (i) the order of the literals on the right-hand side is reversed, and (ii) the order of the variables in each literal is also reversed.

- In addition to the rule

pointsTo(P,S) :– assignStar(P,Q), pointsTo(Q,R), pointsTo(R,S).

we have the rule

$\overline{\text{pointsTo}}$(S,P) :– $\overline{\text{pointsTo}}$(S,R), $\overline{\text{pointsTo}}$(R,Q), $\overline{\text{assignStar}}$(Q,P).

Again, the order of the literals on the right-hand side is reversed, and the order of the variables in each literal is also reversed.

In all three cases, both the new rule and the old rule are chain rules.

The eighth rule of Figure 9(b) is obtained by applying the reversal process to the seventh rule (*i.e.*, rule (4.5))

pointsTo(R,S) :– $\overline{\text{pointsTo}}$(R,P), starAssign(P,Q), pointsTo(Q,S).

to obtain

$\overline{\text{pointsTo}}$(S,R) :– $\overline{\text{pointsTo}}$(S,Q), $\overline{\text{starAssign}}$(Q,P), pointsTo(P,R). (4.7)

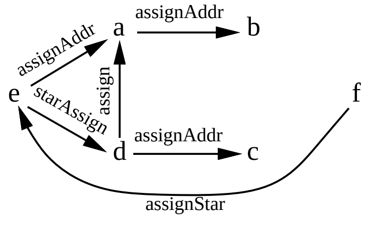
Here, not only is the order of the literals and variables reversed, but in the final literal we have substituted "pointsTo(P,R)" for "$\overline{\overline{\text{pointsTo}}}$(P,R)". Notice that rule (4.7) is a chain rule (*i.e.*, all eight rules of Figure 9(b) are chain rules).

The final step is to extract a context-free grammar from the chain program of Figure 9 by the method discussed in Section 3:

pointsTo → assignAddr	$\overline{\text{pointsTo}}$ → $\overline{\text{assignAddr}}$
pointsTo → assign pointsTo	$\overline{\text{pointsTo}}$ → $\overline{\text{pointsTo}}$ $\overline{\text{assign}}$
pointsTo → assignStar pointsTo pointsTo	$\overline{\text{pointsTo}}$ → $\overline{\text{pointsTo}}$ $\overline{\text{pointsTo}}$ $\overline{\text{assignStar}}$
pointsTo → $\overline{\text{pointsTo}}$ starAssign pointsTo	$\overline{\text{pointsTo}}$ → $\overline{\text{pointsTo}}$ $\overline{\text{starAssign}}$ pointsTo

Points-to facts can now be determined by solving L(pointsTo)-reachability problems in the graph created from the facts generated according to the table given in Figure 9(a); that is, we solve L(pointsTo)-reachability problems in a graph whose nodes correspond to program variables, and whose edges are labeled with the symbols assignAddr, $\overline{\text{assignAddr}}$, assign, $\overline{\text{assign}}$, assignStar, $\overline{\text{assignStar}}$, starAssign, and $\overline{\text{starAssign}}$.

Example 4.8. The graph corresponding to the six assignment statements of Example 4.2 is



(To avoid clutter, we have left out all reversed edges.) Using this graph and the grammar given above, we can discover, for instance, that f points to both b and c:

- Because of the paths

$$a \xrightarrow{\text{assignAddr}} b,\ d \xrightarrow{\text{assignAddr}} c, \text{ and } e \xrightarrow{\text{assignAddr}} a,$$

  there are paths

$$a \xrightarrow{\text{pointsTo}} b,\ d \xrightarrow{\text{pointsTo}} c, \text{ and } e \xrightarrow{\text{pointsTo}} a, \text{ respectively.}$$

- Because of the paths

$$b \xrightarrow{\overline{\text{assignAddr}}} a,\ c \xrightarrow{\overline{\text{assignAddr}}} d, \text{ and } a \xrightarrow{\overline{\text{assignAddr}}} e,$$

  there are paths

$$b \xrightarrow{\overline{\text{pointsTo}}} a,\ c \xrightarrow{\overline{\text{pointsTo}}} d, \text{ and } a \xrightarrow{\overline{\text{pointsTo}}} e, \text{ respectively.}$$

- There is a path $f \xrightarrow{\text{pointsTo}} b$ because of the following path:

$$f \xrightarrow{\text{assignStar}} e \xrightarrow{\text{pointsTo}} a \xrightarrow{\text{pointsTo}} b.$$

- There is a path $a \xrightarrow{\text{pointsTo}} c$ because of the following path:

$$a \xrightarrow{\overline{\text{pointsTo}}} e \xrightarrow{\text{starAssign}} d \xrightarrow{\text{pointsTo}} c,$$

  and hence there is a path $f \xrightarrow{\text{pointsTo}} c$ because of the following path:

$$f \xrightarrow{\text{assignStar}} e \xrightarrow{\text{pointsTo}} a \xrightarrow{\text{pointsTo}} c.$$

