Efficient Symbolic Analysis of Programs*

JOHN H. REIF† AND HARRY R. LEWIS‡

Center for Research and Computing Technology, Harvard University, Cambridge, Massachusetts 02138

This paper is concerned with constructing, for each expression in a given program text, a symbolic expression whose value is equal to the value of the text expression for all executions of the program. A cover is a mapping from text expressions to such symbolic expressions. Covers can be used for constant propagation, code motion, and a variety of other program optimizations. Covers can also be used as an aid in symbolic program execution and for finding loop invariants for program verification. We describe a direct (non-iterative) algorithm for computing a cover. The cover computed by our algorithm is characterized as a minimum of a certain fixed point equation, and is in general a better cover than might be computed by iteration methods (which can compute fixed point covers which are not minimal). Our algorithm is efficient and applicable to all flow graphs. A variant of our algorithm is implemented by Kalman and Kortesoja (*IEEE Trans. Software Eng.* **SE-6** (1980), 512–519) in an optimizing compiler.

1. INTRODUCTION

We begin by formulating a global flow model for a computer program to which we wish to apply various optimizations as in [H] and [MJ].

1.1. *The Global Flow Model*

All intraprogram control flow is reduced to a digraph 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). The *control flow graph* $F=(N, A, s)$ is a flow graph whose nodes are called *blocks* (to distinguish it from other graphs considered in our paper) and rooted at the *start* dis-

* A preliminary draft of this paper appeared as Symbolic evaluation and the global value graph, *in* "Proceedings, 4th ACM Symposium on Principles of Programming Languages," 1977.

† Supported by National Science Foundation Grant NSF-MCS79-21024 and by Office of Naval Research Contract N00014-80-C-0647.

‡ Supported by National Science Foundation Grants NSF-MCS76-09375 and NSF-MCS80-05386.

Preprint of paper appearing in *Journal of Computer and System Sciences*, Vol. 32, No. 3, June 1986, pp. 280-314.

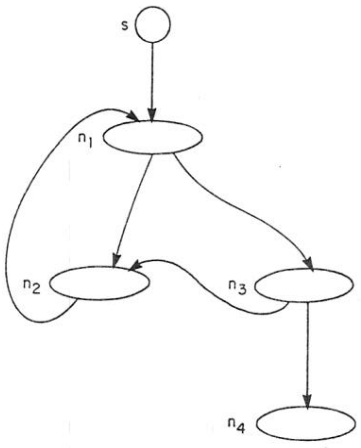

FIG. 1. A program's control flow graph.

tinguished block $s \in N$. A *control path* is a path in F. Executions of the program correspond to control paths beginning at the start blocks, although not every such path in this graph need correspond to a possible execution of the program. (See Fig. 1.)

The only statements in the programming language retained in the model are assignment statements. An *assignment statement* is of the form $X := \mathscr{E}$. The left-hand side of the assignment is a program variable taken from the set $\{X, Y, Z, \ldots\}$. The right-hand side is an expression $\mathscr{E}$ built from program variables and fixed sets C of *constant symbols* and θ of *function symbols.*

Each node $n \in N$ contains a block of assignment statements. These blocks do not contain conditional or branch statements; control information is specified by the control flow graph as in [C]. A program variable occurring within only a single block $n \in N$ is *local to n.* Let Σ be the set of program variables not local to any block. For each program variable $X \in \Sigma$ and block $n \in N - \{s\}$ we introduce as in [RT] the *input variable* X^n to denote the value of X on *entry* to block n. We use the symbol X^s, considered to be a constant symbol, to denote the value of X on entry to the program at the start block s.

Let EXP be the set of expressions built from input variables, C, θ. Thus, $\mathscr{E} \in \text{EXP}$ is a finite expression consisting of either a constant symbol $c \in C$, an input variable X^n representing the value of program variable X^n on input to block n, or a k-adic function symbol $\theta \in \Theta$ prefixed to a k-tuple of expressings in EXP. (Note: In the standard terminology of mathematical logic, $\mathscr{E}$ is a term in a first order language; it is an expression containing no predicates and built from function symbols, constant symbols, and variables on input to particular blocks of assignment statements.)

For each $X \in \Sigma$ and node $n \in N$ where X is assigned to, let the *output expression* $\mathscr{E}(X, n)$ be an expression in EXP for the value of X on exit from block n in terms of constants and input variables at block n. A *text expression* t is an output expression

or a subexpression of an output expression. Note that each text expression t is a substitution instance of an expression on the right-hand side of an assignment statement of the program.

For example, let n be the block of code

$$X := X - 1;$$
$$Y := Y + 4;$$
$$Z := X * Y.$$

Then $\mathscr{E}(Z, n) = (X^n - 1) * (Y^n + 4)$ (or in the more proper prefix notation, $(*(-X^n\ 1)(+Y^n\ 4))$) is the text expression associated with the string of text "$X*Y$" at the last assignment statement of n.

