SYMBOLIC EVALUATION AND THE GLOBAL VALUE GRAPH

by
John H. Reif[1] and Harry R. Lewis[2]
Center for Research in Computing Technology
Harvard University

Summary.

This paper is concerned with difficult global flow problems which require the symbolic evaluation of programs. We use, as is common in global flow analysis, a model in which the expressions computed are specified, but the flow of control is indicated only by a directed graph whose nodes are blocks of assignment statements. We show that if such a program model is interpreted in the domain of integer arithmetic then many natural global flow problems are unsolvable. We then develop a direct (non-iterative) method for finding general symbolic values for program expressions. Our method gives results similar to an iterative method due to Kildall and a direct method due to Fong, Kam, and Ullman. By means of a structure called the global value graph which compactly represents both symbolic values and the flow of these values through the program, we are able to obtain results that are as strong as either of these algorithms at a lower time cost, while retaining applicability to all flow graphs.

1. Introduction.

This paper is concerned with the problem of symbolic evaluation in the context of a global flow model of computation. Thus the only statements in the programming language retained in the model are assignment statements whose left-hand sides are variables and whose right-hand sides are expressions built up from fixed sets of variables, function signs, and constant signs. The language does not provide for subroutining, and all intraprogram control flow is reduced to a directed graph indicating which blocks of assignment statements may be reached from which others, but giving no information about the conditions under which such branches might occur.

Now in this context we mean by symbolic evaluation the task of discovering, for each expression t in the program text, an expression α for the value of t which is valid for all possible executions of the program. Such an expression α will be said to cover t. Note that lexically identical expressions located at different positions in the program text may well have different covers; thus t must be considered to be "tagged" with its location in the program. Such an expression t, considered as located at a particular position in the program, is called a text expression.

Kildall[Ki] presents various "expression optimizations" for improving the efficiency of object code derived from text expressions. Many, but not all, of Kildall's expression optimizations reduce to the problem of constructing covers for text expressions.

1) A text expression t is constant if t has a cover which is a constant sign.

2) The birthpoint of a text expression t is the earliest node in the flow graph (relative to the partial ordering of nodes by domination discussed below) at which the computation of t is defined. Any node n of the flow graph occuring between (relative to this domination ordering) the birthpoint of t and the original location of t has a cover for t in terms of covers for the variables at n. The earliest such node n, with the further property that the computation of t induces no new errors at node n, is called the safepoint of t. The computation of t may safely be moved to the safepoint n. (The text expression appropriate at node n may not be lexically identical to t.) The safety of code movement is also discussed in [CA,G,Ke1,R].

1. Research supported by ARPA grant 738-7367.
2. Research supported by NSF grant MCS76-09375.

Preprint of paper appearing in 4th ACM Symposium on Principals of Programming Languages, Los Angeles, CA, January 1977, pp. 104-118.

3) Text expression t_1 is redundant with respect to text expression t_2 if t_1 occurs in the same block as t_2 and t_1 is covered by the same formula as t_2.

4) A cover for a global variable on exit from a node in a loop is a loop invariant. This problem is dicussed in detail in Fong and Ullman[FU] and Wegbreit[W].

5) The available expressions at node n are the computations which occur on every execution path from the start node to n.

Various algorithms[A,C,GW,HU,KU1,Ke2,Ke3,S, T4,U] have been developed for solving "easy" versions of global flow problems where the transformations through blocks can be computed by bit vector operations. Expression optimizations may give considerably more powerful results than the easier code improvements. However, we shall demonstrate that it is not possible in general to compute exact solutions of Kildall's expression optimization problems in the arithmetic domain (Kam and Ullman[KU2] have recently demonstrated that there exist global flow problems posed in certain global flow analysis frameworks which are unsolvable). It follows that we must look for heuristic methods for good, but not optimal, solutions to these problems.

In order to compare our methods with others we must fix the relevant parameters of the program and flow graph. We let n and a be the cardinality of the node and edge sets, respectively, of the flow graph; we let σ be the number of variables in the program; and let ℓ be the length of the program text. Our careful consideration of the parameter ℓ - avoiding, for example, redundant representations of the same expression - is one of the chief novelties of our approach; previous authors have analyzed their algorithms primarily from the point of view of the flow graph parameters n and a. (If we built into the programming language a construct for the declaration of variables local to a block, then the parameter σ here could be the number of global variables. For the sake of simplicity we do not make this refinement, but the interested reader will have no difficulty in seeing how our time bounds could be tightened in this way.)

Kildall[Ki] presents an algorithm, based on an iterative method, for computing approximate solutions to various expression optimization problems. A version of the Kildall algorithm used for the discovery of constant text expressions may require $\Omega(\sigma(\ell+a))$ elementary steps and $\Omega(\sigma a)$ operations on bit vectors of length $O(\sigma\ell)$. ($\Omega(f(x))$ is a function bounded from below by $k*f(x)$ for some k. See Knuth[Kn2].) Fong and Ullman [FU] show that the Kildall algorithm discovers onl restricted class of constant text expressions wh will be called simple constants. Figure 1 gives example of a text expression which is constant is not a simple constant.

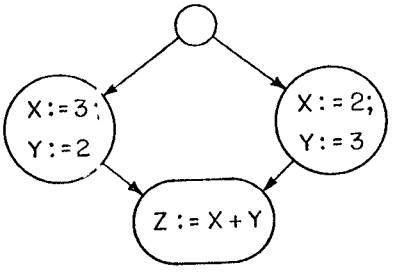


FIGURE 1

Kildall's algorithm may also be used compute a certain class of covers, which we w call minimal fixed point covers. Fong, Kam, Ullman[FKU] give another algorithm, based c direct (noniterative) method which gives wea results than Kildall's algorithm and is restric to reducible flow graphs. Kildall's algorithm require $\Omega(\ell n^2)$ elementary steps and Fong, Kam, Ullman's algorithm may require $\Omega(\ell a \log($ elementary steps. A main inefficiency of both these algorithms is in the representation of covers. Directed acyclic graphs (dags) are used represent expressions, but separate dags are nee at each node of the flow graph. Since a representing a cover may be of size $\Omega(\ell)$ the tc space cost may be $\Omega(\ell n)$. Various operations these dags, which are considered to be "extend steps by Fong, Kam, and Ullman[FKU], cost Ω elementary steps and cannot be implemented by fixed number of bit vector operations. In gener any global flow algorithm for symbolic evaluat which attempts to pool information separately each node of the flow graph will have time cost $\Omega(\ell a)$, since the pools on every pair of adjac nodes must be compared. Since $\ell \geq n$, such a t cost is unacceptable for practical applications.

The global value graph (GVG) used in t paper is related to a structure used Schwartz[Sc] to represent the flow of val through the program. The use of the GVG leads t relatively efficient direct method for symbo evaluation which works for all flow graphs. method derives its efficiency by representing covers with a single dag, rather than a separ dag at each node. In general, the GVG is of s $O(\sigma a+\ell)$, although as described in Section 4 in m

discovery of simple constants (these are the constants found by Kildall's algorithm) is linear in the size of the GVG, and the time cost of our algorithm for finding minimal fixed point covers is almost linear in the size of the GVG. (Our algorithms work for all flowgraphs.) Thus our algorithm for symbolic evaluation takes time almost linear in $\sigma a + \ell$ ($a + \ell$ in many cases), as compared to Kildall's which may require $\Omega(\ell n^2)$ steps.

2. Graph Theoretic Notions.

A digraph $G = (V, E)$ consists of a set V of elements called nodes and a set E of ordered pairs of nodes called edges. The edge (u,v) departs from u and enters v. We say u is a predecessor of v and v is a successor of u. The outdegree of a node v is the number of successors of v and the indegree is the number of predecessors of v. A path from u to w in G is a sequence of nodes $p = (u=v_1, v_2, \ldots, v_k=w)$ where $(v_i, v_{i+1}) \in E$ for all i, $1 \leq i < k$. The length of the path p is $k-1$. The path p is a cycle if $u = w$. A node u is reachable from a node v if either $u = v$ or there is a path from u to v.

We shall require various sorts of special digraphs. A rooted digraph (V, E, r) is a triple such that (V, E) is a digraph and r is a distinguished node in V, the root, such that r has no predecessors and every node in V is reachable from r. A digraph is labeled if it is augmented with a mapping whose domain is the vertex set. A multigraph is a digraph augmented with an ordering of the edges departing from each node.

A digraph G is acyclic if G contains no cycles, cyclic otherwise. Let G be acyclic. A successor of a node v is called a son of v. The nodes of G have a partial ordering, called a topological ordering of G, under which each node precedes all its sons. A rooted acyclic digraph T is a tree if every node v other than the root has a unique predecessor, the father of v. If u is reachable from v in T, u is a descendant of v and v is a ancestor of u. The reverse of a topological ordering of a tree is called a postordering. A spanning tree of digraph $G = (V, E)$ is a tree with node set V and an edge set contained in E.

Let $G = (V, E, r)$ be a rooted digraph. A node u dominates a node v if $u \neq v$ and every path from the root to v includes u. It is easily shown that there is a unique tree T_G, called the dominator tree of G, such that u dominates v in G iff $u \neq v$ and u is an ancestor of v in T_G. We write $u \rightarrow v$ if there is a path from u to v in T_G. Thus $\overset{*}{\rightarrow}$ is a partial order.

All of the above properties of digraphs may be computed very efficiently. An algorithm has linear time cost if the algorithm runs in time $O(n)$ on input of length n and has almost linear time cost if the algorithm runs in time $O(n\alpha(n))$ where α is the extremely slow growing function of [T2] (α is related to a functional inverse of Ackermann's function). Using adjacency lists, a graph with V nodes and E edges may be represented in space $O(V+E)$. Knuth[Kn1] gives a linear time algorithm for computing a topological ordering of an acyclic digraph. Tarjan [T1] gives a linear time algorithm for computing a spanning tree and in [T3] gives an almost linear time algorithm for computing the dominator tree of a rooted digraph.