  □

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 [11,102,78,30,49,88]. 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 [15]. 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 [62]. 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 [8]. 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 CFL-reachability problems, demand algorithms are obtained for free, by solving single-source, single-target, multi-source, or multi-target CFL-reachability problems. For instance, the problem transformation described in Section 4.1 has been used to devise demand algorithms for interprocedural dataflow analysis [76,49,48]. Because an algorithm for solving, for example, 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, which were discussed in Section 4.1, to answer a single demand we need to solve a single-source/single-target $L(\mathit{realizable})$-path problem: "Is there a realizable path in $G^{\#}$ from node $\langle \mathit{start}_{\mathit{main}}, \Lambda \rangle$ to node $\langle n, d \rangle$?" For an exhaustive algorithm, we need to solve a single-source $L(\mathit{realizable})$-path problem: "What is the set of nodes $\langle n, d \rangle$ such that there is a realizable path in $G^{\#}$ from $\langle \mathit{start}_{\mathit{main}}, \Lambda \rangle$ 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. However, 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 $L(\mathit{realizable})$-path problems) runs faster than an exhaustive algorithm (*i.e.*, for the single-source $L(\mathit{realizable})$-path problem) [49,48].

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 [46,77]. 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 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 nodes in any procedure's PDG. There are no more than $\mathit{CallSites}\ \mathit{Params}^2$ summary edges, and the task of identifying all summary edges can be performed in time $O((P \times E \times \mathit{Params}) + (\mathit{CallSites} \times \mathit{Params}^3))$ [77]. 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:

$$\begin{array}{rcl}
\mathit{unbalanced\text{-}right}' & \rightarrow & \mathit{unbalanced\text{-}right}'\ \mathit{summary} \\
& \mid & \mathit{unbalanced\text{-}right}'\ e \\
& \mid & \mathit{unbalanced\text{-}right}'\ )_i \quad 1 \le i \le \mathit{CallSites} \\
& \mid & \varepsilon
\end{array}
\qquad
\begin{array}{rcl}
\mathit{realizable}' & \rightarrow & \mathit{summary}\ \mathit{realizable}' \\
& \mid & e\ \mathit{realizable}' \\
& \mid & (_i\ \mathit{realizable}' \quad 1 \le i \le \mathit{CallSites} \\
& \mid & \varepsilon
\end{array}$$

Suppose we wish to find the backward slice with respect to SDG node n. First, we solve the single-target $L(\mathit{realizable}')$-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 $L(\mathit{unbalanced\text{-}right}')$-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 [78,88].

One way to generalize the CFL-reachability approach stems from the observation that CFL-reachability problems correspond to chain programs, which are a restricted class of Datalog programs. 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 [78].

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 [88]. 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. This class of dataflow problems has been called the *I*nterprocedural *D*istributive *E*nvironment problems (or IDE problems, for short).

Two of the dataflow-analysis problems that the IDE 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 4.1, 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") [88]. 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. 10 shows the exploded representations of four environment-transformer functions used in constant propagation. Fig. 10(a) shows how the identity function $\lambda env.env$ is represented. Figs. 10(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 dataflow 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 4.1, 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

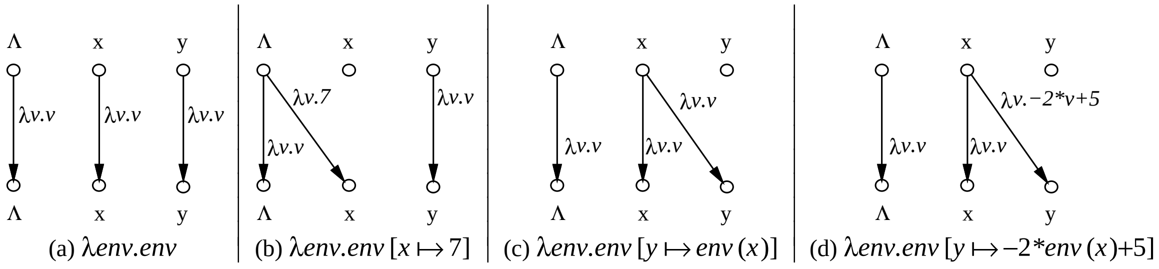


Fig. 10. The exploded representations of four environment-transformer functions used in constant propagation.

particular program point [88]. 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 [37], 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.

7. Related Work

This paper describes how a number of program-analysis problems can be solved by transforming them to CFL-reachability problems. This section discusses how the CFL-reachability approach relates to previous work on program analysis.

7.1. The Use of Graph Reachability for Analyzing Programs

A variety of work exists that has applied graph reachability (of various forms) to analysis of imperative programs. Kou [57] and Hecht [38] gave linear-time graph-reachability algorithms for solving intraprocedural "bit-vector" dataflow-analysis problems. This approach was later applied to intraprocedural bidirectional bit-vector problems by Khedker and Dhamdhere [53]. Cooper and Kennedy used reachability to give efficient algorithms for interprocedural side-effect analysis [23] and alias analysis [24].