An *interpretation* for a program is an ordered pair (U, I). The *universe* U contains (among other things) a distinct value $I(c)$ for each constant symbol $c \in C$. For each k-adic function symbol $\theta \in \Theta$, there is a unique *k-adic operator* $I(\theta)$ which is a partial mapping from k-tuples in U^k into U. We assume $I(c_1) \neq I(c_2)$ for each distinct $c_1, c_2 \in C$ (every value has at most one name). For example, a program is in the *arithmetic domain* if it has the interpretation (Z, I_Z), where Z is the set of integers and I_Z maps symbols $+, -, *, /$ to the arithmetic operations addition, subtraction, multiplication, and integer division.

An expression in EXP is put in *reduced form* by repeatedly substituting for each subexpression of the form $(\theta\ c_1 \cdots c_k)$ that contant symbol c such that $I(c) = I(\theta)(I(c_1),..., I(c_k))$, until no further substitutions of this kind can be made. (Using the techniques of Aho and Ullman [AU1] we would in linear time reduce each block, so each text expression is a reduced expression. Note that this only results in the detection of certain locally redundant expressions; we shall describe a method for global detection of redundant expressions.)

A *global flow system* ρ is a quadruple (F, Σ, U, I), where F is the control flow graph, Σ is the set of program variables, and (U, I) is an interpretation. The next definitions deal with a fixed global flow system $\rho = (P, \Sigma, U, I)$.

1.2. *Covers*

The utility of the global flow model is that many program analysis and improvement problems may be formulated as combinatorial problems on digraphs. The fundamental program analysis problem of interest here is the discovery, for each text expression t, of a symbolic expression $\mathscr{E}$ for the value of t which holds for all executions of the program.

Let $\mathscr{E}$ be an expression in EXP and let p be a control path. We give a recursive definition for VALUE$(\mathscr{E}, p)$, the expression for the value of $\mathscr{E}$ in the context of a program execution on this control path p. VALUE$(\mathscr{E}, p)$ is defined formally as follows:

(i) If $p = (s)$ then VALUE(E, p) is the reduced expression derived from $\mathscr{E}$.

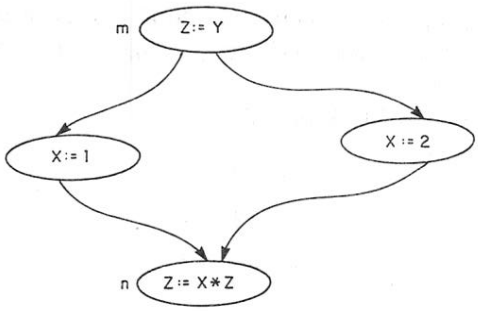

FIG. 2. A minimal fixed point of μ covers $\mathscr{E}(Z, n)$ with the expression $X^n * Y^m$.

(ii) Otherwise, if $p = p \cdot (m, n)$ then $\text{VALUE}(\mathscr{E}, p) = \text{VALUE}(\mathscr{E}', p')$, where $\mathscr{E}'$ is the expression obtained from $\mathscr{E}$ by substituting the output expression $\mathscr{E}(X, m)$ for each input variable X^n, and putting the result in reduced form.

We now define origin($\mathscr{E}$), where $\mathscr{E} \in \text{EXP}$, which intuitively is the earliest block at which all the quantities referred to in $\mathscr{E}$ are defined. Let $N(\mathscr{E}) = \{n \in N \mid \text{the input variable } X^n \text{ occurs in } \mathscr{E}\}$. If $N(\mathscr{E})$ is empty then origin($\mathscr{E}$) is the block s and otherwise origin($\mathscr{E}$) is the latest (i.e., furthest block from s) block in $N(\mathscr{E})$ relative to the dominator ordering (see Appendix I). The origin need not exist for arbitrary expressions in EXP, but will be well defined in all the relevant cases (i.e., origin exists for all text expressions and their covers). Note that if a text expression t contains no input variables then origin(t) $= s$, and otherwise origin(t) is the block in N where an assignment statement to a variable in $N(\mathscr{E})$ is located.

An expression $\mathscr{E} \in \text{EXP}$ *covers* a text expression t if $\text{VALUE}(t, p) = \text{VALUE}(\mathscr{E}, p)$ for every control path p from s to origin(t). Hence, if $\mathscr{E}$ covers t then $\mathscr{E}$ correctly represents the value of t on every execution of program Π. (See Fig. 2)

A *cover* is a mapping ψ from text expressions to expressions in EXP in reduced form such that for each text expression t, $\psi(t)$ covers t. Note that the origin of any cover $\mathscr{E}$ of a text expression t is always well defined since the elements of $N(\mathscr{E})$ will form a chain relative to the dominator ordering.

LEMMA 1. *If* $\mathscr{E} \in \text{EXP}$ *covers text expression* t *then* origin($\mathscr{E}$) *dominates origin*(t).