3. The Global Flow Model.

Let P be a program to which we wish to apply various global code improvements. In this section we formulate a global flow model for P, similar to a model described by Aho and Ullman[AU].

The program flow graph $F = (N, A, s)$ is a digraph rooted at the start node $s \in N$. An execution path is a path in F beginning at s. Hereafter $\overset{*}{\rightarrow}$ will denote the partial order by domination with respect to the fixed rooted digraph F.

As described in Section 1, associated with each node of the program graph is a block of assignment statements. These blocks do not contain conditional or branch statements; control information is specified by the program flow graph. Program variables are taken from the set $\{X_1, X_2, \ldots\}$. A text expression t is an expression built from program variables and fixed sets C of constant signs and Θ of function signs. The text expression t is considered to be located at a particular line of code in block $loc(t) \in N$; thus two lexically identical expressions located in different blocks, or even in different lines of the same block, will be considered distinct text expressions. An assignment statement is of the form

$$X := t$$

where X is a program variable and t is a text expression.

Let Σ be the set of program variables of P. We introduce another set of variables $\Sigma^N = \{X^n \mid n \in N, X \in \Sigma\}$ to denote the values of variables on

entry to nodes. Let EXP be the set of expressions over C, Θ, and the variables in the set Σ^N. Thus, $\alpha \in EXP$ is a finite expression consisting of either a constant symbol $c \in C$, a symbol X^n representing the value of global variable X on input to node n, or a k-ary function symbol $\theta \in \Theta$ prefixed to a k-tuple of expressions in EXP. The covering expressions sought are expressions in EXP.

Now we define birthpoint(α), which intuitively is the earliest point at which all the quantities referred to in α are defined. Let $N(\alpha) = \{n \mid$ the symbol X^n occurs in $\alpha\}$. If $N(\alpha)$ is empty then birthpoint(α) = s and otherwise birthpoint(α) is the minimum node in $N(\alpha)$ relative to $\overset{*}{\rightarrow}$. The birthpoint need not exist for arbitary expressions in EXP, but will be well-defined in all the relevant cases (i.e. birthpoint exists for all covers of text expressions).

An interpretation I for the program P is an ordered pair (U, *). The universe U contains a distinct value c^* for each constant sign $c \in C$. For each k-ary function sign $\theta \in \Theta$, there is a unique k-ary operator θ^* which maps k-tuples in U^k into U. Also $c_1^* \neq c_2^*$ for each distinct $c_1, c_2 \in C$ (every constant has at most one name). A program is in the arithmetic domain if it has the interpretation (Z, *) where Z is the set of integers and $\Theta^* = \{\theta^* \mid \theta \in \Theta\}$ = the arithmetic operations of addition, multiplication, and integer division.

An expression in EXP is put in reduced form by repeatedly substituting for each subexpression of the form $'(\theta\ c_1 \ldots c_k)'$ a constant sign c such that $c^* = \theta^*(c_1^*, \ldots, c_k^*)$ until no further substitutions of this kind can be made. It will be useful to assume a equivalence relation $\equiv$ such that for any expressions $\alpha_1, \alpha_2 \in EXP$, $\alpha_1 \equiv \alpha_2$ iff α_1 and α_2 have the same reduced form.

We take as given functions ψ and δ_n as described below, and trust that the reader can supply definitions if he chooses.

For each text expression t let $\psi(t)$ be an expression in EXP which represents the value of text expression t in terms of the values of the global variables on input to loc(t), i.e. in terms of the $X^{loc(t)}$. For example, if the block of code at node n of P is

```
X := X - 1;
Y := Y + 4;
Z := X * Y
```

and t is the expression X * Y located in this block, then $\psi(t) = '(X^n-1)*(Y^n+4)'$. We assume the blocks are reduced in the sense of Aho and Ullman[AU], so that $\psi(t) \neq \psi(t')$ if t and t' are distinct text expressions. Also, for each $n \in N$ and $X \in \Sigma$ let $\delta_n(X)$ be an expression in EXP for the value of X on output from n in terms of the values of global variables on input to n. In the example just given, $\delta_n(Y) = 'Y^n+4'$.

A global flow system Π is a triple (F, Σ, I) where F is a program flow graph, Σ is the set of program variables and I = (U, *) is an interpretation. The next definitions deal with a fixed global flow system Π = (F, Σ, I). Let α be an expression in EXP and let $p = (s, n_1, \ldots, n_k)$ be an execution path containing birthpoint(α). Then $N(\alpha) \subseteq \{s, n_1, n_2, \ldots, n_k\}$ and EVAL(α,p) is an expression for the value of α in the context of this execution path. EVAL(α,p) is defined as follows:

(1) $EVAL(\alpha, p_0) = \alpha$ where $p_0 = (s)$

(2) for $0 < i \leq k$ let $p_i = (s, n_1, \ldots, n_i)$; then $EVAL(\alpha, p_i) = EVAL(\alpha', p_{i-1})$ where α' is obtained from α by substituting $\delta_{n_{i-1}}(X)$ for X^{n_i} for all $X \in \Sigma$.

If t is a text expression located at node n then $EVAL(\psi(t), p)$ represents the value of the text expression t after execution of path p from s to node n.

An expression $\alpha \in EXP$ covers a text expression t if

(1) birthpoint(α) $\overset{*}{\rightarrow}$ loc(t).

(2) for every execution path p from s to loc(t), $EVAL(\psi(t), p) \equiv EVAL(\alpha, p)$.

The second condition insures that α correctly represents the value of t on every execution path reaching loc(t). Note that the birthpoint of any cover α of a text expression t is always well defined since the elements of $N(\alpha)$ will form a chain relative to $\overset{*}{\rightarrow}$.

A cover is a mapping SVAL (for Symbolic eVALuation) from the text expressions of P to expressions in EXP in reduced form such that for each text expression t, SVAL(t) covers t. SVAL is minimal if birthpoint(SVAL(t)) $\overset{*}{\rightarrow}$ birthpoint(α) for each α which covers text expression t.

In the introduction we gave a number of global flow problems which reduce to the problem of determining covers of text expressions. Our first result is a negative one, which rules out the possibility of finding minimal covers even in simple domains.

Theorem 3.1. In the arithmetic domain, the problem of discovering all text expressions covered by constant signs is undecidable.

Proof. Let $\{X_0, X_1, X_2, \ldots, X_k\}$ be a set of variables. Matijasevic[M] has shown that the

problem of determining if a polynomial $Q(X_1,X_2,\ldots,X_k)$ has a root in the natural numbers is recursively unsolvable. The method of proof will be to reduce this problem to that of the discovery of constant text expressions.

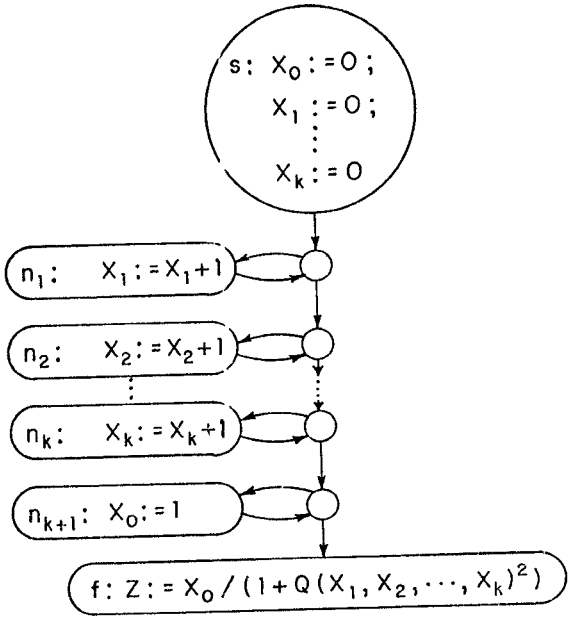


FIGURE 2

Consider the flow graph in Figure 2. Let t be the text expression "$(X_0/(1+Q(X_1,X_2,\ldots,X_k)^2)$" located at node f and let α be the cover of t in terms of the value of the variables on input to f. For any path p from s to f and for $i = 0,\ldots,k$ let $x_{i,p}$ be the value of X_i just on entering f. Observe that for any k-tuple of natural numbers z, there is an execution path from s to f such that $z = (x_{1,p},x_{2,p},\ldots,x_{k,p})$.

Suppose Q has a root in the natural numbers, say $(z_1,z_2,\ldots,z_k)$. Then it is possible to find execution paths p and q to f such that $z_i = x_{i,p} = x_{i,q}$ for all i, $1 \leq i \leq k$, and such that $x_{0,p} = 0$, and $x_{0,q} = 1$. Since $EVAL(\alpha,p) \equiv 0$ and $EVAL(\alpha,q) \equiv 1$, t is not a constant text expression.

Suppose Q has no root in the natural numbers. Then for each execution path p to f, $F(x_{1,p},x_{2,p},\ldots,x_{k,p}) \neq 0$, so $EVAL(\alpha,p) \equiv 0$. Thus, t is constant. □

Corollary 3.1. In the arithmetic domain, the following global flow problems are unsolvable: discovery of birth and safe points of code motion, redundant text expressions, loop invariants, and available expressions.