The first uses of CFL-reachability for program analysis were in 1988, in Callahan's work on flow-sensitive side-effect analysis [20] and Horwitz, Reps, and Binkley's work on interprocedural slicing [43,46]. In both cases, only limited forms of CFL-reachability are employed, namely various kinds of matched-parenthesis (Dyck) languages, and neither paper relates the work to the more general concept of CFL-reachability. (Dyck languages had been used in earlier work on interprocedural dataflow analysis by Sharir and Pnueli to specify that the contributions of certain kinds of infeasible execution paths should be filtered out [90]; however, the dataflow-analysis algorithms given by Sharir and Pnueli are based on machinery other than pure graph reachability.)

Dyck-language reachability was shown by Reps, Sagiv, and Horwitz to be of utility for a wide variety of interprocedural program-analysis problems [76]. These ideas were elaborated on in a sequence of papers [77,81,49], and also applied to the shape-analysis problem [80].

All of these papers use only very limited forms of CFL-reachability, namely variations on Dyck-language reachability. Although the author became aware of the connection to the more general concept of CFL-reachability sometime in the fall of 1994, of the papers mentioned above, only [80] mentions CFL-reachability explicitly and references Yannakakis's paper [100]. The construction of Melski's presented in Section 4.4, as well as the constructions of Melski and Reps for converting set-constraint problems to CFL-reachability problems [66,67] show that CFL-reachability using path languages other than Dyck languages is also of utility for program analysis.

Both the IFDS framework for interprocedural dataflow analysis [76,81,49,48] as well as the IDE framework for interprocedural dataflow analysis [88], which are summarized in Sections 4.1 and 6, respectively, are related to earlier interprocedural dataflow-analysis frameworks defined by Sharir and Pnueli [90] and Knoop and Steffen [55]. The IFDS and IDE frameworks are both basically variants of the Sharir-Pnueli framework with three modifications:

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

Conditions (i) and (ii) are restrictions that make the IFDS and IDE frameworks 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 [55].)

IFDS and IDE problems can be solved by a number of previous algorithms, including the "elimination", "iterative", and "call-strings" algorithms given by Sharir and Pnueli, and the algorithm of Cousot and

Cousot [26]. However, for general IFDS and IDE problems, both the iterative and call-strings algorithms can take exponential time in the worst case. Knoop and Steffen [55] give an algorithm similar to Sharir and Pnueli's "elimination" algorithm. The efficiencies of the Sharir-Pnueli and Knoop-Steffen elimination algorithms depend, among other things, on the way functions are represented. No representations are discussed in [90] and [55]; however, because the Sharir-Pnueli and Knoop-Steffen algorithms manipulate functions as a whole, rather than pointwise, they are not as efficient as the algorithms presented in [81] and [88].

Recently, Ramalingam has shown how a framework very similar to the IDE framework can be used to develop a theory of "dataflow frequency analysis", in which information is obtained about how often and with what probability a dataflow fact holds true during program execution [75].

Holley and Rosen investigated "qualified" dataflow analysis problems, where "qualifications" are a device to specify that only certain paths in the flow graph are to be considered [42]. They employ an "expansion" phase that has some similarities to the creation of the exploded supergraph described in Section 4.1. However, Holley and Rosen do not take advantage of distributivity to do the expansion pointwise. Furthermore, for interprocedural problems the Holley-Rosen approach is equivalent to the Sharir-Pnueli call-strings approach.

Until very recently, work on demand-driven dataflow analysis only considered the *intra*procedural case [11,102,29]. Besides the work based on CFL-reachability discussed in Section 5, other work on demand-driven *inter*procedural dataflow analysis includes [79,78,30,88].

The fact that demand algorithms are obtained for free for any program-analysis problem expressed as a CFL-reachability problem, means that demand algorithms also exist for the shape-analysis and points-to-analysis problems described in Section 4.3 and 4.4, respectively, as well as for certain classes of set-constraint problems (see Section 7.3).

7.2. The Use of Logic Programming for Analyzing Programs

The use of logic programming for performing interprocedural dataflow analysis—with particular emphasis on obtaining demand algorithms—was described by Reps [79,78]. He presented a way in which algorithms that solve demand versions of interprocedural analysis problems can be obtained automatically from their exhaustive counterparts (expressed as logic programs) by making use of the "magic-sets transformation", a general transformation developed in the logic-programming and deductive-database communities for creating efficient demand versions of (bottom-up) logic programs [86,13,17,94]. Reps illustrated this approach by showing how to obtain a demand algorithm for the interprocedural "locally separable problems"—the interprocedural versions of classical "bit-vector" or "gen-kill" problems (*e.g.*, reaching definitions, available expressions, live variables, *etc.*).

The work by D.S. Warren and others concerning the use of tabulation techniques in top-down evaluation of logic programs [96] provides an alternative method for obtaining demand algorithms for such program-analysis problems. Rather than applying the magic-sets transformation to a Horn-clause encoding of the (exhaustive) dataflow-analysis algorithm and then using a bottom-up evaluator, the original (untransformed) Horn-clause encoding can simply be evaluated by an OLDT (top-down, tabulating) evaluator. Thus, another way to obtain implementations of demand algorithms for interprocedural dataflow analysis, interprocedural slicing, and other CFL-reachability problems is to make use of the SUNY-Stony Brook XSB system [97].