Proof (by contradiction). Suppose origin($\mathscr{E}$) does not dominate origin(t). Then $\mathscr{E}$ must contain an input variable X^n such that n is not a dominator of origin(t). Hence, there is an n-avoiding control path p from the start block s to origin(t) such that $\text{VALUE}(\mathscr{E}, p)$ contains X^n but $\text{VALUE}(t, p)$ does not, so $\text{VALUE}(\mathscr{E}, p) \neq \text{VALUE}(t, p)$, contradicting the assumption that $\mathscr{E}$ covers t. ∎

We now define a partial ordering of covers. For each pair of covers ψ_1 and ψ_2, $\psi_1 \leqslant \psi_2$ iff origin($\psi_1(t)$) dominates origin($\psi_2(t)$) for all text expressions t.

We wish to compute a cover minimal with respect to this partial ordering. Unfortunately, Appendix II shows this is an undecidable problem. It follows that we must

look for heuristic methods for good, but not minimal covers. Subsection 1.4 defines a class of covers which are fixed points of an iterative process. The minimal fixed point cover is efficiently computed by our direct algorithm given in Section 2. The next subsection describes applications of covers to program optimization.

1.3. *Applications of Covers*

We give below a number of program analysis problems and optimizations which reduce to the problem of determining covers of text expressions. These examples indicate that computing covers is of fundamental importance to program analysis. [R1] and the paper of [RT] were the first to consider the problem of computing covers. [KK] have made practical application of our work in the implementation of an optimizing computer for PASCAL.

(a) *Constant propagation* (or *folding*) is the substitution of the appropriate constant symbols for text expressions covered by constants (see [Ki]). (Although constant propagaton is useful in itself as a program optimization, it happens to be only the first step of our procedure for computing covers.)

(b) More generally, a text expression t located at block n is *redundant* if on all paths from the start block to n another text expression t' yields a computation equivalent to that of t. Thus t may be replaced by a load operation from a temporary address containing the result of some such equivalent previous computation (see [C; CA; E; G; FKU; U]). Thus it would suffice that each such t has the same cover as t.

(c) *Code motion* is the process of moving code as far as possible out of cycles in the control flow graph (i.e., out of program loops). The *birth point* of text expression t is the earliest block n in the control flow graph (relative to the partial ordering of blocks by domination with the start block first), where the computation of t is defined. Any block occurring between (relative to this domination ordering) n and the original location of t has a cover for t in terms of covers for the variables at n. This best possible birth point for t is the origin of the minimal covering expression for t. Hence, code motion is fundamentally related to the computation of covers. The earliest such block, m, with the further property that the computation of t can induce no new errors at that block m, is called the *safe point* of t; the computation of t may safely be moved to any block between m and $\mathrm{loc}(t)$. The text expression appropriate at the chosen block may not be lexically identical to t, but is given by the cover of t in terms of the variables on input to that block. Preliminary work on simple motions, primarily emphasizing safety, but not considering birth points is given in [CA; G; E]. [R2] gives a complete formulation of code motions considering birth points and safepoints, also considering the movement as far as possible out of cycles, and gives an efficient algorithm for carrying out these code motion optimizations.

(d) A cover for a variable in a program loop is a *loop invariant* (see [FU;

W]. The discovery of loop invariants is often crucial for proving the correctness of a program; see, for example, [U1; KM; HK].

(e) *Symbolic execution* of a program as described in [K2; CHT] and a *program transformation* as described in [L; SHKN] generally requires a powerful program *simplifier*. Domain specific simplifiers such as in [NO] may require the solution of logical decision problems which require much time and space. The covers give domain independent simplifications of program text, which can be computed efficiently. A practical simplification system may use a combination of these techniques.

1.4. *A Computible Class of Covers*

In Appendix II we show that the problem of computing minimal covers over arithmetic domains is undecidable. Here we consider a class of covers that can be characterized by fixed point equations. These covers can be computed inefficiently by an iterative algorithm (later in this paper we describe how to efficiently compute them by our direct algorithm). To iteratively construct this class of covers, we would first taken a pass through the program and construct a mapping ψ_0 from text expressions to EXP; ψ_0 may not be a cover but has the property that for all text expressions t,

$$\text{VALUE}(\psi_0(t), p) = \text{VALUE}(t, p)$$

for some (rather than *all*) control paths p from s to origin(t). The algorithm would then iteratively compare possible covering expressions of input variables at particular blocks to the corresponding output expressions of preceding blocks, and propagate the results to predecessor blocks. More precisely, for any mapping ψ from text expressions to EXP, let $\mu(\psi)$ be the mapping ψ' from text expressions to EXP such that for each input variable X^n,

$$\psi'(X^n) = \mathscr{E} \quad \text{if } \mathscr{E} = \psi(\mathscr{E}(X, m)) \text{ for all blocks } m \text{ immediately preceding } n \text{ in the control flow graph } F,$$

$$= X^n \quad \text{otherwise,}$$

and $\psi'(t)$ is the reduced expression derived from text expression t after substituting $\psi'(X^n)$ for each input variable X^n occurring in t. This iterative algorithm then computes $\mu^k(\psi_0)$ for $k = 1, 2, \ldots$, until a fixed point of μ is obtained. Fig. 2 gives an example of a minimal fixed point cover. We shall show that the resulting fixed point ψ is a cover; however, the simple example given in Fig. 3 shows that ψ is not necessarily a minimal fixed point cover.