Proof. It is easy to show that the problem of discovery of constant text expressions reduces to each of these problems. □

The above results indicate that we must look for methods for computing approximations to minimal covers. The method of Kildall[Ki] may be applied to compute a class of covers which we call fixed point covers (FPCs), so called because they are fixed points of an iterative process. Let τ be a mapping: $\Sigma^N \to EXP$ extended homomorphically to a mapping $EXP \to EXP$ so that $\tau(\alpha)$ is the reduced expression derived after substituting $\tau(X^n)$ for each $X^n \in \Sigma^N$ occuring in α. A FPC is a mapping $\tau^*\psi$ for some such τ with the further property that for each $X^n \in \Sigma^N$, either (1) $\tau(X^n) = X^n$ or (2) $n \neq s$ and $\tau(X^n) = \tau(\delta_m(X))$ for every predecessor m of n in F. That is, either $\tau(X^n) = X^n$ and τ gives no information about the value of X entering node n or case (2) holds and $\tau(X^n)$ represents the common output value of X on exit from all nodes from which n can be reached by one edge. That $\tau^*\psi$ is in fact a cover in this case is proved by Theorem 3.2. We first show

Lemma 3.1. $EVAL(\tau(\alpha),p) \equiv EVAL(\alpha,p)$ for all expressions $\alpha \in EXP$ with a defined birthpoint and execution paths p containing birthpoint(α).

Proof by contradiction. Let α be an expression in EXP with a minimal birthpoint n such that $EVAL(\tau(\alpha),p) \not\equiv EVAL(\alpha,p)$ for some execution path $p = (s=n_1,\ldots,n_k)$ containing n. Thus α must have a subexpression of the form X^n such that $EVAL(\tau(X^n),p) \not\equiv EVAL(X^n,p)$. The case $\tau(X^n) = X^n$ is clearly impossible, so $n \neq s$. Otherwise, let n_i be the last occurrence of node n in p and let $m = n_{i-1}$. Then $EVAL(\tau(X^n),p)$

$= EVAL(\tau(\delta_m(X)),p)$ by definition of τ

$\equiv EVAL(\delta_m(X),p)$ by our assumption of the minimality of the birthpoint of α and since $n \overset{+}{\to} birthpoint(\delta_m(X))$

$= EVAL(X^n,p)$ by definition of EVAL.

which contradicts with our assumption that $EVAL(\tau(X^n),p) \not\equiv EVAL(X^n,p)$. □

Theorem 3.2. $\tau^*\psi$ is a cover.

Proof. By Lemma 3.1,

$EVAL(\tau(\psi(t)),p) \equiv EVAL(\psi(t),p)$

for all execution paths p to loc(t). □

Now call a FPC SVAL a minimal FPC if $birthpoint(SVAL(t)) \overset{*}{\to} birthpoint(SVAL'(t))$ for all text expressions t and all FPCs SVAL'. Then while the problem of finding minimal covers is hopeless, that of finding minimal fixed point covers is not only solvable but can be done efficiently, as stated in the introduction and described below.

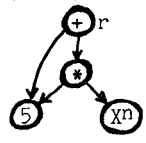
The main purpose of this paper is to establish that a minimal fixed point cover exists and is unique (Theorem 7.2) and to give an efficient algorithm for finding it (Section 9). To this end we introduce in Section 4 the global value graph and show in Sections 5-7 how it can be used to derive these results.

4. Dags and the Global Value Graph.

In this section we continue to assume a fixed global flow system $\Pi = (F, \Sigma, I)$ as introduced in Section 3, where $F = (N, A, s)$.

A labeled dag $D = (V, E, L)$ is a labeled, acyclic multigraph with a node set V, an edge list E giving the order of edges departing from nodes, and a labeling L of the nodes in V. Let $R_D(r)$ be the parenthesized inorder listing of the labels of the subgraph of D rooted at r.

Example $R_D(r) = (5+(5*X^n))$ where D =



The dag D is minimal if R_D is 1-1. Any expression or set of expressions may be represented, with no redundancy, by a minimal labelled dag D. In particular, we use the minimal dag $D_n = (V_n, E_n, L_n)$ to represent efficiently $\{\psi(t) \mid loc(t) = n\}$ (recall that $\psi(t)$ is an expression for the value of text expression t in terms of the values of variables on input to the node in which t occurs). We have assumed that each block is reduced, so we may identify the text expression located at node n of the flow graph with the nodes of the dag D_n associated with n. Hereafter we take a cover to be a mapping whose domain is, not the set of text expressions, but the set of nodes of the dag D_n representing those text expressions. Aho and Ullman[AU] introduced the use of dags for representing computations within blocks. Kildall[Ki] and Fong, Kam, and Ullman[FKU] have applied dags to various global flow problems.

We now define two partial mappings from Σ to V_n. If X occurs in the right-hand side of an assignment in block n then we call X an input variable of n and let $IN_n(X) = R_{D_n}^{-1}(X^n)$, i.e. the node of dag D_n labelled X^n. Also, if X is assigned to in block n then X is called an output variable of n and let $OUT_n(X) = R_{D_n}^{-1}(\delta_n(X))$.

We say $X \in \Sigma$ is constant between m and n, where $m \neq n$, if X is not an output variable of any node, except possibly m or n, on any path from m to n which avoids all dominators of m. Let W_n be a function from input variables of n to N such that for each input variable X of node n, $W_n(X)$ is the earliest node m (relative to $\neq$) such that either $m = n$ or X is constant between m and n. Let IDEF(n) be the set of global variables which are not constant between the immediate dominator of n and n. We can always modify the program P, at moderate cost, so that W_n and IDEF are easily computable. For example, for each $X \in \Sigma$ and for each node n of which X is not an output, add at n a dummy assignment $X := X$. This method increases the size of P to $O(\sigma n+\ell)$ and trivializes the calculation of W_n and IDEF. Alternatively, an algorithm in [R] computes W_n and IDEF directly, for P unmodified, in $O((a+t)\alpha(a+\ell))$ extended bit vector operations.

It will be convenient to assume that if $X \in IDEF(n)$ then X is also an output variable of n. We therefore add dummy assignments, if necessary, to accomplish this. Note that this modification changes neither the semantics of the program P, nor the functions IDEF and W_n; nor does it much increase the size of P.

Let n, m be a pair of nodes in the program flow graph F. A pair of nodes (v,u), where v is a node in the dag of n and u is a node in the dag of m, is a value pair if there is an input variable X of n such that $u = OUT_m(X)$, $v = IN_n(X)$ and either (a) $X \in IDEF(n)$ and $(m,n) \in A$ or (b) $X \notin IDEF(n)$ and $m = W_n(X)$.

We now come to the central definition. To model the flow of values through a program P, we introduce the global value graph GVG = (V, E, L) which is a possibly cyclic labeled multigraph such that:

1) the node set V is the union of the node sets of the dags of F,

2) E is an edge list containing (a) the edge list of each D_n and (b) all value pairs, and

3) L is a labeling of V compatible with the labeling of each D_n.

Note that the only edges departing from nodes labeled with function signs in Θ are of type (a), and the only edges departing from nodes labeled with variables in Σ^N are of type (b), and no edge departs from any node labeled with a constant in C. Also, note that the value edges of type (b) depart from a node labeled X^n and enter a node representing a value of X on entry to node n; i.e. the direction is opposite to the implied flow of control. A path in GVG consisting entirely of value pairs is called a value path; this notion is due to Schwartz[Sc].

Below is a simple algorithm for building a global value graph.

Algorithm A.

INPUT Π with each block n represented by its dag D_n for all $n \in N$.
OUTPUT GVG = (V, E, L)
begin
  compute W and IDEF;
  Modify Π so that if $X \in IDEF(n)$,
    then X is output at n;
  L:= an array of length $dn+\ell$;
  V := E := ϕ;
  for all nodes $n \in N$ do
    begin
      Let $D_n = (V_n, E_n, L_n)$:
      append E_n to E, V_n to V;
      for all $v \in V_n$ do $L(v) := L_n(v)$;
      for all input variables X of node n do
        if $X \in IDEF(n)$ then
          for all predecessors m of n in F
            do add $(IN_n(X), OUT_m(X))$ to E;
        else add $(IN_n(X), OUT_{W_n(X)}(X))$ to E;
    end;
end.

Let d be the average cardinality of IDEF(n) for nodes n in the flow graph; many block structured programs have d of order 1, however d may be σ in the worst case.

Theorem 4.1. Algorithm A is correct and has time and space bounds $O(da+\ell)$.

Proof. The correctness of Algorithm A follows directly from the definition of the global value graph GVG. The size of the program is increased by O(dn) by the modification in the second line. The time to process each node $n \in N$ is

$$O((1 + |IDEF(n)|)\cdot indegree(n)+|V_n|).$$

But $dn+\ell \geq |V| = \sum_{n\in N} |V_n|$, $a = |A| = \sum_{n\in N} indegree(n)$,

and $da \geq \sum_{n\in N} |IDEF(n)|\cdot indegree(n)$.

Thus, the total time cost is $O(da+\ell)$. □

We now partially characterize minimal FPCs in terms of the GVG.

Theorem 4.2. If SVAL is a minimal FPC then for all $v \in V$,

(a) if $L(v) \in C$ then $SVAL(v) = L(v)$;

(b) if $L(v) \in \Theta$ then $SVAL(v) \equiv$ '$(L(v)\ SVAL(u_1)\ldots SVAL(u_k))$' (that is, the expression whose first subexpression is the symbol $L(v) \in \Theta$, whose second subexpression is $SVAL(u_1) \in EXP$, etc.) where $u_1,\ldots,u_k$ are the successors (in order) of v in GVG,

(c) if $L(v) \in \Sigma^N$ then either (i) $SVAL(v) = L(v)$ or (ii) v has at least one successor in GVG and $SVAL(v) = SVAL(u)$ for all such successors u.

