Shape Analysis as a Generalized Path Problem

Thomas Reps
University of Wisconsin

Abstract

This paper concerns a method for approximating the possible "shapes" that heap-allocated structures in a program can take on. We present a new approach to finding solutions to shape-analysis problems that involves formulating them as generalized graph-reachability problems. The reachability problem that arises is not an ordinary reachability problem (*e.g.*, transitive closure), but one in which a path is considered to connect two vertices only if the concatenation of the labels on the edges of the path is a word in a certain context-free language. This graph-reachability approach allows us to give polynomial bounds on the running time of an algorithm for shape analysis. It also permits us to obtain a demand algorithm for shape analysis. (In a demand algorithm, the goal is to determine shape information selectively—*i.e.*, for particular variables at particular points in the program, rather than for every variable at every point in the program.)

1. Introduction

This paper concerns a method for approximating the possible "shapes" that heap-allocated structures in a program can take on. In the context of partial evaluation, variations on the method have been used for a variety of purposes, notably in Mogensen's binding-time analysis for partially static structures [18] and later in his binding-time analysis for program bifurcation [19].

The shape-analysis problem was originally formulated by Reynolds [26]. Reynolds treated the problem as one of simplifying a collection of set equations. In particular, his algorithm for solving the shape-analysis problem repeatedly applies a "reduction step" to eliminate selection operations from the set of equations.

The shape-analysis problem was formulated independently a few years later by Jones and Muchnick [14]. They treated the problem as one of solving (*i.e.*, finding the least fixed-point of) a collection of equations using regular tree grammars. They regarded each dataflow equation as a production in a kind of "extended regular tree grammar" and then showed that these grammars can be transformed into ordinary regular tree grammars. The goal of the transformation is to eliminate productions that represent applications of selection operations. The Jones-Muchnick algorithm is essentially equivalent to the algorithm given by Reynolds.

In this paper, we present a different approach to finding a solution: We formulate the shape-analysis problem as a generalized ***graph-reachability problem***. The reachability problem that arises is not an ordinary reachability problem (*e.g.*, transitive closure), but one in which a path is considered to connect two vertices only if the concatenation of the labels on the edges of the path is a word in a certain context-free language.

The technique we present is not as precise as the Reynolds and Jones-Muchnick techniques. The reason for the difference in precision is that, for a given variable at a given point in the program, we approximate the shapes the variable can take on via a *single* "shape descriptor", whereas the Reynolds and Jones-Muchnick techniques use a *collection* of "shape descriptors". Other techniques that are imprecise in the same way that ours is have been employed by Mogensen in his binding-time analysis for program bifurcation [19] and by Heintze in his work on "set-based analysis problems" [9].

Compared to previous work, the novelty of our work stems from our formulation of the shape-analysis problem as a generalized graph-reachability problem. By treating the problem in this way, we gain two important benefits:

(i) There is a general result that all "context-free-language reachability problems" can be solved in time cubic in the number of vertices in the graph [29]. Cubic bounds on the cost of shape analysis follow immediately:
   - For ***Lisp-like imperative languages*** without "rplaca" and "rplacd" (â la Jones and Muchnick's paper, but with recursive procedures), the associated graph-reachability problem can be solved in time $O(N^3 Var^3)$, where N is the number of vertices in a program's control-flow graph and Var is the maximum number of variables visible in any scope.
   - For ***Lisp-like functional languages*** (â la Reynolds's paper) the associated graph-reachability problem can be solved in time $O(Prog^3)$, where $Prog$ is the size of the expression tree that represents the program.[1]

(ii) We can obtain a ***demand algorithm*** for the shape-analysis problem. In a demand algorithm, the goal is to determine shape information selectively—*i.e.*, for particular variables at particular points in the program, rather than for every variable at every point in the program. Demand algorithms have the potential to be very useful in compilers [24,22,8]: The conventional approach is to compute *all* information for *all* program

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant CCR-9100424, and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

Author's address: Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706.
Telephone: (608) 262-1204.
Electronic mail: reps@cs.wisc.edu.

[1]For functional languages, this bound has also been achieved for "set-based analysis problems", which are a generalization of the shape-analysis problem. Set-based analysis problems involve solving a set of constraint equations, and Heintze has shown that they can be solved in time $O(Prog^3)$ [9].

```
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
```

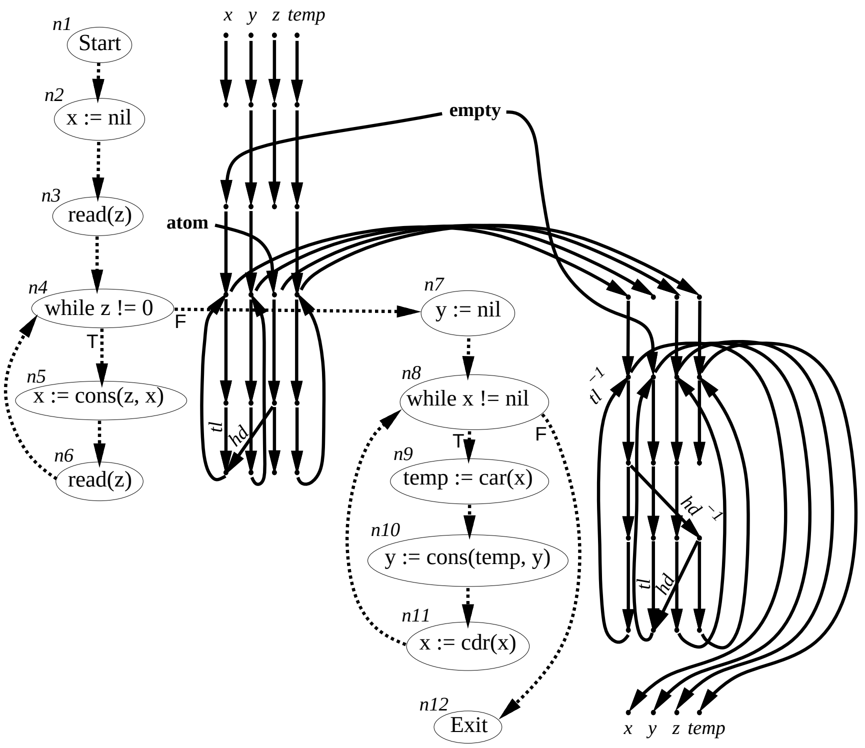

Figure 1. An example program, its control-flow graph, and its equation dependence graph. All edges of the equation dependence graph shown without labels have the label ***id***.

points; however, during optimization, information is typically needed only at program points in the innermost loops. Demand algorithms could also be useful in an interactive setting. In the context of partial evaluation, one such setting would be in a tool to aid a user who is trying to introduce binding-time improvements by selectively restructuring parts of his program.