THEOREM 1. *If ψ is a fixed point of μ then ψ is a cover.*

Proof. We must show $\text{VALUE}(\psi(t), p) = \text{VALUE}(t, p)$ for all text expressions t and control paths p from s to the block where t is located. Let p be the shortest

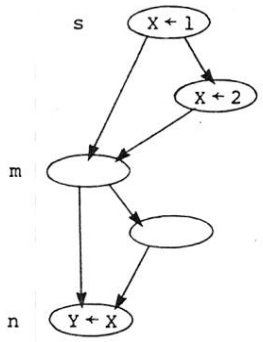

FIG. 3. X^m, the value of X on input to m, covers X^n, the value of X on input to n.

control path from s to a block n where there is located a text expression t such that

$$\text{VALUE}(\psi(t), p) \neq \text{VALUE}(t, p).$$

Thus t must contain an input variable X^n such that

$$\text{VALUE}(\psi(X^n), p) \neq \text{VALUE}(X^n, p).$$

Clearly, $\psi(X^n) \neq X^n$. Let m be the next to last block in p, so $p = p' \cdot (m, n)$. By definition of Ψ, $\psi(X^n) = \psi(\mathscr{E}(X, m))$. Since $\psi(X^n)$ contains no input variables at n,

$$\begin{aligned}
\text{VALUE}(\psi(X^n), p) &= \text{VALUE}(\psi(X^n), p') \\
&= \text{VALUE}(\psi(\mathscr{E}(X, m)), p') && \text{since } \psi(X^n) = \psi(\mathscr{E}(X, m)), \\
&= \text{VALUE}(\mathscr{E}(X, m), p') && \text{by the induction hypothesis,} \\
&= \text{VALUE}(X^n, p) && \text{by definition of VALUE.} \quad \blacksquare
\end{aligned}$$

In Appendix III, we show that μ has a unique minimal fixed point ψ^*. (See Figs. 2 and 3 for examples of the minimal fixed point cover.) We then show how to efficiently compute ψ^*.

The overall plan of Section 2 is to introduce (in Sect. 2.1) a special class of graphs called *global value graphs* which represent the flow of *values* (rather than *control*) through the program. We define, for each global value graph GVG, a set Γ_{GVG} of approximate covers associated with it. Appendix III shows Γ_{GVG} is in each case a finite semilattice which thus has a unique minimal element $\min(\Gamma_{\text{GVG}})$, and which is efficiently calculated by the algorithm presented in Sections 2.2–2.5. As we show in Appendix III, for a particular choice of $\text{GVG} = \text{GVG}_0$, $\min(\Gamma_{\text{GVG}_0})$ is actually ψ^*, the minimal fixed point of the functional μ, so our general algorithm does indeed compute ψ^*.

1.5. *Comparison with Previous Work*

In order to compare our methods with others we must fix the relevant parameters of the program and control flow graph. Let n and a be the cardinality of

the node and edge sets, respectively, of the control flow graph. Let σ be the number of variables occurring within more than one block of the program (if we build into the programming language a construct for the declaration of variables local to a block, then the parameter σ is the number of *global* variables). Let l be *length* of the program text, that is, the total number of subexpressions, assignments, and control statements occurring in the program text. Previous authors have analyzed program optimization algorithms primarily from the point of view of the control flow graph parameters n and a, without taking into proper account the general case where l is significantly longer than these parameters.

Kildall [Ki] presents an iterative algorithm for computing approximate solutions to various expression optimization problems including constant detection. Since Kildall's algorithm requires that the value of each global variable be approximated at each node, and since these values are propagated across each edge, it follows that each iteration of Kildall's algorithm takes $\Omega(l+\sigma(n+a))$ elementary steps. Each iteration of Kildall's algorithm may result only in the change of only one program variable value (out of σ program variables) at only a single block (out of n blocks). A total of n iterations of Kildall's algorithm for constant detection can thus be required for convergence. Thus the total time in the worse case is at least $\Omega((l+\sigma(n+a))\, n)$. ($\Omega(f(x))$ is a function bounded from *below* by $k \cdot f(x)$ for some constant k. See Knuth [Kn2].) No previous papers considered the more general problem of computing covers of text expressions.