Proof. Let $SVAL = \tau\cdot\psi$ be a minimal fixed point cover and let $v \in V$.

Case a. If $L(v) = c$ where $c \in C$, then $\tau(\psi(v)) = \psi(v) = c$.

Case b. If $L(v) = \theta$ is a k-ary function sign in θ and $u_1,\ldots,u_k$ are the successors of v in G, then

$$\tau(\psi(v)) = \tau('(\theta\ \psi(u_1)\ldots\psi(u_k))')$$
$$\equiv '(\theta\ \tau(\psi(u_1))\ldots\tau(\psi(u_k)))'.$$

Case c. If $L(v) = X^n$ where $X^n \in \Sigma^N$, we have $\psi(v) = X^n$ so $v = IN_n(X)$. By definition of the FPC, either (1) $\tau(\psi(v)) = L(v) = X^n$ or (2) $n \neq s$ and $\tau(\psi(v)) = \tau(X^n) = \tau(\delta_m(X))$ for all predecessors m of n in F. In case (2), consider some edge $(v,u) \in E$. Since (v,u) is a value pair, $u = OUT_m(X)$ for some $m \in N$ and either (i) $X \in IDEF(n)$ and $(m,n) \in A$ or (ii) $X \notin IDEF(n)$ and $m = W_n(X)$. In case (i) recall that $\delta_m(X) = \psi(OUT_m(X))$, so we have $\tau(\psi(v)) = \tau(\psi(OUT_m(X))) = \tau(\delta_m(X))) = \tau(\psi(u))$. Otherwise, in case (ii) X is constant between m and n; and also u is the unique successor of v in GVG. If $\tau(\psi(v)) \neq \tau(\psi(u))$ then we can construct a FPC SVAL' such that $SVAL'(v) = \tau(\psi(u))$, which violates the assumption that $\tau\cdot\psi$ is minimal. Thus, $\tau(\psi(v)) = \tau(\psi(u))$. □

5. Discovery of Simple Constants.

A text expression t is called a simple constant iff $SVAL(t) \in C$ for some FPC SVAL. We call $v \in V$ a simple constant if $R_D(v)$ is, where D is the dag of which v is a node, and the expression $R_D(v)$ is taken as located at the appropriate point in the text of the program P. It follows from Theorem 4.2 that simple constants can be detected by propagating possible constants through the GVG, starting from nodes labeled with constants, and then testing for conflicts. This leads to an algorithm for simple constant detection with time cost linear in the size of the GVG. The algorithm computes a new labeling L' such that if $v \in V$ is a simple constant covered by $c \in C$ then $L'(v) = c$, and otherwise $L'(v) = L(v)$. The result of this relabeling of simple constants with corresponding constant signs is the reduced global value graph RGVG = (V,E,L').

Let > be a postordering of some spanning tree of F. We construct an acyclic subgraph of GVG by retaining just those edges which are oriented between nodes whose loc values are compatible with >. Formally, let $E_> = \{(v,u) | (v,u) \in E$ and either (1) $loc(v) > loc(u)$ or (2) $loc(v) = loc(u)$ and $v \neq u\}$. Then observe that $(V, E_>)$ is acyclic. We shall propagate constants in the reverse of a topological order of $(V, E_>)$.

Our algorithm for simple constant discovery is given below.

Algorithm B.
INPUT GVG = (V, E, L), F.
OUTPUT RGVG = (V, E, L').

```
begin
  FLAG := a bit vector of length |V|;
  L' := an array of length |V|;
  Let > be a postordering of a spanning tree of F;
  E> := φ;
  for all (v,u) ε E do
    if either (1) loc(v)>loc(u) or
    (2) loc(v)=loc(u) and v≠u
      then add (v,u) to E>;
  comment propagate constants;
a:for each v ε V in reverse topological
  order of (V,E>) do
    begin
      FLAG(v) := true;
      if L(v) = c where c ε C then d: L'(v) := c;
      else if L(v) = θ where θ ε Θ,
          u1,...,uk are successors of v in GVG,
        L'(ui) = ci with ci ε C for 1 ≤ i ≤ k
        and there is c ε C such that
        c* = θ*(c1*,...,ck*)
        then e: L'(v) := c;
      else if L(v) ε Σ^N, v has a successor in GVG,
        and there is c ε C such that
        L'(u) = c for all such successors u,
        then f: L'(v) := c;
      else g: add v to Q;
    end;
  comment test for conflicts;
b:for all v ε V s.t. L(v) ε Σ^N, L'(v) ε C do
    if there is u ε V such that (v,u) ε E-E>
     and L'(u) ≠ L'(v) then
      h: add v to Q;
c:till Q = φ do
    begin
      delete some v from Q;
      if FLAG(v) then
        begin
        i:FLAG(v) := false;
          L'(v) := L(v);
          add all predecessors of v in GVG to Q;
        end;
    end;
end.
```

Lemma 5.1. If v ε V is a simple constant covered by some constant symbol c, then L'(v) is set to c in step (d), (e), or (f).

Proof, by induction on the reverse of the topological order of (V,E>).

Basis step. Observe that, by Theorem 4.2, if v is a leaf of (V,E>) and v is a simple constant covered by c ε C, then L(v) = c.

Induction step. Suppose for some v ε V, Lemma 5.1 is correct for all w occurring after v in the topological order of (V, E>). Suppose v is a simple constant covered by c. Consider the following cases.

Case 1. If L(v) ε C, then L'(v) is set to L(v) in step (d). But by our observation in the basis step, L(v) = c. Thus, L'(v) is correctly set to c in step (d).

Case 2. If L(v) = θ where θ ε Θ, let $u_1,\ldots,u_k$ be the successors of v in GVG. Here L'(v) is set to c only if $c^* = \theta^*(c_1^*,\ldots,c_k^*)$ where L'(u_i) has value c_i ε C for $1 \le i \le k$. But by Theorem 4.2, the u_i are simple constants and so by the induction hypothesis, L'(u_i) has been previously set to c_i. Thus, L'(v) is set to c in step (e).

Case 3. If L(v) ε Σ^N, let S be the set of successors of v in (V,E>). L'(v) is set to c in step (f) only if v has a successor in GVG and L'(u) = c for all u ε S. To show that |S| > 0, we must recall how the GVG is constructed. We have v = $IN_n(X)$ for some n ε N-{s} and X ε Σ. If X ε ASSIGN(n), then S = {$OUT_m(X)$ | n>m} and otherwise S = {$OUT_{m_0}(X)$}, where $n > m_0 = W_n(X)$.

By Theorem 4.2, all successors of v in GVG must be simple constants covered by c. By the induction hypothesis, each u ε S has been previously set to c. Thus, L'(v) is set to c in step (f). □

Let L^+ be the value of L' at step (b), and observe that just before this step Q has the value

$$Q_0 = \{v \mid L^+(v) \notin C\}.$$

At step (b), Q is set to

$Q_1 = Q_0 \cup \{v \mid L^+(v) \in C$ but L(v) ε Σ^N and there is u ε V such that (v,u) ε E-E> and $L^+(u) \neq L^+(v)\}$.

Then v ε V is eventually added to Q and L'(v) set to L(v) iff some element of Q_1 is reachable in GVG from v. If v ε V is labeled by L^+ with a constant sign, then we show

Lemma 5.2. v is not a simple constant iff some element of Q_1 is reachable in GVG from v.

Proof. IF. Suppose v is not a simple constant, but no element of Q_1 is reachable from v. Then let SVAL be the mapping: V -> EXP such that for each w ε V, if w is reachable from v then SVAL(w) = $L^+(w)$ and otherwise SVAL(w) = α, where α is derived from ψ(w) by substituting $L^+(v)$ for the subexpression ψ(v). Then we can show that SVAL is a FPC. But SVAL(v) = $L^+(v)$ ε C, which implies that v is a simple constant, contradiction.

ONLY IF. Suppose some element of Q_1 is reachable from v in GVG. Clearly if v ε Q_1, then v is not a simple constant. Assume for some k > 0, if there is a path of length less than k in GVG from some u ε V to an element of Q_1, then u is not a simple constant. Suppose there is a path $p = (v=x_0,x_1,\ldots,x_k)$ of length k from v to some element of Q_1. If k = 1, then x_1 ε Q_1, and otherwise if k > 1, then $(x_1,\ldots,x_k)$ is a path of length k-1. By the induction hypothesis, x_1 is not a simple constant. But (v,x_1) ε E and by Theorem 4.2, v is not a simple constant. □

Theorem 5.1. Algorithm B is correct and has time

cost linear in the size of the GVG.

Proof. The correctness of Algorithm B follows directly from Lemmas 5.1 and 5.2.

In addition we must show Algorithm B has time cost linear in $|V|+|E|$. The initialization costs time linear in $|V|$. The postordering $>$ may be computed in time linear in $|N|+|A|$ by the depth first search algorithm of [T1]. The time to process each $v \in V$ at step (a) and (b) is $O(1+\text{outdegree}(v))$. Step (i) can be reached at most $|V|$ times and the time cost to process each node v at step (i) is $O(1+\text{indegree}(v))$. Thus, the total time cost is linear in $|V|+|E|$. □

In general, we may improve the power of Algorithm B for particular interpretations by applying algebraic identities to reduce expressions in EXP more often to constant symbols. For example, in the arithmetic domain we can use the fact that 0 is the identity element under integer multiplication to modify Algorithm B so that if node v is labeled by L with the multiplication sign and a successor of v in GVG is covered by 0, then at step (e) we may set $L'(v)$ to the constant sign corresponding to 0.

6. The RGVG and the Minimal FPC.

In this section we use the RGVG to further characterize a minimal FPC. The following theorem follows easily from Theorems 4.2 and 5.1.