The algorithms described in [81] and [88] for the IFDS and IDE frameworks, respectively, have straightforward implementations as logic programs. Thus, demand algorithms for these frameworks can be obtained either by applying the magic-sets transformation or by using a tabulating top-down evaluator.

There has been some previous work in which *intra*procedural dataflow-analysis problems have been expressed using Horn clauses. For instance, one of the examples in Ullman's book shows how a logic database can be used to solve the intraprocedural reaching-definitions problem [94, pp. 984-987]. Assmann has examined a variety of other intraprocedural program-analysis problems [10]. Although Assmann expresses these problems using a certain kind of graph grammar, he points out that this formalism is equivalent to Datalog.

7.3. Relationship of CFL-Reachability Problems to Set-Constraint Problems

Following earlier work by Reynolds [85] and Jones and Muchnick [50], a number of people in recent years have explored the use of set constraints for analyzing programs. Set constraints are typically used to collect a superset of the set of values that the program's variables may hold during execution. Typically, a set variable is created for each program variable at each program point; set constraints are generated that approximate the program's behavior; program analysis then becomes a problem of finding the least solution of the set-constraint problem. Set constraints have been used both for program analysis [85,50,5,40,41], and for type inference [6,7].

In some papers in the literature, the value of converting set-constraint problems to graph-reachability problems has been exploited, both from the conceptual standpoint [70], as well as from the implementation standpoint [31].

Melski and Reps have obtained a number of results on the relationship between set constraints and graph reachability [66]. Their results establish a relationship in both directions: They showed that CFL-reachability problems and a subclass of what have been called *definite set constraints* [39] are equivalent. That is, given a CFL-reachability problem, it is possible to construct a set-constraint problem whose answer gives the solution to the CFL-reachability problem; likewise, given a set-constraint problem, it is possible to construct a CFL-reachability problem whose answer gives the solution to the set-constraint problem. In [67], these results have been extended to cover a class of set constraints that is known to be useful for analyzing programs written in higher-order languages [40].

There are several benefits that are gained from knowing that CFL-reachability problems and certain classes of set-constraint problems are interconvertible:

- There is an advantage from the conceptual standpoint: When confronted with a program-analysis problem, one can think and reason in terms of whichever paradigm is most appropriate. (This is analogous to the situation one has in formal-language theory with finite-state automata and regular expressions, or with pushdown automata and context-free grammars.) For example, as we have seen in Sections 4.1 and 4.2, CFL-reachability leads to natural formulations of interprocedural dataflow analysis and interprocedural slicing. Set constraints lead to natural formulations of shape analysis [85,50]. Each of these problems could be formulated using the (respective) opposite formalism, but it may be awkward.
- As discussed in Section 1, the interconvertibility of the two formalisms offers some insight into the "$O(n^3)$ bottleneck" for various program-analysis problems.
- The interconvertibility of the two formalisms allows results previously obtained for CFL-reachability problems to be applied to set-constraint problems, and vice versa:
  - CFL-reachability is known to be log-space complete for polynomial time (or "PTIME-complete") [1,93,82]. Because the CFL-reachability-to-set-constraint construction can be performed in log-space, the set-constraint problems considered in [66] are also PTIME-complete. Because PTIME-complete problems are believed not to be efficiently parallelizable (*i.e.*, cannot be solved in polylog time on a polynomial number of processors), this extends the class of program-analysis problems that are unlikely to have efficient parallel algorithms.
  - As discussed in Section 5, a demand algorithm computes a partial solution to a problem, when only part of the full answer is needed. Because CFL-reachability problems can be solved in a demand-driven fashion, Melski and Reps's results show that, in principle, set-constraint problems can also be solved in a demand-driven fashion. To the best of our knowledge, this had not been previously investigated in the literature on set constraints.
  - Set constraints have been used for analyzing programs written in higher-order languages, whereas applications of CFL-reachability to program analysis have been limited to first-order languages. However, recent work on the interconvertibility of CFL-reachability problems and the class of set constraints used in [40] for analyzing programs written in higher-order languages, shows that the CFL-reachability framework is capable of expressing analysis problems, such as program slicing and shape analysis, for higher-order languages [67]. In particular, one of the constructions given in [67] shows how CFL-reachability can be applied to the problem of slicing programs written in a higher-order language that manipulates heap-allocated data structures (but does not permit destructive updating of fields). This problem had not been previously addressed in the literature on program slicing.

7.4. Other Applications of CFL-Reachability

Yannakakis surveys the literature up to 1990 on applications of graph-theoretic methods in database theory [100]. He discusses many types of graph-reachability problems, including CFL-reachability.