As described in Section 1.4 an iterative algorithm may also be used to compute a certain class of covers, which we have characterized as fixed points of an update functional Ψ, mapping approximate covers to improved covers. Fong, Kam, and Ullman [KFU] give a direct (noniterative) method for solving various expression optimization problems such as constant detection. They do not consider the computation of covers, but their method could be adapted to give covers. However, these resulting covers would be weaker than our fixed point covers and their direct algorithm is restricted to reducible flow graphs. The iterative algorithm requires $\Omega(ln^2)$ elementary steps and Fong, Kam, and Ullman's algorithm requires $\Omega(la \log(a))$ elementary steps. One source of inefficiency of both of these algorithms is in the representation of the covers. Directed acyclic graphs (dags) are used to represent expressions, but separate dags are needed at each node of the flow graph. Since a dag representing a cover may be of size $\Omega(l)$, the total space cost may be $\Omega(ln)$. Various operations on these dags, which are considered to be "extended" steps by Fong, Kam, and Ullman [FKU], cost $\Omega(l)$ elementary steps and cannot be implemented by any fixed number of bit vector operations. In general, any similar algorithm for computing a cover which attempts to pool information separately at each node of the flow graph will have time cost of $\Omega(la)$, since the pools on every pair of adjacent nodes must be compared. Since $l \geqslant n$, such a time cost may be unacceptable for practical applications.

Another problem with these previous methods is they do not necessarily compute good covers. The iterative algorithm only computes a fixed point of Ψ, but not necessarily its minimal fixed point. Soms of the difficulties of computing covers is

illustrated in Fig. 3. Note that X^m covers X^n, and this is discovered by our algorithm since the minimal fixed point of X^n is X^m. In fact, our algorithm always gives the minimal fixed point. While iterative algorithms might be modified to do well for specific examples such as in Fig. 3, it remains an open question (open for at least 6 years) whether some efficient iterative method can provably compute the minimal fixed point cover. At any rate, this paper was the first in the literature directly concerned with computing covers.

The *global value graphs* used in this paper contain dags of program blocks as well as the use-def edges of [Sc] to represent the global flow of values through the program. The use of a global value graph leads to our efficient direct algorithm for computing covers which works for all flow graphs. The method derives its efficiency by representing the covers with a single dag, rather than a separate dag at each node. The global value graph GVG_0 is of size $O(\sigma a + l)$, although the results of [RT] may be used to build a global value graph which in many cases is of size $O(a + l)$ (see Sect. 3). In elementary operations the time cost of our algorithm for the discovery of constants is linear in the size of GVG, and our algorithm for finding the cover which is the minimal fixed point of Ψ requires time almost linear in the size of the GVG. Thus our algorithm for symbolic evaluation takes worst case time almost linear in $\sigma a + l$ ($a + l$ in many cases), as compared to the iterative algorithm which may require $\Omega(ln^2)$ steps. Recently, Reif and Tarjan [RT] give an algorithm which computes simple covers (weaker than minimal fixed points of Ψ) in time almost linear in $l + n + a$. This algorithm also uses a single dag for representing the simple cover and works for all flow graphs.

1.6. *Further Work*

Reif, [R1] extends our algorithm to symbolic analysis of programs with records, such as LISP and PASCAL programs. Wegman and Zadeck [WZ] recently gave a very interesting extension of our constant propagation algorithm to utilize information from conditional branches.

2. An Efficient Algorithm for Computing a Cover

2.1. *Dags and Global Value Graphs*

A *labeled dag* $D = (V, E, L)$ is a labeled, acyclic, oriented digraph 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. A rooted labeled dag (D, r) *represents* an expression $\mathscr{E}$ if $\mathscr{E}$ is the parenthesized listing of the lables of the subgraph of D rooted at r in topological order (see Appendix I for definition of such a topological ordering) from r to the leaves and from left to right among immediate successors. When D is fixed, we simply say r *represents* $\mathscr{E}$ if (D, r) so represents $\mathscr{E}$. (See Fig. 4.)

The dag D is *minimal* if each node $r \in V$ represents a distinct expression. Any expression or set of expressions may be represented, with no redundancy, by a

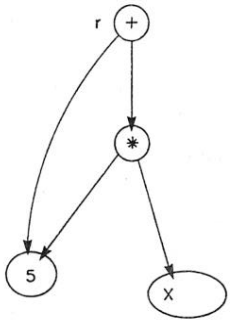

FIG. 4. (D, r) represents $(5+(5*X^n))$ (or more properly in prefix notation $(+5(*5X^n))$), where D is the above dag.

minimal dag $D(n)$ to represent efficiently the set of text expressions located at block n. We have assumed that each block is reduced, so each node in $D(n)$ corresponds to a unique text expression. [AU1] describe the use of dags for representing computations within blocks. [Ki] and FKU] have applied dags to various global flow problems.