Theorem 6.1. If SVAL is a minimal FPC then for each $v \in V$,

(a) if $L'(v) \in C$ then $SVAL(v) = L'(v)$,

(b) if $L'(v) \in \Theta$ then $SVAL(v) = '(L'(v)\ SVAL(u_1)\ldots SVAL(u_k))'$ where $u_1,\ldots,u_k$ are the successors of v in RGVG.

(c) if $L'(v) \in \Sigma^N$ then either (1) $SVAL(v) = L'(v)$ or (2) v has some successor in RGVG and $SVAL(v) = SVAL(u)$ for all such successors u.

Proof.

Case a. If $L'(v) = c$ where $c \in C$, then by Theorem 5.1 v is a simple constant covered by c and so $SVAL(v) = c$.

Case b. If $L'(v) = \theta$ where θ is a k-ary function sign in θ, then by Theorem 5.1 v can not be a simple constant. So by Theorem 4.2, $SVAL(v) \equiv '(\theta\ SVAL(u_1)\ldots SVAL(u_k))'$ where $u_1,\ldots,u_k$ are the successors of v in RGVG. But the $SVAL(u_i)$ are in reduced form so

$SVAL(v) = '(\theta\ SVAL(u_1)\ldots SVAL(u_k))'$.

Case c. Otherwise, if $L'(v) \in \Sigma^N$ then again by Theorem 5.1 v can not be a simple constant and part (c) follows directly from Theorem 4.2. □

Let $\hat{V}$ be the set of all nodes of RGVG labeled with constant signs, function signs, or variables of the form X^s (s is the start node of F). Then Theorem 6.1 characterizes exactly the value of any minimal FPC SVAL on nodes in $\hat{V}$ in terms of the values of SVAL on the nodes in $V-\hat{V}$, i.e. in terms of the values on nodes whose labels are of the form X^n for $n \neq s$. For $v \in V-\hat{V}$ it follows from Theorem 6.1 that SVAL(v) must be SVAL(u) for some u that lies on a value path starting from v. The next theorem is critical in locating u. Call two paths almost disjoint if they have exactly one node in common. For any $v \in V-\hat{V}$ let H(v) be the set of nodes in $\hat{V}$ that lie at the ends of maximal value paths in RGVG starting from v.

Theorem 6.2. If SVAL is a minimal FPC and $v \in V-\hat{V}$, then either

(a) $SVAL(u_1) = SVAL(u_2)$ for all $u_1, u_2 \in H(v)$ and SVAL(v) is this common value; or

(b) $SVAL(u_1) \neq SVAL(u_2)$ for some $u_1, u_2 \in H(v)$, and $SVAL(v) = L'(u)$, where u is the unique node such that (1) u lies on all value paths from v to members of H(v), but (2) there are almost disjoint value paths from u to nodes $u_1, u_2 \in H(v)$ such that $SVAL(u_1) \neq SVAL(u_2)$.

We shall require a few technical lemmas to aid us in the proof of Theorem 6.2. Recall that T_F is the dominator tree of the flow graph F, $m \overset{+}{\to} n$ if there is a path from m to n in T_F, i.e. if m dominates n, and $m \overset{*}{\to} n$ if $m \overset{+}{\to} n$ or $m = n$. Then let LCA(N') be the latest (i.e. maximum relative to $\overset{*}{\to}$) common ancester in T_F of all nodes in set $N' \subseteq N$.

Lemma 6.1. Let $n \in N-\{s\}$ and let IDOM(n) be the immediate dominator of n in F. Then

$LCA(\{m \mid (m,n) \in A\}) = IDOM(n)$.

Proof. Let $n' = LCA(\{m \mid (m,n) \in A\})$. All paths from s to each m such that $(m,n) \in A$ contain n', so all paths from s to n must contain n', implying that $n' \overset{*}{\to} n$. It is obvious that $IDOM(n) \overset{+}{\to} m$ for all $(m,n) \in A$, with $m \neq IDOM(n)$, for otherwise there is a path from s to n, avoiding IDOM(n). So $IDOM(N) \overset{*}{\to} n'$. Finally, suppose $n \overset{*}{\to} m$ for all $(m,n) \in A$. Then there can be no path from s to n, which is impossible. So $IDOM(n) = n'$. □

Extend LCA to subsets S of V so that $LCA(S) = LCA(\{loc(v) \mid v \in S\})$.

Lemma 6.2. $LCA(\{w \mid (v,w) \in E\}) \overset{+}{\to} loc(v)$ for all $v \in V-\hat{V}$.

Proof. We have $v = IN_n(X)$ for some $n \in N-\{s\}$ and $X \in \Sigma$. Consider the following cases.

Case 1. $X \in IDEF(n)$. By definition of the GVG, $(v,w) \in E$ iff $w = OUT_m(X)$ for some $(m,n) \in A$. So $LCA(\{w \mid (v,w) \in E\}) = LCA(\{m \mid (m,n) \in A\}) \overset{+}{\to} n$ by Lemma 6.1.

Case 2. $X \notin IDEF(n)$. By definition of the GVG, w_0

= $OUT_m(X)$ is the unique successor of v in GVG, where $m = W_n(X)$. But $m \neq n$, so $LCA(\{w|(v,w)\varepsilon E\}) = loc(w_0) = m \neq n = loc(v)$. □

Lemma 6.3. Let $v \varepsilon V-\hat{V}$ and let $S \subseteq V-\{v\}$ such that each maximal value path in RGVG from v contains an element of S. Then $LCA(S) \neq loc(v)$.

Proof. For $k \geq 0$, let $S_k = \{w \mid$ there is a value path p from v to w and either (1) $w \varepsilon S$ with p of length $\leq k$ or (2) p does not contain any element of S and is of length k$\}$. We proceed by induction on k.

Basis step. $LCA(S_1) = LCA(\{w|(v,w) \varepsilon E\}) \neq loc(v)$.

Inductive step. Suppose, for some $k \geq 0$, $LCA(S_k) \neq loc(v)$. Then $LCA(S_{k+1})$

$= LCA(\{LCA(S_k),$
$LCA(\{w' | w \varepsilon S_k, w \varepsilon V-\hat{V}, (w,w')\varepsilon E\})\})$
$\neq LCA(S_k)$ by Lemma 6.2
$\neq loc(v)$ by the induction hypothesis. □

Now we prove Theorem 6.2. As noted above, SVAL(v) = SVAL(u) for some u reachable from v by a value path.

Case 1. Suppose there is some $\alpha \varepsilon EXP$ such that α = SVAL(w) for all $w \varepsilon H(v)$. Let us assume $SVAL(u) \neq \alpha$. If $u \varepsilon \hat{V}$, then $u \varepsilon H(v)$ which is impossible. Otherwise suppose $u \varepsilon V-\hat{V}$. Then $birthpoint(\alpha) \neq loc(w)$ for each $w \varepsilon H(v)$, so

$birthpoint(\alpha) \neq LCA(H(v))$
$\neq LCA(H(u))$ since $H(u) \subseteq H(v)$
$\neq loc(u)$ by Lemma 6.3
$= birthpoint(SVAL(v))$

which contradicts with the assumption that SVAL is minimal.

Case 2. Suppose there are $u_1, u_2 \varepsilon H(v)$ such that $SVAL(u_1) \neq SVAL(u_2)$. Let $VS(v) = \{w \mid w$ is the last element of a maximal value path in RGVG from v such that SVAL(v) = SVAL(w)$\}$. Let u be some element of VS(v).

To demonstrate that SVAL(v) = L'(u), let us suppose $SVAL(u) \neq L'(u)$. Then, by Theorem 6.1, SVAL(u) = SVAL(w) for all w such that $(v,w) \varepsilon E$. But if $p = (v=v_1,\ldots,v_i,v_{i+1},\ldots,v_k)$ is a value path in RGVG from v containing u and v_i is the last occurrence of u in p, then $SVAL(v_{i+1}) = SVAL(u) = SVAL(v)$. This implies that $u \notin VS(v)$, which is impossible. So SVAL(v) = L'(u).

Now suppose there is a $u' \varepsilon VS(v)$ distinct from u. But then SVAL(v) = L'(u') which is impossible since $L'(u) \neq L'(u')$. Thus, all value paths in RGVG from v to an element of H(v) must contain u.

Now suppose part (2) of Theorem 6.2 does not hold. Then there is no pair of almost disjoint value paths from w to u_1, $u_2 \varepsilon H(v)$ such that $SVAL(u_1) \neq SVAL(u_2)$. Hence there is a $w \varepsilon V$, distinct from u, such that w is contained on all maximal value paths from u. Let SVAL' be the FPC such that for each $y \varepsilon V$, if all maximal value paths in RGVG from y contain w, then SVAL'(y) = L'(w) and otherwise SVAL'(y) = SVAL(y). So birthpoint(SVAL'(u))

$= loc(w)$ by definition of SVAL'
$\neq loc(u)$ by Lemma 6.3
$= birthpoint(SVAL(u))$

which implies that SVAL is not minimal. This is a contradiction, so part (2) holds. □

Theorem 6.2 suggests a procedure for calculating SVAL, but there is an implicit circularity since for $v \varepsilon V-\hat{V}$ the determination of SVAL(u) for $u \varepsilon H(v)$ may require the determination of SVAL(w) for some other $w \varepsilon V-\hat{V}$. The way out is by rank decomposition of RGVG as discussed in the next section. There will remain the problem of finding almost disjoint paths, which we consider in Section 8.