The remainder of the paper is organized as follows: Section 2 discusses how the shape-analysis problem for imperative programs can be formulated as a generalized path problem. Section 3 briefly outlines how the shape-analysis problem for functional programs can be formulated as a generalized path problem. Section 4 describes how the so-called "Magic-sets transformation" [27,2,4,28], a transformation that was developed in the logic-programming and logic-database communities for optimizing the evaluation of recursive Horn-clause programs, can be used to obtain a demand algorithm for the shape-analysis problem. Section 5 discusses related work. (For readers not familiar with the Magic-sets transformation, the Appendix presents two examples that illustrate its capabilities.)

2. Formulating the Shape-Analysis Problem as a Path Problem (Imperative Languages)

In this section, we address shape analysis for imperative languages that support non-destructive manipulation of heap-allocated objects.

We assume we are working with an imperative language that meets the following general description: Programs consist of assignment statements, conditional statements, loops (while, for, repeat), read statements, write 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)"). This assumption is not essential, but it simplifies the presentation and also has a small effect on how we state our bound on the time to solve shape-analysis problems.

Example. An example program, taken from [14], is shown in Figure 1. 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

Form of source-vertex p	Equations associated with edge $p \rightarrow q$	
$x := a$, where a is an atom	$v_{\langle q,\, x\rangle} = \{\, \mathbf{at} \,\}$	$v_{\langle q,\, z\rangle} = v_{\langle q,\, z\rangle}$, 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)	$v_{\langle q,\, x\rangle} = \mathbf{cons}(v_{\langle p,\, y\rangle}, v_{\langle p,\, z\rangle})$	"

Figure 2. Dataflow-equation schemas for shape analysis.

the paper to illustrate our techniques. □

As in [26] and [14], a collection of dataflow equations is introduced to capture the shapes of (a superset of) the possible terms at the various points in the program. The domain of shape descriptors is similar to the domain of trees, except that all atoms are replaced by the special value **at**. Formally, the domain Shape is the set of selector sequences terminated by **at** or **nil**: Shape $= 2^{\{\, \mathbf{hd}, \mathbf{tl} \,\}^* \times \{\, \mathbf{at}, \mathbf{nil} \,\}}$. Each sequence in $\{\, \mathbf{hd}, \mathbf{tl} \,\}^* \times \{\, \mathbf{at}, \mathbf{nil} \,\}$ represents a possible root-to-leaf path. Strictly speaking, a shape descriptor s in Shape is not a tree because, for example, s could contain both the selector sequences ***hd.tl*.at** and ***hd.tl.hd*.at**, which cannot both occur in a tree.

Following Jones and Muchnick, dataflow variables correspond to $\langle$*program-point*, *program-variable*$\rangle$ 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 vertex 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 Figure 2.

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 [7]. The question, however, is: "Over what domain are they to be solved?" In the Reynolds and Jones-Muchnick approach, the value of each dataflow variable is a *set* of shapes (*i.e.*, a set of sets of root-to-leaf paths), and the join operation is union. 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 follow Mogensen [19]: The value of each dataflow variable is a *single* Shape (*i.e.*, a set of root-to-leaf paths), and the join operation is union. 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* (â la Reynolds's paper) or, equivalently, as a *regular tree grammar* that satisfies the equations (â la Jones and Muchnick's and Mogensen's papers). In Section 2.1, we show how the desired indirect "shape information" can also be viewed as *reachability information* that can be obtained by solving a generalized path problem.