Dolev, Even, and Karp used CFL-reachability to devise a formal model for studying the vulnerability to intrusion by a third party of a class of two-party ("ping-pong") protocols in distributed systems to intrusion by a third party [28]. Although messages in the system are protected by public-key encryption, in the setting studied by Dolev, Even, and Karp, the intruder

> . . . may be a legitimate user in the network. He can intercept and alter messages, impersonate other users or initiate instances of the protocol between himself and other users, in order to use their responses. It is possible that through such complex manipulations he can read messages, which are supposed to be protected, without cracking the cryptographic systems in use.

Dolev, Even, and Karp reduce the security-validation problem to a single-source/single-target CFL-reachability problem in which labeled edges represent possible operations, and the context-free language captures natural laws of cancellation between pairs of actions that can take place during the protocol: encryption via user *X*'s encryption function cancels with decryption via *X*'s decryption function; decryption via *X*'s decryption function cancels with encryption via *X*'s encryption function; the action of sender *X* appending his name (*i.e.*, *X*) to a message cancels with the action of a recipient stripping off the name appended to a message and comparing the name with *X*; *etc.* A protocol is shown to be insecure if and only if the graph contains a path whose word is in a certain language—namely, the language whose words correspond to sequences of actions that an intruder could provoke that would reveal the contents of the unencrypted message.

Horwitz, Reps, and Binkley pointed out a correspondence between the call structure of a program and a context-free grammar, and between the *intra*procedural transitive dependences among a PDG's parameter nodes and the dependences among attributes in an attribute grammar [46]. They exploited this correspondence to compute the summary edges used in the hybrid (*i.e.*, partly exhaustive, partly demand) scheme for interprocedural slicing discussed in Section 5. More generally, the computation of IO graphs of attribute grammars [52,27,69] and other similar approximations to the characteristic graphs of an attribute grammar's nonterminals that can be computed in polynomial time, such as TDS graphs of ordered attribute grammars [51] can be expressed as CFL-reachability problems.

CFL-reachability also provides useful algorithms for a number of problems in formal-language theory, including the following:

- Determining whether the regular language resulting from the intersection of a regular language and a context-free language is empty. (CFL-reachability is particularly useful if the regular language is given as a finite-state automaton, and the context-free language is given as a context-free grammar.)
- Solving the recognition problem for two-way pushdown automata, which has applications to certain string-matching problems [56,3,4]. A two-way pushdown automaton is a pushdown automaton in which the head reading the input tape is permitted to move in both directions (*i.e.*, in the course of a single transition of the machine, the input head may remain where it was, move one square to the right, or move one square to the left). A two-way pushdown automaton can either be deterministic (2DPDA) or non-deterministic (2NPDA). The problem of 2DPDA-recognition (resp., 2NPDA-recognition) is: Given a 2DPDA (resp., 2NPDA) M and a string ω, determine whether $\omega \in L(M)$.
- Determining whether an input-free (deterministic or non-deterministic) pushdown automaton accepts.

Acknowledgements

This paper is based, in part, on joint work with S. Horwitz, M. Sagiv, G. Rosay, and D. Melski. Section 4.4 is based on a heretofore unpublished result by D. Melski. J. Kleinberg pointed out the use of CFL-reachability in reference [28].

References

1. Afrati, F. and Papadimitriou, C. H., "The parallel complexity of simple chain queries," pp. 210-214 in *Proc. of the Sixth ACM Symp. on Princ. of Database Syst.,* (San Diego, CA, Mar. 1987), (1987).
2. Agrawal, H., "On slicing programs with jump statements," *Proc. of the ACM SIGPLAN 94 Conf. on Prog. Lang. Design and Implementation,* (Orlando, FL, June 22-24, 1994), *SIGPLAN Not.* **29**(6) pp. 302-312 (June 1994).
3. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA (1974).
4. Aho, A.V., "Algorithms for finding patterns in strings," pp. 255-300 in *Handbook of Theor. Comp. Sci., Vol. A: Algorithms and Complexity*, ed. J. van Leeuwen,The M.I.T. Press, Cambridge, MA (1990).
5. Aiken, A. and Murphy, B.R., "Implementing regular tree expressions," pp. 427-447 in *Func. Prog. and Comp. Arch., Fifth ACM Conf.,* (Cambridge, MA, Aug. 26-30, 1991)*, Lec. Notes in Comp. Sci.,* Vol. 523, ed. J. Hughes,Springer-Verlag, New York, NY (1991).
6. Aiken, A. and Murphy, B.R., "Static type inference in a dynamically typed language," pp. 279-290 in *Conf. Rec. of the Eighteenth ACM Symp. on Princ. of Prog. Lang.,* (Orlando, FL, Jan. 1991), ACM, New York, NY (1991).

7. Aiken, A. and Wimmers, E.L., “Type inclusion constraints and type inference,” pp. 31-41 in *Sixth Conf. on Func. Prog. and Comp. Arch.,* (Copenhagen, Denmark), (June 1993).

8. Allen, R., Baumgartner, D., Kennedy, K., and Porterfield, A., “PTOOL: A semi-automatic parallel programming assistant,” pp. 164-170 in *Proc. of the 1986 Int. Conf. on Parallel Processing*, IEEE Comp. Soc. Press, Wash., DC (1986).