7. Rank Decomposition of the RGVG.

Fong, Kam, and Ullman[FKU] introduced the use of a rank decomposition for dags. Here we generalize the rank decomposition to the possibly cyclic RGVG; this gives us a method of partitioning the nodes of the RGVG into sets of nodes which may have the same cover. This allows us to apply Theorem 6.2 without circularity and thus demonstrate that the minimal FPC is unique. In section 9 we apply the rank decomposition to implement our direct method for symbolic evaluation.

The rank of a node $v \varepsilon V$ is defined:

rank(v) = 0 if v has no successor in RGVG
= 1 + MAX{rank(u) | (v,u) is in E}
for $L'(v) \varepsilon \theta$,
= MIN{rank(u) | (v,u) in E}, otherwise.

Lemma 7.1. $SVAL(w) = SVAL(v) \Rightarrow rank(w) = rank(v)$.

Proof. We proceed by induction.

Basis step. If either SVAL(w) = SVAL(v) = c for some $c \varepsilon C$ or SVAL(w) = SVAL(v) = X^S for some $X \varepsilon \Sigma$, then neither w nor v has a successor in RGVG and by definition of rank, rank(w) = rank(v) = 0.

Inductive step. Suppose for some $r > 0$, rank(w) = rank(v) for all $w,v \varepsilon V$ such that $rank(v) \leq r$ and SVAL(w) = SVAL(v).

Consider some $w, v \varepsilon V$ such that rank(v) = r. It is easy to establish from the definition of rank that rank(w) = MIN{rank(z) | $z \varepsilon H(w)$} and rank(v) = MIN{rank(z) | $z \varepsilon H(v)$}.

Case a. Suppose $SVAL(w) = SVAL(v) = \lceil(\theta\ \alpha_1 \ldots \alpha_k)\rceil$. Let $w \varepsilon H(v)$ and $w' \varepsilon H(w)$. Then by Theorem 6.1, $L(w) = L(w') = \theta$ and $SVAL(z_i) = SVAL(y_i)$ for i=1,..,k where $z_1,\ldots,z_k$ (respectively $y_1,\ldots,y_k$) are the successors of w (respectively w') in RGVG. By the induction hypothesis, $rank(z_i) = rank(y_i)$ for i=1,k. So $rank(w) = MAX\{rank(z_i) \mid 1 \leq i \leq k\}+1 = MAX\{rank(y_i) \mid 1 \leq i \leq k\}+1 = rank(w')$. Thus by the above remark rank(w) = rank(v) = r.

Case b. Suppose SVAL(w) = SVAL(v) where SVAL(v) $\in \Sigma^N$. By Theorem 6.2, there is a $u \in V-\hat{V}$ such that H(u) = H(w) = H(v). Thus, rank(w) = rank(u) = rank(v). □

To compute the rank of all nodes in RGVG we use a modified version of the depth first search developed by Tarjan[T1]. Because the search proceeds backwards, we require reverse adjacency lists to store edges in E. Note that the RANK(v) is used in two different ways; first to store the number of successors of node v which have not been visited, and later RANK(v) is set to rank(v). Let V_r, $\hat{V}_r$ be the nodes in V, $\hat{V}$ of rank r. We initially compute $\hat{V}_0$ and on the r'th execution of the main loop we compute $V_r-\hat{V}_r$ and $\hat{V}_{r+1}$.

Algorithm C.

```
INPUT RGVG = (V, E, L')
OUTPUT RANK
begin
  RANK:= an array of integers of length |V|;
  for all v ε V do
    RANK(v) := - (outdegree of v);
  r := 0;
  Q' := {v | either L'(v) is a constant sign
    or is a symbol of the form X^s};
  till Q' = φ do
    begin
      Q := Q'; Q' := φ;
      comment Q = V̂_r;
      till Q = φ do
        begin
          delete v from Q;
          for each predecessor u of v do
            if L'(v) ε Θ then
              if RANK(u) = -1 then
                begin
                  comment u ε V̂_{r+1};
                  RANK(u) := r+1;
                  add u to Q'
                end
              else RANK(u) := RANK(u) + 1;
            else if RANK(u) < 0 then
              begin
                comment u ε V_r-V̂_r;
                RANK(u) := r;
                add u to Q
              end;
        end;
      r := r + 1;
    end;
end.
```

Theorem 7.1. Algorithm C is correct and has time cost linear in |V|+|E|.

Proof. We will sketch the proof of correctess by induction on r.

Basis step. Initially, RANK(v) is set to - (outdegree of v) for each $v \in V$. So if $L(v) \in C$, then RANK(v) is set to 0. Also, Q and Q' are initially set to $\hat{V}_0$.

Inductive step. Suppose for some r > 0, we have on entering the inner loop the r'th time:

(1) $Q' = \hat{V}_r$,

(2) For each $v \in V$, RANK(v) is set to rank(v) if rank(v) < r, and RANK(v) is set to - (number of successors of v with rank > r) if rank(v) ≥ r.

By a second inductive proof on the inner loop we may show that on exit to the inner loop:

(3) For each $v \in \hat{V}$, if rank(v) = r+1 then v is added to Q' and RANK(v) set to r+1, otherwise if rank(v) > r+1 then RANK(v) is set to - (the number of successors of v with rank > r).

(4) Each $v \in V-\hat{V}$ with rank(v) = r is added exactly once to Q and RANK(v) is set to r.

Now we show that Algorithm C may be implemented in linear time. For each node $v \in V$ we keep a list (the reverse adjacency list), giving all predecessors of v. To process any $v \in Q'$ requires time O(1 + indegree(v)). Since each node is added to Q' exactly once, the total time cost is linear in |V|+|E|. □

This suffices for

Theorem 7.2. The minimal FPC is unique.

Proof. SVAL(v) for $v \in \hat{V}_0$, $V_0-\hat{V}_0$, $\hat{V}_1$, $V_1-\hat{V}_1$,... may be determined by alternately applying Theorems 6.1 and 6.2. □

Using this method could be inefficient, since Theorem 6.2 could be expensive to apply and the representions of the values could grow rapidly in size. The first problem is solved by reducing it to the problems of P-graph completion and decomposition as described in the next section. The second problem is solved by constructing a special labeled dag; the construction of this dag and the final algorithm are given in Section 9.

8. P-graph Completion and Decomposition.

This section presents an efficient method for applying Theorem 6.2 to nodes in $V_r-\hat{V}_r$.

Let RGVG = (V, E, L') as above. Now to compute SVAL, the minimal FPC, it suffices to find a subset $V^* \subseteq V$ and a mapping M from V onto V^* such that (1) SVAL(u) = SVAL(v) iff M(u) = M(v) and (2) if $v \in V-\hat{V}$ and SVAL(v) = L(v) (i.e. SVAL(v) is of the form $X^{loc(v)}$) then $v \in V^*$. Thus $M^{-1}[V^*]$ partitions V into blocks such that two nodes are in the same block iff they have the same SVAL, which

can then be found by referring to node M(v). Each block contains exactly one fixed point of M, which is called the value source of all nodes in the block; thus V^* is the set of value sources. Note that in general V^* and M are not uniquely determined, so the definition of "value source" depends on the particular M we have chosen.

We shall find value sources and compute M by reducing these problems to the problems of P-graph completion and decomposition stated below.

Let $G = (V_G, E_G)$ be any directed graph and let $S \subseteq V_G$ be a set of vertices of G such that for each vertex $v \in V_G$ there is some vertex $u \in S$ from which v is reachable.

P-Graph Completion Problem. Find

$S^+ = S \cup \{v \in V_G |$ there are almost disjoint paths from distinct elements of S to v not containing any other element of S$\}$.

This form of the problem is due to Karr[K], who shows that it is equivalent to the original formulation due to Shapiro and Saint[SS]. (Actually, this form is slightly more general than Karr's; Karr satisfies our restriction on S by stipulating that there is a single $r \in S$ from which every $v \in V_G$ is reachable.) Karr proves that for each $v \in V_G$ there is one and only one element of S^+ from which v is reachable (and his proof extends directly to our slightly more general problem).

P-Graph Decomposition Problem. Given G and S^+, find, for each $v \in V_G$, the unique $u \in S^+$ from which v is reachable.

We first show these problems can be solved efficiently. Shapiro and Saint give an $O(|V_G|^2)$ algorithm, while Karr gives a more complex $O(|V_G|\log|V_G|+|E_G|)$ algorithm. Here we reduce these problems to the computation of a certain dominator tree, for which there is an almost linear time algorithm as noted in Section 2. (This construction was discovered independently by Tarjan[T5].)

Let h be a new node not in V_G, and let G' be the rooted directed graph

$(V_G \cup \{h\}, E_G \cup \{(h,v) | v \in S\} - \{(u,v) | u \in V_G, v \in S\}, h)$.

Thus G' is derived from G by adding a new root h, linking h to every node in S, and removing the edges of G which lead to nodes in S. Let T be the dominator tree of G'.

Lemma 8.1. The members of S^+ are the sons of h in T.

Proof. IF. Let $v \in S^+$. If $v \in S$ then h is a predecessor of v in G' so h is the father of v in T. If $v \in S^+ - S$ then by definition of S^+ there are almost disjoint paths p_1, p_2 in G from distinct elements of S to v not containing any other element of S. Clearly p_1 and p_2 are also paths in G' since they contain no edge entering a member of S. Then h,p_1 and h,p_2 are paths from h to v in G' which have only their endpoints in common, so v is a son of h in T.

ONLY IF. Suppose v is a son of h in T. If h is a predecessor of v in G' then $v \in S \subseteq S^+$. Otherwise there are in G' paths h,p_1 and h,p_2 from h to v which have only their endpoints in common. Moreover these paths contain no element of S except for the first nodes of p_1, p_2, since no edge of G' enters an element of S except from h. Hence p_1, p_2 are almost disjoint paths in G' from distinct members of S to v not containing any other element of S, and hence $v \in S^+$. □

Lemma 8.2. For each $v \in V_G$, the unique node in S^+ from which v is reachable in G is the unique node which is a son of h and an ancestor of v in T.