We now come to the central definition. To model the flow of values through a program, we introduce a class of labeled digraphs called *global value graphs*. These are derived by combining the dags of all the blocks in N and adding a set of edges called use-def edges (which pair nodes labeled with input variables to other nodes). More precisely, a global value graph is a possibly cyclic, labeled, oriented digraph $\mathrm{GVG} = (V, E, L)$ such that:

(1) The node set V is the union of the node sets of the dags of N.

(2) E is an edge list containing (a) the edge list of each $D(n)$ and (b) a set of pairs in V^2 (use-def edges) such that (i) the first node of each use-def edge is labeled with an input variable and (ii) for each $v \in V$ labeled with an input variable X^n, and control path p from s to n, there is some use-def edge departing from v and entering a node located at a block in p and distinct from n. (Specific use-def edge sets will be used for various global value graphs considered in this paper.)

(3) L is a labeling of V identical to the vertex labeling of each $D(n)$.

Note that for each $v \in V$, if v represents a constant symbol c then v is labeled with c and has no departing edges; if v represents a function application $(\theta\, t_1 \cdots t_k)$ then v is labeled with the k-adic function symbol θ and $u_1, \ldots, u_k$ are the immediate successors of v in GVG representing $t_1, \ldots, t_k$, respectively; if v represents an input variable X^n then v is labeled with X^n and all the edges departing from v are use-def edges. For each node $v \in V$, let $\mathrm{loc}(v)$ be the block in N where the text expression which v represents is located.

We assume here, as in Section 1, that the set of text expressions of each block $n \in N$ includes all input variables at n. This may require adding dummy assignments of the form $X := X$ to satisfy this assumption. Let Γ_{GVG} be the set of mappings ψ from V to EXP such that for all $v \in V$

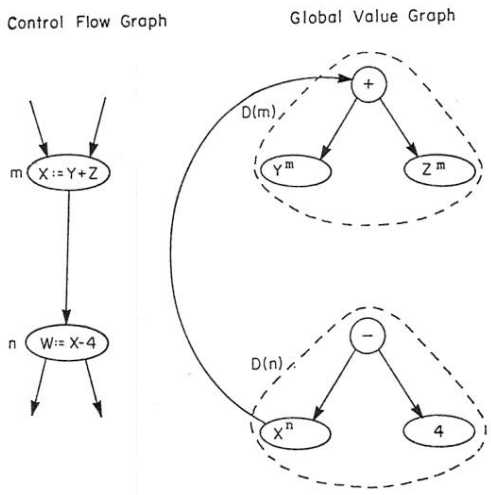

FIG. 5. The program's global value graph GVG_0.

(1) if $L(v)$ is a constant symbol c then $\psi(v) = c$, or

(2) if $L(v)$ is a function symbol θ and v has immediate successors $u_1, \ldots, u_k$ (in this order) then $\psi(v)$ is the reduced expression derived from $(\theta\psi(u_1)\cdots\psi(u_k))$, or

(3) if $L(v)$ is an input variable then either (a) $\psi(v) = L(v)$ or (b) $\psi(v) = \psi(u)$ for all use-def edges (v, u) departing from v.

Note that for any node v satisfying (2), $\psi(v)$ is determining from the input variables occurring in the text expression which v represents. Hence any $\psi \in \Gamma_{GVG}$ is uniquely specified by the set of input variables satisfying case (3a). In Appendix III we show that Γ_{GVG} is a finite semilattice, and hence has a minimal element.

Let GVG_0 be the *standard* global value graph containing only the use-def edges. $\{(v, u) | v$ represents input variable X^n and u represents the output expression $\mathscr{E}(X, m)$ for each program variable $X \in \Sigma$ and edge $(m, n) \in A$ of the control flow graph $F.\}$ (See Fig. 5.) Computing this set of use-def edges is easy, since the set of text expressions of each block $n \in N$ includes all input variables at n. Note that while there are in the worst case ln possible use-def edges, GVG_0 contains at most $l\sigma$ use-def edges. (Sect. 3.1 defines a global value graph using a somewhat different definition for use-def edges, which is even more efficient.) Let ψ^* be the minimal fixed point of μ, the functional defined in Section 1.4. Appendix III shows ψ^* identical to be the minimal element of Γ_{GVG} applied to the standard global value graph GVG_0. (Also, in Sect. 3 we define a global value graph GVG_1 with the same property, but which often is of size linear in $l + a$.)