9. Andersen, L.O., “Program analysis and specialization for the C programming language,” Ph.D. diss. and Report TOPPS D-203, Datalogisk Institut, Univ. of Copenhagen, Copenhagen, Denmark (May 1994).

10. Assmann, U., “On edge addition rewrite systems and their relevance to program analysis,” in *Proc. of the 5th Workshop on Graph Grammars and their Application to Comp. Sci.*, (Williamsburg, VA, Nov. 1994), *Lec. Notes in Comp. Sci.,* Vol. 1073, ed. J. Cuny,Springer-Verlag, New York, NY (1995).

11. 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).

12. Ball, T. and Horwitz, S., “Slicing programs with arbitrary control flow,” pp. 206-222 in *Proc. of the First Int. Workshop on Automated and Algorithmic Debugging,* (Linköping, Sweden, May 1993), *Lec. Notes in Comp. Sci.*, Vol. 749, Springer-Verlag, New York, NY (1993).

13. 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).

14. Bannerjee, U., “Speedup of ordinary programs,” Ph.D. diss. and Tech. Rep. R-79-989, Dept. of Comp. Sci., Univ. of Illinois, Urbana, IL (Oct. 1979).

15. 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).

16. Bates, S. and Horwitz, S., “Incremental program testing using program dependence graphs,” pp. 384-396 in *Conf. Rec. of the Twentieth ACM Symp. on Princ. of Prog. Lang.,* (Charleston, SC, Jan. 10-13, 1993), ACM, New York, NY (1993).

17. 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).

18. Binkley, D., “Using semantic differencing to reduce the cost of regression testing,” pp. 41-50 in *Proc. of the IEEE Conf. on Softw. Maint.* (Orlando, FL, Nov. 9-12, 1992), IEEE Comp. Soc., Wash., DC (1992).

19. Binkley, D. and Gallagher, K., “Program slicing,” in *Advances in Computers,* Vol. 43, ed. M. Zelkowitz,Academic Press, San Diego, CA (1996).

20. 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).

21. Chase, D.R., Wegman, M., and Zadeck, F.K., “Analysis of pointers and structures,” *Proc. of the ACM SIGPLAN 90 Conf. on Prog. Lang. Design and Implementation,* (White Plains, NY, June 20-22, 1990), *SIGPLAN Not.* **25**(6) pp. 296-310 (June 1990).

22. Choi, J.-D. and Ferrante, J., “Static slicing in the presence of GOTO statements,” *ACM Trans. Program. Lang. Syst.* **16**(4) pp. 1097-1113 (July 1994).

23. Cooper, K.D. and Kennedy, K., “Interprocedural side-effect analysis in linear time,” *Proc. of the ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *SIGPLAN Not.* **23**(7) pp. 57-66 (July 1988).

24. Cooper, K.D. and Kennedy, K., “Fast interprocedural alias analysis,” pp. 49-59 in *Conf. Rec. of the Sixteenth ACM Symp. on Princ. of Prog. Lang.,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).

25. 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).

26. Cousot, P. and Cousot, R., “Static determination of dynamic properties of recursive procedures,” pp. 237-277 in *Formal Descriptions of Programming Concepts,* (IFIP WG 2.2, St. Andrews, Canada, Aug. 1977), ed. E.J. Neuhold,North-Holland, New York, NY (1978).

27. Deransart, P., Jourdan, M., and Lorho, B., *Attribute Grammars: Definitions, Systems and Bibliography, Lec. Notes in Comp. Sci.,* Vol. 323, Springer-Verlag, New York, NY (1988).

28. Dolev, D, Even, S., and Karp, R.M., “On the security of ping-pong protocols,” *Information and Control* **55**(1-3) pp. 57-68 (1982).

29. Duesterwald, E., Gupta, R., and Soffa, M.L., “Demand-driven program analysis,” Tech. Rep. TR-93-15, Dept. of Comp. Sci., Univ. of Pittsburgh, Pittsburgh, PA (Oct. 1993).

30. 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).

31. Fähndrich, M., Foster, J.S., Su, Z., and Aiken, A., “Partial online cycle elimination in inclusion constraint graphs,” *Proc. of the ACM SIGPLAN 98 Conf. on Prog. Lang. Design and Implementation,* (Montreal, Canada, June 17-19, 1998), pp. 85-96 ACM Press, (1998).

32. 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).

33. Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler,* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).

34. Gallagher, K.B. and Lyle, J.R., “Using program slicing in software maintenance,” *IEEE Trans. on Softw. Eng.* **SE-17**(8) pp. 751-761 (Aug. 1991).

35. 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).

36. Goff, G., Kennedy, K., and Tseng, C.-W., “Practical dependence testing,” *Proc. of the ACM SIGPLAN 91 Conf. on Prog. Lang. Design and Implementation,* (Toronto, Ontario, June 26-28, 1991), *SIGPLAN Not.* **26**(6) pp. 15-29 (June 1991).

37. 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).