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:  ···
```

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:

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

(a) Jones and Muchnick [14]

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

(b) Mogensen [19]

Grammar (a) uses multiple **cons** right-hand sides for a given nonterminal. In grammar (b), the link between branches in different **cons** alternatives is broken, and a single **cons** right-hand side is formed with the union of a set of nonterminals in each arm. The shape descriptions are sharper with grammars of type (a): With grammar (a), 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 (b) associates nonterminal $v_{\langle r,A\rangle}$ with trees of the form $\mathbf{cons}(v_{\langle p,B\rangle}, v_{\langle q,E\rangle})$. However, retaining the sharper information is achieved at some cost [19].

2.1. Shape Analysis Via Context-Free-Language Reachability

We now show how shape-analysis information can be obtained by solving a path problem in which a path is considered to connect two vertices only if the concatenation of the labels on the edges of the path is a word in a certain 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 ***context-free-language reachability problems*** (***CFL-reachability problems***) as follows:

(i) The ***all-pairs L-path problem*** (or ***L-transitive closure problem***) is to determine all pairs of vertices v_1, $v_2 \in V(G)$ such that there exists an L-path in G from v_1 to v_2.
(ii) The ***single-source L-path problem*** is to determine all vertices $v_2 \in V(G)$ such that there exists an L-path in G from a given source v_1 to v_2.
(iii) The ***single-sink L-path problem*** is to determine all vertices $v_1 \in V(G)$ such that there exists an L-path in G from v_1 to a given target v_2.
(iv) The ***source-target L-path problem*** is to determine whether there exists an L-path in G from a given source v_1 to a given target v_2. □

To determine shape-analysis information, we solve CFL-reachability problems on a graph obtained from the program's dataflow equations. In a slight abuse of terminology, we will term this the "equation dependence graph", defined as follows:

Definition 2.2. 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 vertices **atom** and **empty**, together with a vertex $\langle p,z\rangle$ for each variable $v_{\langle p,z\rangle}$ in Eqn_G. The edges of the graph, each of which is labeled with one of $\{$ ***id***, ***hd***, ***tl***, ***hd⁻¹***, ***tl⁻¹*** $\}$, are defined as shown in the following table:

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

□

Example. The equation dependence graph for the running example is shown in Figure 1. □

Shape-analysis information can be determined by solving *three* CFL-reachability problems on the equation dependence graph. We use three different context-free languages, defined by the following context-free grammars:

L_1	$id\_path \rightarrow \mathbf{hd}\ id\_path\ \mathbf{hd}^{-1}\ id\_path$ $id\_path \rightarrow \mathbf{tl}\ id\_path\ \mathbf{tl}^{-1}\ id\_path$ $id\_path \rightarrow \mathbf{id}\ id\_path$ $id\_path \rightarrow \varepsilon$
L_2	$hd\_path \rightarrow id\_path\ \mathbf{hd}\ id\_path$
L_3	$tl\_path \rightarrow id\_path\ \mathbf{tl}\ id\_path$

For both L_2 and L_3, *id_path* is the nonterminal defined in grammar L_1.

The language of L_1 represents paths in which each ***hd*** (***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 ***hd*** or ***tl*** label on an edge in the path) is eventually "taken apart" by a car ($\boldsymbol{hd}^{-1}$) or cdr ($\boldsymbol{tl}^{-1}$) operation. Thus, the first two rules of grammar L_1 are the grammar-theoretic analogs of McCarthy's rules [17]:

$$\text{car}(\text{cons}(x, y)) = x \qquad\qquad \text{cdr}(\text{cons}(x, y)) = y.$$

The language of L_2 represents paths that are slightly unbalanced—those with one unmatched ***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 of L_3 also represents paths that are slightly unbalanced—in this case, those with one unmatched ***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 2.3. Suppose we are interested in the "shape" of program-variable y just before the **exit** statement of the program shown in Figure 1. We can determine information about balanced paths to vertex $\langle n12, y\rangle$—and hence information about the possible origin of the value (*i.e.*, the root constituent) of y at *n12*—by solving the single-sink L_1-problem for $\langle n12, y\rangle$. The vertices for which there are *id_path*'s that lead to vertex $\langle n12, y\rangle$ are

$\langle n12, y\rangle$, $\langle n8, y\rangle$, $\langle n11, y\rangle$, **empty**.

This information indicates that either y is nil or it was allocated during an execution of the second while loop.

Similarly, the vertices for which there are *hd_path*'s that lead to $\langle n12, y\rangle$ are the solution to the single-sink L_2-path problem for $\langle n12, y\rangle$:

$\langle n10, temp\rangle$, $\langle n5, z\rangle$, $\langle n4, z\rangle$, **atom**.

This information indicates that the atom in car(y) is one originally read in as the value of z.

Finally, the vertices for which there are *tl_path*'s that lead to $\langle n12, y\rangle$ are the solution to the single-sink L_3-path problem:

$\langle n10, y\rangle$, $\langle n9, y\rangle$, $\langle n8, y\rangle$, $\langle n11, y\rangle$, **empty**.

This information indicates that either the tail of y is nil or it was allocated during an execution of the second while loop.

We now show how this information relates to the information reported by other shape-analysis methods. The relationship is simple: The reachability information obtained above can be interpreted as the following regular tree grammar production (of essentially the type used by Mogensen):

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

Thus, by putting together results from the L_1-, L_2-, and L_3-path problems, reachability information can be interpreted as the same type of information that is furnished by other shape-analysis methods. □

A more complete "picture" of the shapes of values in the program can be built up by solving the L_1-, L_2-, and L_3-path problems for other vertices of the equation dependence graph. For instance, to obtain information for every point in the program that is equivalent to the information gathered by Mogensen's analysis, we would solve all *three* problems for *all* of the vertices of the equation dependence graph. (The results are equivalent to Mogensen's analysis, modulo the fact that his analysis also carries along binding-time annotations.)

In some cases, it may be desirable to solve other related CFL-reachability problems. For example, the following context-free grammar allows us to determine (an approximation to) the origin of all components in the structure subordinate to a program variable at a given point in a program:

L_4	*unmatched_path* → *id_path* **hd** *unmatched_path*
	unmatched_path → *id_path* **tl** *unmatched_path*
	unmatched_path → *id_path*

The language of L_4 represents paths that are unbalanced to an unbounded degree—those with an arbitrary number of unmatched ***hd***'s and ***tl***'s.

Example 2.4. Suppose we want to find out the origin of all components in the structure subordinate to y just before the **exit** statement of the program shown in Figure 1. This can be obtained by solving the single-sink L_4-path problem for vertex $\langle n12, y\rangle$:

$\langle n12, y\rangle$, $\langle n8, y\rangle$, **empty**, $\langle n11, y\rangle$, $\langle n10, y\rangle$, $\langle n10, temp\rangle$, $\langle n9, y\rangle$, $\langle n5, z\rangle$, $\langle n4, z\rangle$, **atom**.

Among other things, this indicates that the structure subordinate to y is built up of constituents that were never the value (*i.e.*, the root constituent) of variable x anywhere in the program. □

2.2. The Cost of Solving CFL-Reachability Problems

There is a general result that all CFL-reachability problems can be solved in time cubic in the number of vertices in the graph [29]. The bound on the cost of shape analysis follows from this: Because the number of vertices in an equation dependence graph is bounded by $O(N\,Var)$, where N is the number of vertices in the program's control-flow graph and Var is the maximum number of variables visible in any scope, the shape-analysis problem can be solved in time $O(N^3 Var^3)$.

The algorithms for solving the four kinds of CFL-reachability problems are dynamic-programming algorithms. They can be thought of as generalizations of the CYK algorithm for context-free recognition [15,30]. (For instance, the recognition problem for any context-free language L can be formulated as a source-target L-path problem on a linear graph whose edges are labeled by the letters of the input string [29].)

We delay further discussion of solution methods until Section 4. There we give a Horn-clause program that, when viewed as rules to be evaluated "bottom-up", specifies a dynamic-programming algorithm for all sixteen L_1-, L_2-, L_3-, and L_4-path problems.

3. Formulating the Shape-Analysis Problem as a Path Problem (Functional Languages)

For Lisp-like functional languages, our version of the shape-analysis problem can also be formulated as a CFL-reachability problem. For these languages, the equation dependence graph is essentially the expression tree (abstract-syntax tree) that represents the program (but with child-to-parent pointers), together with edges from the bodies of called functions to call sites and from actual parameters to formal parameters.

The size of such a graph is $O(Prog)$, where $Prog$ is the number of vertices in the expression tree that represents the program. Consequently, the associated CFL-reachability problems can be solved in time $O(Prog^3)$. The context-free grammars used with the graphs that arise with functional languages are again L_1, L_2, and L_3 (and L_4, if desired).

4. A Demand Algorithm for Shape Analysis

In this section, we describe how to obtain a demand algorithm for our version of the shape-analysis problem. In a demand algorithm, the goal is to determine shape information selectively—*i.e.*, for particular variables at particular points in the program, rather than for every variable at every point in the program.

The CFL-reachability problem can be solved with a dynamic-programming algorithm. In this section, we will express such an algorithm as a Horn-clause program, specifically as a program written in the Coral language [20,21]. This approach has one very important benefit, namely that it becomes trivial to obtain a demand algorithm for the shape-analysis problem; we merely have to apply the so-called "Magic-sets transformation", which is a general transformation for creating efficient demand versions of logic programs [27,2,4,28].

Remark. The approach we are using presents us with somewhat of a challenge for communicating our results to the programming-languages community, for the following reasons:

- Many people in the programming-languages community are unfamiliar with the ***bottom-up*** approach to evaluating logic programs, and try to apply their intuitions about Prolog programs (which are evaluated using a ***top-down*** strategy) to such programs. However, there are two important differences between bottom-up and top-down evaluation strategies:
  - — A top-down (goal-directed) evaluation strategy can take exponential time on examples that a bottom-up