2.2. *Detection of Constants*

Let $GVG = (V, E, L)$ be an arbitrary global value graph. Let ψ be a minimal element of Γ_{GVG}. We wish to compute a new labeling L' of vertices in V such that

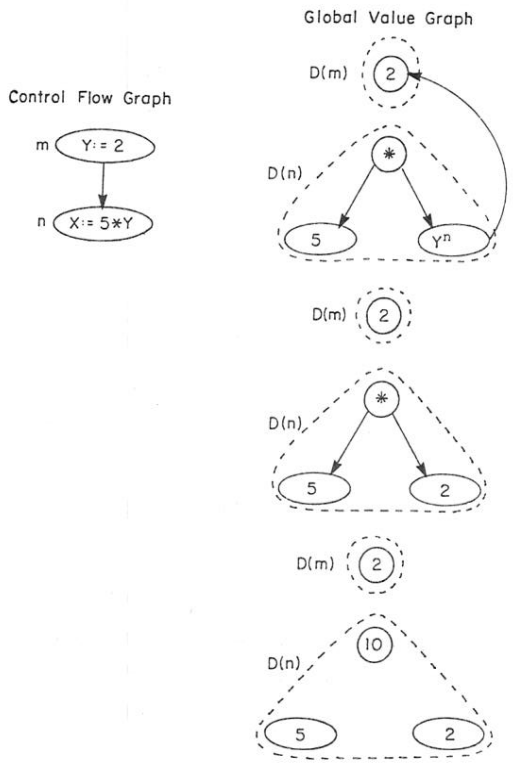

FIG. 6. A simple example of constant propagation through the global value graph.

for each $v \in V$, if $\psi(v)$ is a constant sign then $L'(v) = c$ and otherwise $L'(v) = L(v)$. The new labelling can be discovered by propagating possible constants through GVG, starting from nodes originally leveled with constants and then testing for conflicts. This is an algorithm for constant propagation with time cost linear in the size of the GVG. (See Fig. 6 for a simple example of constant propagation through the GVG.)

Recall that a *spanning tree* of the control flow graph $F = (N, A, s)$ is a tree rooted at s with node set N and edge set contained in A. A *preordering* of a tree orders fathers before sons. Let $<$ be a preordering of some spanning tree of F. For each $v \in V$, let $\text{loc}(v)$ be the node in N at which the text expression associated with V is located. We construct an acyclic subgraph of GVG by deleting the set of use-def edges $\bar{E} = \{(v, u) | \text{loc}(v) \leqslant \text{loc}(u)\}$. Observe that $(V, E - \bar{E})$ is acyclic. We shall propagate constants in a topological order (see Appendix I for definition) of $(V, E - \bar{E})$, from leaves to roots. (See Fig. 6).

Our algorithm for computing the new labeling L' is given below. In our initial constant propagation phase at the do loop at label L0 we ignore the fact that there can be successors of v in GVG that do not precede v in the topological ordering of

$(V, E-\bar{E})$. However, we take these ignored edges into account in the later portion of the algorithm following label L4, by resetting $L'(v)$ to $L(v)$ if v is discovered not to be constant via these ignored edges.

ALGORITHM A.

Input: global value graphs GVG $= (V, E, L)$ and control flow graph F.

output: L'.

```
begin
    declare L' to be an array of length |V|;
    Let < be a preordering of a spanning tree of F;
    Q := Ē := the empty set { };
    for all use-def edges (v, u) ∈ E such that loc(v) ⩽ loc(u)
        do add (v, u) to Ē od;
    comment propagate constants;
L0: for each v ∈ V in topological order of (V, E − Ē)
    from leaves to roots do
        if L(v) is a constant sign c then L1: L'(v) := c;
        else if L(v) is a k-adic function symbol θ,
            u_1,..., u_k are the immediate successors of v in
            GVG, and (θ L'(u_1) ··· L'(u_k)) reduces to a
            constant c then L2: L'(v) := c;
            else if L(v) is an input variable and there
                is a constant c such that L'(u) = c
                for all use-def edges (v, u) ∈ E − Ē departing from v
                then L3: L'(v) := c;
            else add v to Q; L'(v) := L(v) fi;
            fi;
        fi;
    od;
    comment test for conflicts;
L4: for each v ∈ V labeled with an input variable do
        if v has a departing use-def edge (v, u) ∈ E such that
        L'(v) ≠ L'(u) then add v to Q fi;
    till Q = the empty set { } do
        delete some node v from Q;
        if L'(v) is a constant then
L5: L'(v) := L(v);
        add all immediate predecessors of v in GVG to Q;
        fi
    od;
end.
```

LEMMA 2.1. *If $\psi(v)$ is a constant then $L'(v)$ is set to $\psi(v)$ at L1, L2, or L3.*

Proof (by induction on the topological order of $(V, E-\bar{E})$).

Basis step. Suppose $v \in V$ is a leaf of $(V, E-\bar{E})$. Then $L(v)$ is a constant sign and so $L'(v)$ is set to $L(v)=\psi(v)$ at L1.

Induction step. Suppose $v \in V$ is not a leaf of $(V, E-\bar{E})$ and $L'(u)$ has been set to $\psi(u)$ for all u occurring before v in the topological order where $\psi(u)$ is a constant. Then v represents either a function application or an input variable.