38. Hecht, M.S., *Flow Analysis of Computer Programs,* North-Holland, New York, NY (1977).

39. Heintze, N. and Jaffar, J., “A decision procedure for a class of set constraints,” Tech. Rep. CMU-CS-91-110, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA (1991).

40. Heintze, N., “Set-based analysis of ML programs,” Tech. Rep. CMU-CS-93-193, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA (July 1993).

41. Heintze, N. and Jaffar, J., “Set constraints and set-based analysis,” in *2nd Workshop on Principles and Practice of Constraint Programming,* (May 1994).

42. Holley, L.H. and Rosen, B.K., “Qualified data flow problems,” *IEEE Trans. on Softw. Eng.* **SE-7**(1) pp. 60-78 (Jan. 1981).

43. Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *Proc. of the ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *SIGPLAN Not.* **23**(7) pp. 35-46 (July 1988).

44. Horwitz, S., Pfeiffer, P., and Reps, T., “Dependence analysis for pointer variables,” *Proc. of the ACM SIGPLAN 89 Conf. on Prog. Lang. Design and Implementation,* (Portland, OR, June 21-23, 1989), *SIGPLAN Not.* **24**(7) pp. 28-40 (July 1989).

45. Horwitz, S., Prins, J., and Reps, T., “Integrating non-interfering versions of programs,” *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

46. Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (Jan. 1990).

47. Horwitz, S., “Identifying the semantic and textual differences between two versions of a program,” *Proc. of the ACM SIGPLAN 90 Conf. on Prog. Lang. Design and Implementation,* (White Plains, NY, June 20-22, 1990), *SIGPLAN Not.* **25**(6) pp. 234-245 (June 1990).

48. Horwitz, S., Reps, T., and Sagiv, M., “Demand interprocedural dataflow analysis,” TR-1283, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI (Aug. 1995).

49. 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).

50. 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).

51. Kastens, U., “Ordered attribute grammars,” *Acta Inf.* **13**(3) pp. 229-256 (1980).

52. Kennedy, K. and Warren, S.K., “Automatic generation of efficient evaluators for attribute grammars,” pp. 32-49 in *Conf. Rec. of the Third ACM Symp. on Princ. of Prog. Lang.,* (Atlanta, GA, Jan. 19-21, 1976), ACM, New York, NY (1976).

53. Khedker, U.P. and Dhamdhere, D.M., “A generalized theory of bit vector data flow analysis,” *ACM Trans. Program. Lang. Syst.* **16**(5) pp. 1472-1511 (Sept. 1994).

54. 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).

55. 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).

56. Knuth, D.E., Morris, J.H., and Pratt, V.R., "Fast pattern matching in strings," *SIAM J. Computing* **6**(2) pp. 323-350 (1977).

57. Kou, L.T., "On live-dead analysis for global data flow problems," *J. ACM* **24**(3) pp. 473-483 (July 1977).

58. 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).

59. 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).

60. Larus, J.R. and Hilfinger, P.N., "Detecting conflicts between structure accesses," *Proc. of the ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *SIGPLAN Not.* **23**(7) pp. 21-34 (July 1988).

61. Lyle, J. and Weiser, M., "Experiments on slicing-based debugging tools," in *Proc. of the First Conf. on Empirical Studies of Programming,* (June 1986), Ablex Publishing Co. (1986).

62. 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).

63. Maydan, D.E., Hennessy, J.L., and Lam, M.S., "Efficient and exact data dependence analysis," *Proc. of the ACM SIGPLAN 91 Conf. on Prog. Lang. Design and Implementation,* (Toronto, Ontario, June 26-28, 1991), *SIGPLAN Not.* **26**(6) pp. 1-14 (June 1991).

64. 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).

65. Melski, D., Personal communication to T. Reps, Fall 1996.

66. 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).

67. Melski, D. and Reps, T., "Interconvertibility of set constraints and context-free language reachability," *Theor. Comp. Sci.*, (). (Accepted, pending minor revisions.)

68. 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).

69. Möncke, U. and Wilhelm, R., "Grammar flow analysis," pp. 151-186 in *Attribute Grammars, Applications and Systems,* (Int. Summer School SAGA, Prague, Czechoslovakia, June 1991), *Lec. Notes in Comp. Sci.*, Vol. 545, ed. H. Alblas and B. Melichar,Springer-Verlag, New York, NY (1991).

70. Nielson, F., Nielson, H.R., and Hankin, C., *Principles of Program Analysis: Flows and Effects,* Cambridge Univ. Press, Cambridge, UK (). (In preparation.)

71. Ning, J.Q., Engberts, A., and Kozaczynski, W., "Automated support for legacy code understanding," *Commun. ACM* **37**(5) pp. 50-57 (May 1994).

72. 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).

73. Pugh, W., "The Omega test: A fast and practical integer programming algorithm for dependence analysis," in *Supercomputing 1991*, (Nov. 1991).

74. Pugh, W. and Wonnacott, D., "Eliminating false data dependences using the Omega test," *Proc. of the ACM SIGPLAN 92 Conf. on Prog. Lang. Design and Implementation,* (San Francisco, CA, June 17-19, 1992), *SIGPLAN Not.* **27**(7) pp. 140-151 (July 1992).