Proof. Let w be that ancestor of v which is a son of h in T. By Lemma 8.1, $w \in S^+$, and clearly v is reachable from w in G since it is reachable from w in T. Conversely, if $w \in S^+$ is reachable from v in G then w is a son of h in T by Lemma 8.1, and w must be an ancestor of v since otherwise v would be reachable from some other member of S^+. □

Now we establish the relation of these problems to the problem of finding V^* and M as stated above. Fix some V^* and M by choosing one node of RGVG for each value of SVAL on V. For each rank r, let $G_r = (V_r, E_r)$, where V_r is the set of all nodes of RGVG of rank r as defined in Section 7 and

$E_r = \{(v,u) | (u,v) \in E, u,v \in V_r - \hat{V}_r\}$
$\cup \{(M(v),u) | (u,v) \in E, u \in V_r - \hat{V}_r, v \in \hat{V}_r\}$.

Thus G_r is the graph derived from RGVG by deleting all nodes except those in V_r and deleting all edges except those both entering and departing from nodes in V_r, and then reversing all remaining edges. (Note that any edge of RGVG departing from a member of $\hat{V}_r$ enters a node of rank r-1.) Let S_r = $\{M(v) \mid v \in \hat{V}_r\} \cup \{v \in V_r - \hat{V}_r \mid (v,z) \in E$, for some $z \notin V_r\}$. Finally, let S_r^+ be defined from S_r as in the statement of the P-graph completion problem.

Lemma 8.3. The members of S_r^+ are the value sources (relative to M) of rank r.

Proof. IF. Suppose $v \in S_r^+$.

Case a. By definition, all elements of $\{M(v) \mid v \in \hat{V}_r\}$ are value sources.

Case b. Consider some $v \in V_r - \hat{V}_r$ such that $(v,z) \in E$ for some $z \notin V_r$. Call a value path in RGVG, containing only nodes of rank r, a r-value path. Since v is of rank r, there is a r-value path p_1 from v to some node $z_1 \in \hat{V}_r$. Also, there is a r'-value path p_2, where $r' = \text{rank}(z) \neq r$, from z to some node $z_2 \in \hat{V}_{r'}$. By Lemma 7.1, $\text{SVAL}(z_1) \neq$

SVAL(z_2) so $M(z_1) \neq M(z_2)$. Since p_1 and v,p_2 are almost disjoint, by Theorem 6.2 SVAL(v) = $L'(v)$. But no other node of RGVG has label $L'(v) \in \Sigma^N$, so v is a value source.

Case c. Suppose there are in G_r almost disjoint paths $(x_1,\ldots,x_j=v)$ and $(y_1,\ldots,y_k=v)$ in G_r from distinct elements of S_r to v. By construction of G_r, there exist distinct $\overline{x}_1,\overline{y}_1 \in H(v)$ such that $M(\overline{x}_1) = x_1$, $M(\overline{y}_1) = y_1$, and $(x_2,\overline{x}_1)$ and $(y_2,\overline{y}_1)$ are value edges, and so $(v=x_j,x_{j-1},\ldots,x_2,\overline{x}_1)$ and $(v=y_k,y_{k-1},\ldots,y_2,\overline{y}_1)$ are almost disjoint value paths. As in Case (b), we show v is a value sink by constructing two almost disjoint value paths q_1 and q_2 from v to two nodes in $\hat{V}$ with distinct SVAL, and then applying Theorem 6.2.

Case c1. If $x_1,y_1 \in \hat{V}_r$, then $M(\overline{x}_1) = x_1 \neq y_1 = M(\overline{y}_1)$ so by definition of M, $SVAL(\overline{x}_1) \neq SVAL(\overline{y}_1)$ and $q_1 = (v=x_j,x_{j-1},\ldots,x_2,\overline{x}_1)$, $q_2 = (v=y_k,y_{k-1},\ldots,y_2,\overline{y}_1)$ are the desired almost disjoint value paths.

Case c2. Otherwise, without loss of generality, assume $x_1 \in V_r-\hat{V}_r$. By definition of S_r, $\overline{x}_1 = x_1$. Just as in Case (b) there is a r'-value path p_1, where $r' \neq r$, from a successor of x_1 in RGVG to a node $w_1 \in \hat{V}_{r'}$.

Case c2.1. If $y_1 \in \hat{V}$, then by Lemma 7.1 $SVAL(y_1) \neq SVAL(w_1)$ and let $q_1 = (v=x_j,x_{j-1},\ldots,x_1,p_1)$, $q_2 = (v=y_k,y_{k-1},\ldots,y_1)$.

Case c2.2. Otherwise, $\overline{y}_1 = y_1 \in V_r-\hat{V}_r$. Then we may similarily find a r^+-value path p_2, where $r^+ \neq r$, from some successor of y_1 in RGVG to some $w_2 \in \hat{V}_r$‡ For each $x \in V_r$, we may again apply Lemma 7.1 to show that SVAL(x) is distinct from both $SVAL(w_1)$ and $SVAL(w_2)$. Since y_1 is of rank r, there is a r-value path $(y_1=z_1,\ldots,z_d)$ from y_1 to some $z_d \in \hat{V}_r$.

Case c2.2.1. If $\{z_1,\ldots,z_d\} \cap \{x_1,\ldots,x_j\} = \phi$ then let $q_1 = (x_j,x_{j-1},\ldots,x_1,p_1)$, $q_2 = (y_k,y_{k-1},\ldots,y_1,z_2,\ldots,z_d)$.

Case c2.2.2. Otherwise, let $z_e = x_i$ be the last element of $z_1,\ldots,z_d$ contained in $\{x_1,\ldots,x_j\}$.

Case c2.2.2.1. If $\{z_{e+1},\ldots,z_d\} \cap \{y_1,\ldots,y_k\} = \phi$ then let $q_1 = (x_j,x_{j-1},\ldots,x_i,z_{e+1},\ldots,z_d)$, $q_2 = (y_k,y_{k-1},\ldots,y_1,p_2)$.

Case c2.2.2.2. Otherwise, let $z_f = y_{i'}$ be the last element of $z_{e+1},\ldots,z_d$ contained in $\{y_1,\ldots,y_k\}$; then $q_1 = (x_j,x_{j-1},\ldots,x_1,p_1)$, $q_2 = (y_k,y_{k-1},\ldots,y_{i'},z_{f+1},\ldots,z_d)$ are the required almost disjoint value paths.

ONLY IF. Suppose v is a value source of rank r. Then v is a fixed point of M. If $v \in \hat{V}_r$ then $v = M(v) \in S_r$ and so $v \in S_r^+$. Otherwise, by Theorem 6.2 there are in RGVG two almost disjoint value paths $(v=x_1,x_2,\ldots,x_j)$ and $(v=y_1,y_2,\ldots,y_k)$, where x_j,y_k are members of H(v) with distinct SVAL. Let

$j' = MIN(\{j\} \cup \{i \mid x_i \in S_r\})$
and $k' = MIN(\{k\} \cup \{i \mid y_i \in S_r\})$.

Then $j',k' > 1$ and $M(x_{j'}),M(y_{k'}) \in S_r$, but $x_1,\ldots,x_{j'-1}$, $y_1,\ldots,y_{k'-1} \notin S_r$. Hence $(M(x_{j'}),x_{j'-1},\ldots,x_1)$ and $(M(y_{k'}),y_{k'-1},\ldots,y_1)$ are almost disjoint paths in G_r from elements of S_r to v, containing no other members of S_r. Thus $v \in S_r^+$. □

By Karr's proof of the uniqueness of the P-graph decomposition of G_r on S_r, we have

Lemma 8.4. For $v \in V_r-\hat{V}_r$, M(v) is the unique value source contained on all paths in G_r from elements of S_r to v.

Thus the problem of computing M reduces to the problem of decomposing the reduced global value graph by rank and then constructing dominator trees. The former can be done in linear time by Algorithm C of Section 7, the latter in almost linear time by Tarjan's Algorithm[T3].

9. The Algorithm for Symbolic Evaluation.

In this section we pull together the various pieces developed in Sections 5-8 to give a unified presentation of our algorithm for symbolic evaluation. Instead of using the RGVG directly to represent SVAL, as suggested in the beginning of Section 8, we more economically represent SVAL by a dag $D^* = (V^*, E^*, L^*)$ such that

$V^* = \{M(v) \mid v \in V\}$,
$E^* = \{(M(v),M(u)) \mid (v,u) \in E \text{ and } L'(v) \in \theta\}$,
L^* is the restriction of L' to V^*.

Recall from Section 4 that R_D maps from the rooted dag D to expressions in EXP.

Lemma 9.1. $R_D^*(M(v)) = SVAL(v)$ for all $v \in V$.

Proof. Note that by definition of M, for each $v \in V$

$$SVAL(M(v)) = SVAL(v)$$

so we need only show for $v \in V^*$

$$R_D^*(v) = SVAL(v).$$

We proceed by induction on the length of the longest path in D^* from $v \in V^*$.

Basis step. If v is a leaf of D^*, then by definition of D, $L(v) \in C \cup \Sigma^N$ so $L^*(v) = SVAL(v)$. By definition of a dag, $R_D^*(v) = L^*(v) = SVAL(v)$.