```
%- n1: start -> n2: x := nil
edge(v(n1,x), v(n2,x), id).
edge(v(n1,y), v(n2,y), id).
edge(v(n1,z), v(n2,z), id).
edge(v(n1,temp), v(n2,temp), id).

%- n2: x := nil -> n3: read(z)
edge(empty, v(n3,x), id).
edge(v(n2,y), v(n3,y), id).
edge(v(n2,z), v(n3,z), id).
edge(v(n2,temp), v(n3,temp), id).

%- n3: read(z) -> n4: while z != 0
edge(v(n3,x), v(n4,x), id).
edge(v(n3,y), v(n4,y), id).
edge(atom, v(n4,z), id).
edge(v(n3,temp), v(n4,temp), id).

%- n4: while z != 0 -> n5: x := cons(z,x)
edge(v(n4,x), v(n5,x), id).
edge(v(n4,y), v(n5,y), id).
edge(v(n4,z), v(n5,z), id).
edge(v(n4,temp), v(n5,temp), id).

%- n5: x := cons(z,x) -> n6: read(z)
edge(v(n5,x), v(n6,x), tl).
edge(v(n5,z), v(n6,x), hd).
edge(v(n5,y), v(n6,y), id).
edge(v(n5,z), v(n6,z), id).
edge(v(n5,temp), v(n6,temp), id).

%- n6: read(z) -> n4: while z != 0
edge(v(n6,x), v(n4,x), id).
edge(v(n6,y), v(n4,y), id).
edge(atom, v(n4,z), id).
edge(v(n6,temp), v(n4,temp), id).

%- n4: while z != 0 -> n7: y := nil
edge(v(n4,x), v(n7,x), id).
edge(v(n4,y), v(n7,y), id).
edge(v(n4,z), v(n7,z), id).
edge(v(n4,temp), v(n7,temp), id).

%- n7: y := nil -> n8: while x != nil
edge(v(n7,x), v(n8,x), id).
edge(empty, v(n8,y), id).
edge(v(n7,z), v(n8,z), id).
edge(v(n7,temp), v(n8,temp), id).

%- n8: while x != nil -> n9: temp := car(x)
edge(v(n8,x), v(n9,x), id).
edge(v(n8,y), v(n9,y), id).
edge(v(n8,z), v(n9,z), id).
edge(v(n8,temp), v(n9,temp), id).

%- n9: temp := car(x) -> n10: y := cons(temp,y)
edge(v(n9,x), v(n10,x), id).
edge(v(n9,y), v(n10,y), id).
edge(v(n9,z), v(n10,z), id).
edge(v(n9,x), v(n10,temp), hd_inv).

%- n10: y := cons(temp,y) -> n11: x := cdr(x)
edge(v(n10,x), v(n11,x), id).
edge(v(n10,y), v(n11,y), tl).
edge(v(n10,temp), v(n11,y), hd).
edge(v(n10,z), v(n11,z), id).
edge(v(n10,temp), v(n11,temp), id).

%- n11: x := cdr(x) -> n8: while x != nil
edge(v(n11,x), v(n8,x), tl_inv).
edge(v(n11,y), v(n8,y), id).
edge(v(n11,z), v(n8,z), id).
edge(v(n11,temp), v(n8,temp), id).