75. Ramalingam, G., "Data flow frequency analysis," *Proc. of the ACM SIGPLAN 96 Conf. on Prog. Lang. Design and Implementation,* (Philadelphia, PA, May 21-24, 1996), pp. 267-277 ACM Press, (1996).

76. 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).

77. 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).

78. 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).

79. Reps, T., "Solving demand versions of interprocedural analysis problems," pp. 389-403 in *Proc. of the Fifth Int. Conf. on Comp. Construct.*, (Edinburgh, Scotland, Apr. 7-9, 1994), *Lec. Notes in Comp. Sci.*, Vol. 786, ed. P. Fritzson,Springer-Verlag, New York, NY (1994).

80. Reps, T., “Shape analysis as a generalized path problem,” pp. 1-11 in *Proc. of the ACM SIGPLAN Symp. on Part. Eval. and Sem.-Based Prog. Manip. (PEPM 95),* (La Jolla, California, June 21-23, 1995), ACM, New York, NY (1995).

81. 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).

82. Reps, T., “On the sequential nature of interprocedural program-analysis problems,” *Acta Inf.* **33** pp. 739-757 (1996).

83. Reps, T. and Turnidge, T., “Program specialization via program slicing,” pp. 409-429 in *Proc. of the Dagstuhl Seminar on Partial Evaluation,* (Schloss Dagstuhl, Wadern, Ger., Feb. 12-16, 1996), *Lec. Notes in Comp. Sci.,* Vol. 1110, ed. O. Danvy, R. Glueck, and P. Thiemann,Springer-Verlag, New York, NY (1996).

84. Reps, T., “Program analysis via graph reachability,” pp. 5-19 in *Proc. of ILPS ′97: Int. Logic Program. Symp.,* (Port Jefferson, NY, Oct. 12-17, 1997), ed. J. Maluszynski,The M.I.T. Press, Cambridge, MA (1997).

85. 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).

86. Rohmer, R., Lescoeur, R., and Kersit, J.-M., “The Alexander method, a technique for the processing of recursive axioms in deductive databases,” *New Generation Computing* **4**(3) pp. 273-285 (1986).

87. Ross, J.L. and Sagiv, M., “Building a bridge between pointer aliases and program dependences,” in *Proc. of the European Symp. on Programming,* (Lisbon, Portugal, April 2-3, 1998), (1998).

88. Sagiv, M., Reps, T., and Horwitz, S., “Precise interprocedural dataflow analysis with applications to constant propagation,” *Theor. Comp. Sci.* **167** pp. 131-170 (1996).

89. Shapiro, M. and Horwitz, S., “Fast and accurate flow-insensitive points-to analysis,” in *Conf. Rec. of the Twenty-Fourth ACM Symp. on Princ. of Prog. Lang.,* (Paris, France, Jan. 15-17, 1997), ACM, New York, NY (1997).

90. 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).

91. Steensgaard, B., “Points-to analysis in almost-linear time,” pp. 32-41 in *Conf. Rec. of the Twenty-Third ACM Symp. on Princ. of Prog. Lang.,* (St. Petersburg, FL, Jan. 22-24, 1996), ACM, New York, NY (1996).

92. Tip, F., “A survey of program slicing techniques,” *J. Program. Lang.* **3** pp. 121-181 (1995).

93. Ullman, J.D. and Van Gelder, A., “Parallel complexity of logical query programs,” pp. 438-454 in *Proc. of the Twenty-Seventh IEEE Symp. on Found. of Comp. Sci.*, IEEE Comp. Soc., Wash., DC (1986).

94. Ullman, J.D., *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies,* Computer Science Press, Rockville, MD (1989).

95. Valiant, L.G., “General context-free recognition in less than cubic time,” *J. Comp. Syst. Sci.* **10**(2) pp. 308-315 (Apr. 1975).

96. Warren, D.S., “Memoing for logic programs,” *Commun. ACM* **35**(3) pp. 93-111 (Mar. 1992).

97. Warren, D.S., “XSB Logic Programming System,” Software system, Comp. Sci. Dept., State Univ. of New York, Stony Brook, NY (1993). (Available via ftp from sbcs.sunysb.edu.)

98. Weiser, M., “Program slicing,” *IEEE Trans. on Softw. Eng.* **SE-10**(4) pp. 352-357 (July 1984).

99. Wolfe, M.J., “Optimizing supercompilers for supercomputers,” Ph.D. diss. and Tech. Rep. R-82-1105, Dept. of Comp. Sci., Univ. of Illinois, Urbana, IL (Oct. 1982).

100. Yannakakis, M., “Graph-theoretic methods in database theory,” pp. 230-242 in *Proc. of the Ninth ACM Symp. on Princ. of Database Syst.*, (1990).

101. Younger, D.H., “Recognition and parsing of context-free languages in time *n***3,” *Inf. and Cont.* **10** pp. 189-208 (1967).

102. 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).