Induction step. Suppose for some $j > 0$, $R_D^*(u) = SVAL(u)$ for all $u \in V$ such that the longest path from u in D^* is of length less than j. Consider some $v \in V$ such that the longest path from v in D^* is of length j. Let $u_1,\ldots,u_k$ be the successors of v in RGVG and observe that $M(u_1),\ldots,M(u_k)$ are the sons of v in D^*. By the induction hypothesis, $R_D^*(M(u_i)) = SVAL(M(u_i)) = SVAL(u_i)$ for $i = 1,\ldots,k$. Thus $R_D^*(v)$

$= '(\theta\ R_D^*(M(u_1)) \ldots R_D^*(M(u_k)))'$
by definition of R
$= '(\theta\ SVAL(u_1) \ldots SVAL(u_k))'$
$= SVAL(v)$ by Theorem 6.1. □

Our algorithm for symbolic evaluation is given below. As in section 8, we compute SVAL and M in the order of the rank of nodes in V. The array COLOR is used to discover nodes with the same SVAL.

```
Algorithm D.
INPUT GVG = (V, E, L)
OUTPUT M and D* = (V*,E*,L*).
begin
initialize:
  M,COLOR,L* := arrays of length |V|;
  V* := E* := φ;
  Compute new labeling L' of V by Algorithm B
      of Section 5;
  Compute rank of nodes in V by Algorithm C
      of Section 7;
  for r := 0 to MAX{rank(v) | v ε V} do
    begin
      Let V_r, V̂_r be the nodes in V, V̂ of rank r;
      for all v ε V̂_r do
        if r = 0 then COLOR(v) := L'(v)
        else COLOR(v) := <L*(v),M(u_1),...,M(u_k)>
          where u_1,...,u_k are successors of v;
      radix sort nodes in V̂_r by their COLOR;
      for each maximal set Y ⊆ V̂_r containing nodes
      with the same COLOR do
        begin
          choose some u ε Y;
          add u to V*;
          comment u is made a value sink;
          if u has w_1,...,w_k as successors in GVG
          then add (u,M(w_1)),...,(u,M(w_k)) to E*;
          for all v ε Y do
            begin
              comment by Theorem 6.1,
                SVAL(u) = SVAL(v);
              M(v) := u;
            end;
        end;
      Let h be some node not in V_r;
      E_r := S_r := φ;
      for all v ε V_r-V̂_r do
        begin
          for all (v,u) ε E do
            if u ε V_r-V̂_r then
              add (u,v) to E_r;
            else if u ε V̂_r then
              add (M(u),v) to E_r;
            else add v to S_r;
          for all v ε V̂_r do add M(v) to S_r;
        end;
      Let T_r be the dominator tree of G_r' =
        (V_r ∪ {h}, E_r ∪ {(h,v)|v ε S_r}, h);
      for all sons u of h do
        begin
          comment By Lemma 8.3, u is a value sink;
          add u to V*;
          for all ancestors v of u in T_r do
            begin
              comment By Lemma 8.4,
                SVAL(u) = SVAL(v);
              M(v) := u;
            end;
        end;
    end;
  for all v ε V* do L*(v) := L'(v);
end.
```

Theorem 9.1. Algorithm D is correct and can be implemented in almost linear time.

Proof. The correctness of Algorithm D follows directly from Theorems 6.1, 6.2 and Lemmas 8.3, 8.4, and 9.1.

In addition, we must show Algorithm D can be implemented in almost linear time. The storage cost of D^* is linear in $|V|+|E|$. Initialization costs time linear in $|N| + |A|$. Algorithms B and C cost linear time by Theorems 5.1 and 7.1, respectively. The time cost of the r'th execution of the main loop, exclusive of the computation of T_r, is linear in $|V_r| + |E_r|+|\{\text{outdegree}(v)|v \in V_r-\hat{V}_r\}|$. (Here we assume that elements in the range of L' are representable in a fixed number of machine words and that the number of argument-places of function signs is bounded by a fixed constant k, so a radix sort can be used to partition $\hat{V}_r$ by COLOR.) The computation of the dominator tree T_r requires by [T3] time cost almost linear in $|V_r| + |E_r|$. Thus, the total time cost is almost linear in $|V| + |E|$. □

This completes the presentation of our algorithm. Recall that from Section 4 that with the aid of a preprocessing stage[R] costing $O((a+\ell)\alpha(a+\ell))$ bit vector operations, we may construct a GVG of size $O(da+\ell)$ where d is often of order 1 for block-structured programs but may grow to σ. (Thus this preprocessing stage offers no theoretical advantage but in practice often leads to a GVG of size linear in the program and flow graph.) In contrast to Kildall's iterative method, which for a large class of programs has storage cost $\Omega(\ell n)$ and time cost $\Omega(\ell^2 a)$, our direct method has storage cost linear in the size of the GVG and time cost almost linear in the size of the GVG. Although either method may be improved somewhat through the use of domain-specific identities, as shown in Section 3 there is in general no algorithm for computing an optimal symbolic evaluation.

In [R] these methods are extended to programs which operate on structured data in a language such as PASCAL or LISP 1.0.

References.

[AU] Aho, A.V. and Ullman, J.D., The Theory of Parsing, Translation and Compiling, Vol. II, Prentice-Hall, Englewood Cliffs, N.J., (1973).

[A] Allen, F.E., "Control flow analysis," SIGPLAN Notices, Vol. 5, Num. 7, (July 1970), pp. 1-19.

[C] Cocke, J., "Global common subexpression elimination," SIGPLAN Notices, Vol. 5, No. 7, (July 1970), pp. 20-24.

[CA] Cocke, J. and Allen, F. E., "A Catalogue of Optimization Transformations," Design and Optimization of Computers, R. Rustin, ed.), Printice-Hall, (1971), pp 1-30.

[FKU] Fong, E.A., Kam, J.B., and Ullman, J.D., "Application of lattice algebra to loop optimization," Conf. Record of the Second ACM Symp. on Principles of Programming Languages, (Jan. 1975), pp. 1-9.

[FU] Fong, E.A. and Ullman, J.D., "Induction variables in very high level languages," Conf. Record of the Second ACM Symp. on Principles of Programming Languages, (Jan. 1976), pp. 1-9.

[G] Geschke, C.M., "Global program optimizations," Carnegie-Mellon University, Phd. Thesis, Dept of Computer Science, (Oct. 1972).

[GW] Graham, S., and Wegman, M. "A fast and usually linear algorithm for global flow analysis." J. ACM, Vol. 29, No. 1, (1975), pp. 172-202.

[HU] Hecht, M.S. and Ullman, J.D., "Analysis of a simple algorithm for global flow problems," Conf. Record of the ACM Symposium on Principles of Programming Languages, (Oct. 1973), pp 207-217.

[KU1] Kam, J.B. and Ullman, J.D., "Global data flow analysis and iterative algorithms," J. ACM, Vol. 23, No. 1, (Jan. 1976), pp. 158-171.

[KU2] Kam, J.B. and Ullman, J.D., "Monotone data flow analysis frameworks," Technical Report 167, Computer Science Department, Princeton University, (Jan. 1976).

[K] Karr, M, "P-graphs," Massachusetts Computer Associates, CAID-7501-1511, (Jan. 1975).

[Ke1] Kennedy, K., "Safety of code motion," International J. Computer Math., Vol. 3, (Dec. 1971), pp. 5-15.

[Ke2] Kennedy, K., "A comparison of algorithms for global glow analysis," TR 476-093-1, Dept of Mathematical Sciences, Rice Univ., Houston, Texas, (Feb. 1974).

[Ke3] Kennedy, K., "Node listings applied to data flow analysis," Proceedings of the Second ACM Symposium on Principles of Programming Languages, (Jan. 1975), pp. 10-21.

[Ki] Kildall, G.A., "A unified approach to global program optimization," Proc. ACM Symposium on Principles of Programming Languages, (Oct. 1973), pp 194-206.

[Kn1] Knuth, D. E., The Art of Computer Programming, Vol 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., (1968).

[Kn2] Knuth, D. E., "Big omicron and big omega and big theta," SIGACT News, (Apr.-June 1976), pp 18-24.

[LF] Loveman, D. and Faneuf, R., "Program optimization -- theory and practice," Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines, (March 1975).

[M] Matijasevic, Y., "Enumerable sets are diophantine," (Russian), Dodl. Akad. Nauk SSSR 191 (1970), pp. 279-282.

[R] Reif, J. H., Ph.D. Dissertation, in progress.

[S] Schaefer, M., A Mathematical Theory of Global Analysis, Prentice-Hall, Englewood Cliffs, N.J., (1973).

[Sc] Schwartz, J.T., "Optimization of very high level languages -- value transmission and its corollaries," Computer Languages, V. 1, (1975), pp. 161-194.

[SS] Shapiro, R. and Saint, H., "The representation of algorithms," RADC, Technical Report 313, Vol., June (1972).

[T1] Tarjan, R.E., "Depth-first search and linear graph algorithms," SIAM J. Computing, Vol. 1, No. 2, (June 1972), pp. 146-160.

[T2] Tarjan, R., "Efficiency of a good but not linear set union algorithm," J. ACM, Vol. 22, (April 1975), pp 215-225.

[T3] Tarjan, R.E., "Applications of path compression on balanced trees," Stanford Computer Science Dept., Technical Report 512, (Aug. 1975).

[T4] Tarjan, R., "Solving path problems on directed graphs," Stanford Computer Science Dept., Technical report 528, (Oct 1975).

[T5] Tarjan, R., Personal communication to M. Karr, (1976).

[U] Ullman, J.D., "Fast algorithms for elimination of common subexpressions," Acta Informatica, Vol. 2, N. 3, (Dec. 1973), pp 191-213.

[W] Wegbreit, B., "The synthesis of loop predicates," Comm. ACM, Vol. 17, No. 2, (Feb. 1974), pp 102-112.