%- n8: while x != nil -> n12: Exit
edge(v(n8,x), v(n12,x), id).
edge(v(n8,y), v(n12,y), id).
edge(v(n8,z), v(n12,z), id).
edge(v(n8,temp), v(n12,temp), id).
```

Figure 3. The base-relation tuples that represent the equation dependence graph from Figure 1.

evaluation strategy handles in polynomial time.[2]

— For problems such as the CFL-reachability problem, a bottom-up evaluation strategy is ***complete***, whereas Prolog's top-down evaluation strategy can get trapped in an ***infinite loop***.

- Many people in the programming-languages community are unfamiliar with the Magic-sets transformation.

For these reasons, a short tutorial on bottom-up evaluation and the Magic-sets transformation is provided in the Appendix. (See also [5].)

Some programming-languages people are also leery of the (space, time, and conceptual) overheads involved in using logic databases, and question whether the logic-programming approach to obtaining demand algorithms for static-analysis problems can really produce implementations efficient enough for use in real-world program-analysis tools. Although the jury is still out on this issue (*e.g.*, waiting for improved logic-database implementations), in some sense the issue is irrelevant, for the following reasons:

- Horn clauses, when viewed as rules to be evaluated bottom-up, can be thought of as just a particularly concise notation for expressing dynamic-programming algorithms (see the Appendix and the discussion below).
- The two basic ideas used in the Magic-sets transformation are *propagation of queries* and *caching of results*, and it is fairly easy to transfer these notions over to demand algorithms written in an imperative programming language (such as C).

[2]This is similar to the situation one has with functional programs where a recursive Fibonacci program executed with a non-memoizing implementation of the language uses exponential time, whereas an iterative program that computes *fib*(0), *fib*(1), *fib*(2), . . . in bottom-up fashion takes linear time.

A successful application of this approach can be found in the author's recent work on demand algorithms for interprocedural dataflow-analysis problems, which showed how certain classes of interprocedural dataflow-analysis problems can be posed as CFL-reachability problems: In [24], the Magic-sets transformation was used to obtain demand algorithms for a class of interprocedural dataflow-analysis problems; subsequently, a demand algorithm was developed that can be implemented easily in an imperative programming language [22,12]. The latter algorithm can be viewed as an analog of the program that results from applying the Magic-sets transformation to an exhaustive dataflow-analysis algorithm written as a Horn-clause program. However, the algorithm given in [22,12] has a low-overhead implementation in an imperative programming language. The implementation is based on array indexing and linked lists, and involves neither term-unification nor term-matching.

Exactly the same approach can be used with the ideas presented in this paper. □

For a given shape-analysis problem, the equation dependence graph is represented as a base relation consisting of facts of the form

`edge(source,target,annotation)`.

(The same base relation `edge` can be used regardless of whether the problem instance stems from an imperative program or a functional program.) `Source` and `target` represent vertices of the equation dependence graph (and in our Coral implementation have further structure—*e.g.*, `v(n12,y)`), and `annotation` is one of the identifiers `id`, `hd`, `tl`, `hd_inv`, or `tl_inv`. Figure 3 shows the facts that represent the equation dependence graph from Figure 1.

The program given in Figure 4 encodes the grammars L_1, L_2, L_3, and L_4 that were discussed in Section 2. When viewed as rules to be evaluated bottom-up, this program specifies a dynamic-programming algorithm for the sixteen CFL-reachability problems involving L_1, L_2, L_3, and L_4. During bottom-up evaluation, the deduction engine applies an "immediate-consequence" operator until a fixed point is found (*i.e.*, until no new facts can be derived). (In our case, the immediate-consequence operator will derive additional "path facts"; for instance, facts indicating the presence of longer paths in the graph (that meet certain conditions) will be derived from known facts about the graph's edges and from previously acquired facts about shorter paths.) Thus, the fixed-point-finding loop of the deduction engine serves as the iterative loop of a dynamic-programming process.

The `export` declarations indicate what binding patterns for queries will be permitted. They direct the system to transform the program—via the Magic-sets transformation—to a form that is specialized for answering

```
module path.

export id_path(ff).
export id_path(bf).
export id_path(fb).
export id_path(bb).

export hd_path(ff).
export hd_path(bf).
export hd_path(fb).
export hd_path(bb).

export tl_path(ff).
export tl_path(bf).
export tl_path(fb).
export tl_path(bb).

export unmatched_path(ff).
export unmatched_path(bf).
export unmatched_path(fb).
export unmatched_path(bb).

id_path(V,Z) :- edge(V,W,hd), id_path(W,X), edge(X,Y,hd_inv), id_path(Y,Z).
id_path(V,Z) :- edge(V,W,tl), id_path(W,X), edge(X,Y,tl_inv), id_path(Y,Z).
id_path(V,Z) :- edge(V,W,id), id_path(W,Z).
id_path(W,W).

hd_path(W,Z) :- id_path(W,X), edge(X,Y,hd), id_path(Y,Z).
tl_path(W,Z) :- id_path(W,X), edge(X,Y,tl), id_path(Y,Z).

unmatched_path(W,Z) :- id_path(W,X), edge(X,Y,hd), unmatched_path(Y,Z).
unmatched_path(W,Z) :- id_path(W,X), edge(X,Y,tl), unmatched_path(Y,Z).
unmatched_path(W,Z) :- id_path(W,Z).

end_module.
```

Figure 4. A Coral program to solve shape-analysis problems.

queries in the specified binding patterns. For example, the export declaration

```
export id_path(fb).
```

permits `id_path` queries to be processed in which the first argument is free and the second argument is bound, such as

```
?id_path(A,v(n12,y)).
```

By providing all four binding patterns `ff`, `bf`, `fb`, and `bb` for each of the four relations, we can solve CFL-reachability problems of all sixteen kinds.

We do not actually show the program that results from applying the Magic-sets transformation to Figure 4. The transformed program is rather complicated and presenting it would not aid the reader's understanding.

The table shown in Figure 5 presents the results that Coral reports for four "shape-information" queries about program-variable *y* just before the **exit** statement in the program shown in Figure 1. The times reported in column three indicate the time taken to answer the queries when the Magic-sets transformation was *not* applied; the times in column four indicate the time taken by the Magic-sets-transformed version. (These tests were carried out with Release 1.1 of the Coral logic-database system [21] on a Sun SPARCstation 10 Model 30 with 32 MB of RAM.)

Remark. In practice—at least with the Coral system—to obtain the most efficient program for a given binding pattern, it is usually necessary to rewrite the recursive rules slightly and reorder the literals in non-recursive rules before applying the Magic-sets transformation. Such concerns are outside the scope of this paper, except to point out that the results reported in Figure 5 are for a variant of the rules presented in Figure 4 appropriately modified to cause the Magic-sets transformation to produce a more efficient program for `id_path`, `hd_path`, `tl_path`, and `unmatched_path` queries in which the first argument is free and the second argument is bound.

Thus, in practice, we have to provide Coral with four sets of rules for each of the relations `id_path`, `hd_path`, `tl_path`, and `unmatched_path` (*i.e.*, rules that define, say, `id_path_ff`, `id_path_bf`, `id_path_fb`, `id_path_bb`, . . .) □

5. Relation to Previous Work

The version of the shape-analysis problem studied in this paper was originally introduced by Mogensen [19]. This version is simpler but less precise than the one addressed by Reynolds and by Jones and Muchnick. Using the terminology of [13], the simpler version of the problem is an "independent attribute method" for shape analysis, whereas the Reynolds/Jones-Muchnick version of the problem has some of the flavor of a "relational method". In an independent attribute method, each dataflow variable is associated with a function that describes characteristics of the values

Query	Result	Time (user-time + system-time in seconds)	
		Exhaustive Version	Demand Version
`?id_path(A,v(n12,y)).`	`A=v(n12,y).` `A=v(n8,y).` `A=v(n11,y).` `A=empty.`	2.03u + .05s	.03u + .03s
`?hd_path(A,v(n12,y)).`	`A=v(n10,temp).` `A=v(n5,z).` `A=v(n4,z).` `A=atom.`	1.83u + .04s	.10u + .03s
`?tl_path(A,v(n12,y)).`	`A=v(n10,y).` `A=v(n9,y).` `A=v(n8,y).` `A=v(n11,y).` `A=empty.`	1.84u + .03s	.07u + .01s
`?unmatched_path(A,v(n12,y)).`	`A=v(n12,y).` `A=v(n8,y).` `A=empty.` `A=v(n11,y).` `A=v(n10,y).` `A=v(n10,temp).` `A=v(n9,y).` `A=v(n5,z).` `A=v(n4,z).` `A=atom.`	2.12u + .04s	.20u + .01s

Figure 5. Results from four "shape-information" queries about program-variable *y* just before the **exit** statement in the program shown in Figure 1.

the dataflow variable may take on. In a relational method, each program point is associated with a relation that describes relationships that must hold among a collection of dataflow variables. For example, consider the regular tree grammar production

$A \rightarrow \mathbf{cons}(B,\ C)\ \mid\ \mathbf{cons}(D,\ E).$

If we think of $A.\boldsymbol{hd}$ and $A.\boldsymbol{tl}$ as "virtual" dataflow variables, then the two right-hand-side alternatives in the production express a relationship that must hold for the tuple ($A.\boldsymbol{hd}$, $A.\boldsymbol{tl}$) (*i.e.*, a B-value can be paired with a C-value, but not with an E-value). In our version of the shape-analysis problem, such relationships are lost (*i.e.*, the C-E pairing is permitted).

The version of the shape-analysis problem addressed in the paper is related to what Heintze calls "set-based analysis" [9]. Set-based-analysis problems are a generalization of the shape-analysis problem to incorporate data other than just heap-allocated storage. For functional languages, Heintze has shown that these problems can be solved in time $O(Prog^3)$, where *Prog* is the size of the program. (It seems plausible that in the case of imperative languages, the bound would increase to $O(N^3 Var^3)$—our bound for shape-analysis—where N is the number of vertices in a program's control-flow graph and *Var* is the maximum number of variables visible in any scope.)

The grammars L_1, L_2, and L_3 (for nonterminals *id_path*, *hd_path*, and *tl_path*, respectively) take the place of the "reduction step" that Reynolds uses for eliminating selections [26]. These grammars are also similar to the relation that Jones and Muchnick define in order to eliminate productions with selectors [14, pp. 111-113]. Our approach is not as precise as theirs; however, by formulating shape analysis as a CFL-reachability problem, we gain three benefits:

- A bound on the amount of work performed falls out immediately from the fact that all CFL-reachability problems can be solved in time cubic in the number of vertices in the graph.
- We are able to obtain a demand algorithm for the problem automatically by encoding the L_1-, L_2-, and L_3-path problems as Horn-clause programs and applying the Magic-sets transformation.
- The CFL-reachability approach opens up possibilities for gathering other "shape-related" information by making only modest changes in the shape-analysis specification. The *unmatched_path* (L_4-path) information gathered in Example 2.4 illustrates one of the possible ways this flexibility can be exploited.

Related work on pointer analysis in the dataflow-analysis community includes [10,16,11,6]. Our work is less general than the techniques reported in those papers in that our method does not handle destructive-update operations. However, there are a couple of aspects of our work that, compared with previous work, make it unique. For example, the techniques of Hendren and Nicolau [10] and Larus and Hilfinger [16] are based on regular languages, whereas our approach makes use of context-free languages. Another novel aspect of our approach is that, because the dataflow equations themselves are converted into a graph, in some sense our approach builds up only a *single* graph to represent the store, namely the equation dependence graph. The equation dependence graph *indirectly* represents the store for *all* points in the program; the grammars L_1, L_2, L_3, and L_4 permit appropriate information for the individual points in the program to be extracted from the equation dependence graph. In other approaches, a *collection* of graphs is created, one for each point in the program.

There has been some previous work in which dataflow-analysis problems have been expressed using Horn clauses. For instance, one of the examples in Ullman's book on database theory shows how a logic database can be used to solve the intraprocedural reaching-definitions problem [28, pp. 984-987]. Assmann has examined a variety of other intraprocedural dataflow-analysis problems [1]. Although Assmann expresses these problems using a certain kind of graph grammar, he points out that this formalism is equivalent to Datalog. Reps presented a way in which demand algorithms that solve interprocedural versions of the classical gen-kill dataflow-analysis problems (*e.g.*, live variables, available expressions, reaching definitions) can be obtained automatically from their exhaustive counterparts—expressed as Horn-clause programs—by making use of the Magic-sets transformation [24,23].

Reps, Sagiv, and Horwitz have applied CFL-reachability techniques to interprocedural dataflow-analysis problems [22,25]. (In this case, the context-free grammars describe the fact that the only execution paths of interest are ones in which each "return edge" has a matching "call edge" from the corresponding call site.) Although these papers do not make use of logic-programming terminology, the exhaustive algorithms described in them have straightforward implementations as logic programs. Demand algorithms can then be obtained by applying the Magic-sets transformation. However, as already discussed in Section 4, a low-overhead implementation of a demand CFL-reachability algorithm for these problems—not based on Magic-sets, and suitable for implementation in an imperative programming language—has been developed [22]. Some preliminary experimental results from such an implementation are reported in [12].

Appendix: Two Examples of the Magic-Sets Transformation

In this appendix, we present two examples that illustrate the capabilities of the Magic-sets transformation.

The Magic-sets transformation attempts to combine the advantages of a top-down, goal-directed evaluation strategy with those of a bottom-up evaluation strategy. One disadvantage with top-down, goal-directed search (at least the depth-first one employed in Prolog) is that it is incomplete—it can loop endlessly, failing to find any answer at all, even when answers do exist. Another disadvantage of top-down, goal-directed search is that it can take exponential time on examples that a bottom-up evaluation strategy handles in polynomial time.

A bottom-up strategy starts from the base relations and iteratively applies an "immediate-consequence" operator until a fixed point is reached. One advantage of a bottom-up evaluation strategy is that it is complete. It can be thought of as essentially a dynamic-programming strategy: The values for all smaller subproblems are tabulated, then the answer for the item of interest is selected. However, bottom-up evaluation strategies also have the main drawbacks of dynamic programming, namely that (i) much effort might be expended to solve subproblems that are completely irrelevant to the final answer and (ii) a great deal of space might be used storing solutions to such subproblems.

The Magic-sets approach is based on bottom-up evaluation; however, the program evaluated is a *transformed* version of the original program, specialized for answering queries of a given form. In the transformed program, each (transformed) rule has attached to it an additional literal that represents a condition characterizing when the rule is relevant to answering queries of the given form. The additional literal narrows the range of applicability of the rule and hence causes it to "fire" less often.

Example. The gains that can be obtained via the Magic-sets transformation can be illustrated by the example of answering reachability queries in directed graphs. Let "`edge(v,w)`" be a given base relation that represents the edges of a directed graph.

A dynamic-programming algorithm for the reachability problem computes the transitive closure of the entire graph—this information answers *all* possible reachability queries—then selects out the edges in the transitively closed graph that emanate from the point of interest. In a logic-programming system that uses a bottom-up evaluation strategy, the dynamic-programming algorithm can be specified by writing the following program for computing transitive closure:

```
tc(V,W) :- tc(V,X), edge(X,W).
tc(V,W) :- edge(V,W).
```

In the Coral system, which supports the Magic-sets transformation, the additional declaration

```
export tc(bf).
```

directs the system to transform the program to a form that is specialized for answering queries in which the first argument is bound and the second is free (*i.e.*, queries of the form "`?tc(a,W)`"). The transformed program that results is

```
tc_bf(V,W) :-
        magic_tc_bf(V),
        tc_bf(V,X),
        edge(X,W).
tc_bf(V,W) :-
        magic_tc_bf(V),
        edge(V,W).
```

Given a query "`?tc(a,W)`", the additional fact "`magic_tc_bf(a)`" is adjoined to the above set of transformed rules. These are then evaluated bottom-up to produce (as answers to the query) the tuples of the relation `tc_bf`.

A magic fact, such as "`magic_tc_bf(a)`", should be read as an assertion that "The problem of finding tuples of the form `tc(a,_)` arises in answering the query". In this example there are no rules of the form

```
magic_tc_bf(X) :- . . .
```

Consequently, during evaluation no additional facts are ever added to the `magic_tc_bf` relation; that is, the only "magic fact" ever generated is the initial one, `magic_tc_bf(a)`. (Our next example will illustrate the more general situation.) Because all of the rules in the transformed program are guarded by a literal "`magic_tc_bf(V)`", the bottom-up evaluation of the transformed program only visits vertices that are reachable from vertex `a`. In effect, the original "dynamic-programming" algorithm—perform transitive closure on the entire graph, then select out the tuples of interest—has been transformed into a reachability algorithm that searches only vertices reachable from vertex `a`. □

Example. Suppose we have a base relation that records parenthood relationships (*e.g.*, a tuple "`parent(x,x1)`" means that `x1` is a parent of `x`), and we would like to be able to find all cousins of a given person who are of the same generation. (In this example, a person is considered to be a "same-generation cousin" of himself.)

In a logic-programming system that uses a bottom-up evaluation strategy, a dynamic-programming algorithm can be specified by writing the following program:

```
same_generation(X,X).
same_generation(X,Y) :-
        parent(X,X1),
        same_generation(X1,Y1),
        parent(Y,Y1).
```

The directive

```
export same_generation(bf).
```

directs the Coral system to transform the program to a form specialized for answering queries of the form "`?same_generation(a,Y)`", which causes the program to be transformed into

```
magic_same_generation_bf(U) :-
        magic_same_generation_bf(V),
        parent(V,U).
same_generation_bf(X,X) :-
        magic_same_generation_bf(X).
same_generation_bf(X,Y) :-
        magic_same_generation_bf(X),
        parent(X,X1),
        same_generation_bf(X1,Y1),
        parent(Y,Y1).
```

Given a query "`?same_generation(a,Y)`", the additional fact "`magic_same_generation_bf(a).`" is adjoined to the above set of rules, which are then evaluated bottom-up.

Unlike the previous example, the transformed program produced in this example *does* have a rule with "`magic_same_generation_bf(U)`" in the head:

```
magic_same_generation_bf(U) :-
        magic_same_generation_bf(V),
        parent(V,U).
```

The presence of this rule will cause "magic facts" other than the original one to be generated during evaluation. Note that the members of relation `magic_same_generation_bf` will be exactly the ancestors of `a` (the so-called "cone of `a`" [2]). During bottom-up evaluation of the transformed rules, the effect of the `magic_same_generation_bf` predicate is that attention is restricted to just same-generation cousins of ancestors of `a`. □

Note that in a bottom-up evaluation, the transformed program (the demand algorithm) will never perform more work than the untransformed program (the exhaustive algorithm) would—modulo a small amount of overhead for computing magic facts, which are reported to be only a small fraction of the generated facts [3]. In practice, the demand algorithm usually performs far less work than the exhaustive algorithm.

Acknowledgements

I am grateful for the comments and helpful suggestions that Susan Horwitz and Mooly Sagiv provided about the work.

References

1. Assmann, U., "On edge addition rewrite systems and their relevance to program analysis," Unpublished report, GMD Forschungsstelle Karlsruhe, Karlsruhe, Germany (1993).
2. Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., "Magic sets and other strange ways to implement logic programs," in *Proceedings of the Fifth ACM Symposium on Principles of Database Systems,* (Cambridge, MA, March 1986), (1986).
3. Bancilhon, F. and Ramakrishnan, R., "Performance evaluation of data intensive logic programs," pp. 439-517 in *Foundations of Deductive Databases and Logic Programming,* ed. J. Minker,Morgan-Kaufmann (1988).
4. Beeri, C. and Ramakrishnan, R., "On the power of magic," pp. 269-293 in *Proceedings of the Sixth ACM Symposium on Principles of Database Systems,* (San Diego, CA, March 1987), (1987).
5. Ceri, S., Gottlob, G., and Tanca, L., "What you always wanted to know about Datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering* **1**(1) pp. 146-166 (March 1989).
6. Chase, D.R., Wegman, M., and Zadeck, F.K., "Analysis of pointers and structures," *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6) pp. 296-310 (June 1990).
7. Cousot, P. and Cousot, R., "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," pp. 238-252 in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages,* (Los Angeles, CA, January 17-19, 1977), ACM, New York, NY (1977).
8. Duesterwald, E., Gupta, R., and Soffa, M.L., "Demand-driven computation of interprocedural data flow," pp. 37-48 in *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages,* (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).
9. Heintze, N., "Set based analysis of ML programs," Technical Report CMU-CS-93-193, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (July 1993).
10. Hendren, L.J. and Nicolau, A., "Parallelizing programs with recursive data structures," *IEEE Transactions on Parallel and Distributed Systems* **1**(1) pp. 35-47 (January 1990).
11. Horwitz, S., Pfeiffer, P., and Reps, T., "Dependence analysis for pointer variables," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices* **24**(7) pp. 28-40 (July 1989).
12. Horwitz, S., Reps, T., and Sagiv, M., "Demand interprocedural dataflow analysis," Unpublished Report, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI (March 1995).
13. Jones, N.D. and Muchnick, S.S., "Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra," pp. 380-393 in *Program Flow Analysis: Theory and Applications,* ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).
14. 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).
15. Kasami, J., "An efficient recognition and syntax analysis algorithm for context-free languages," Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA (1965).
16. Larus, J.R. and Hilfinger, P.N., "Detecting conflicts between structure accesses," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 21-34 (July 1988).
17. 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).
18. Mogensen, T., "Partially static structures in a self-applicable partial evaluator," pp. 325-347 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation,* (Gammel Avernaes, Denmark, 18-24 October, 1987), ed. D. Bjϕrner, A.P. Ershov, N.D. Jones,North-Holland, New York, NY (1988).
19. Mogensen, T., "Separating binding times in language specifications," pp. 12-25 in *Fourth International Conference on Functional Programming and Computer Architecture,* (London, UK, Sept. 11-13, 1989), ACM Press, New York, NY (1989).
20. Ramakrishnan, R., Seshadri, P., Srivastava, D., and Sudarshan, S., "The Coral user manual: A tutorial introduction to Coral," Unpublished documentation, Computer Sciences Department, University of Wisconsin, Madison, WI (1993). (Available via ftp from ftp.cs.wisc.edu.)
21. Ramakrishnan, R., Seshadri, P., Srivastava, D., and Sudarshan, S., "Coral Release 1.1," Software system, Computer Sciences Department, University of Wisconsin, Madison, WI (May 1994). (Available via ftp from ftp.cs.wisc.edu.)
22. Reps, T., Sagiv, M., and Horwitz, S., "Interprocedural dataflow analysis via graph reachability," TR 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark (April 1994). (Available on the World Wide Web at ftp://ftp.diku.dk/diku/semantics/papers/D-215.ps.Z.)
23. 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).
24. Reps, T., "Solving demand versions of interprocedural analysis problems," pp. 389-403 in *Proceedings of the Fifth International Conference on Compiler Construction,* (Edinburgh, Scotland, April 7-9, 1994), *Lecture Notes in Computer Science,* Vol. 786, ed. P. Fritzson,Springer-Verlag, New York, NY (1994).
25. Reps, T., Horwitz, S., and Sagiv, M., "Precise interprocedural dataflow analysis via graph reachability," pp. 49-61 in *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages,* (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).
26. Reynolds, J.C., "Automatic computation of data set definitions," pp. 456-461 in *Information Processing 68: Proceedings of the IFIP Congress 68,* North-Holland, New York, NY (1968).
27. 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).
28. Ullman, J.D., *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies,* Computer Science Press, Rockville, MD (1989).
29. Yannakakis, M., "Graph-theoretic methods in database theory," pp. 230-242 in *Proceedings of the Symposium on Principles of Database Systems,* (1990).
30. Younger, D.H., "Recognition and parsing of context-free languages in time *n***3," *Information and Control* **10** pp. 189-208 (1967).