Case 1. Suppose $L(v)$ is a k-adic function sign θ and $u_1,..., u_k$ are the immediate successors of v in $(V, E-\bar{E})$. If $\psi(v)$ is a constant c then by definition of Γ, $\psi(u_1),..., \psi(u_k)$ are constants $c_1,..., c_k$, respectively, and $(\theta\, c_1 \cdots c_k)$ reduces to c. By the induction hypothesis $L'(u_1),..., L'(u_k)$ have been previously set to $c_1,..., c_k$ and so $L'(v)$ is set to $\psi(v)=c$ at L2.

Case 2. Otherwise, $L(v)$ is an input variable X^n. If $\psi(v)$ is a constant symbol c then $\psi(v) \neq X^n$, so by definition of Γ_{GVG}, $c=\psi(u)$ for all use-def edges $(v, u) \in E$ departing from v. By the induction hypothesis, $L'(u)$ has been set to $c=\psi(u)$ for each use-def edge $(v, u) \in E-\bar{E}$. Now we must show v has some departing use-def edge $(v, u) \in E-\bar{E}$. Let T be the spanning tree of F with preorder $<$. Consider the path p in T from the start block s to n. By definition of GVG, there is a use-def edge (v, u) such that $\mathrm{loc}(u)$ is distinct from n and is contained in p. Hence $(v, u) \in E-\bar{E}$ and $L(v)$ is set to c at L3. ∎

Let $\bar{Q}$ be the value of Q just after L4. Then $v \in V$ is eventually added to Q and $L'(v)$ reset to $L(v)$ iff some element of $\bar{Q}$ is reachable in GVG from v. If $v \in V$ is labeled by L' with a constant at L4, then we show

LEMMA 2.2. *$\psi(v)$ is not a constant iff some element of $\bar{Q}$ is reachable in* GVG *from v.*

Proof. (*If*) Suppose $\psi(v)$ is not a constant, but no element of $\bar{Q}$ is reachable from v. Then let $\bar{\psi}$ be the mapping from V to EXP such that for each $u \in V$, $\bar{\psi}(u)$ is the reduced expression derived from $\psi(u)$ after substituting $\psi(w)$ for each input variable represented by a node w (i.e., w is the unique node labeled with that input variable) from which an element of $\bar{Q}$ is reachable. Then $\bar{\psi} \in \Gamma_{\mathrm{GVG}}$ but $\mathrm{origin}(\bar{\psi}(v))=s \leqslant \mathrm{origin}(\psi(v))$, contradicting the assumption that ψ is the minimal element of Γ_{GVG}.

(*Only If*) Suppose some element of $\bar{Q}$ is reachable from v in GVG. Clearly if $v \in \bar{Q}$, then $\psi(v)$ is not a constant. Assume for some $k>0$, if there is a path of length less than k in GVG from some $u \in V$ to an element of $\bar{Q}$, then $\psi(u)$ is not a constant sign. Suppose there is a path $(v=w_0, w_1,..., w_k)$ of length k from v to $w_k \in \bar{Q}$. If $k=1$, then $w_1 \in \bar{Q}$, and otherwise if $k>1$, then $(w_1,..., w_k)$ is a path of length $k-1$. By the induction hypothesis, $\psi(w_1)$ is not a constant. But $(v, w_1) \in E$ and by the definition of Γ_{GVG}, $\psi(v)$ is not a constant. ∎

THEOREM 2.1. *Algorithm* A *is correct and has time cost linear in the size of the* GVG.

Proof. The correctness of Algorithm A follows directly from Lemmas 2.1 and 2.2.

In addition we must show Algorithm A has time cost linear in $|V|+|E|$. The initialization costs time linear in $|V|$. The preordering $<$ 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 steps L0 and L4 is $O(1+\text{outdegree}(v))$. Step L5 can be reached at most $|V|$ times and the time cost to process each node v at step L5 is $O(1+\text{indegree}(v))$. Thus, the total time cost is linear in $|v|+|E|$. ∎

In some cases, we may improve the power of Algorithm A 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 modify Algorithm A so that if node v is labeled by L with the multiplication symbol and a successor of v in GVG is covered by 0, then at step L3 we may set $L'(v)$ to the constant 0.

From the new labeling L' and $\text{GVG} = (V, E, L)$, we construct a *reduced global value graph* $\text{GVG}' = (V, E', L')$ with labeling L' and with edge set E' derived from E by deleting all edges departing from nodes labeled by L' with constant symbols. This corresponds to substituting constant symbols for constant text expressions in the program. We assume throughout the next three sections that *GVG* is so reduced.

2.3. *A Partial Characterization of ψ, the Minimal Element of Γ_{GVG}*

Let $\text{GVG} = (V, E, L)$ be a reduced global value graph as constructed by Algorithm A of the last section. Let ψ be the minimal element of Γ_{GVG}. Let $\hat{V}$ be the set of nodes in V labeled with constant and function symbols. Observe that Γ_{GVG} characterized exactly the values of any such ψ over nodes in $\hat{V}$ in terms of the values of ψ over the nodes in $V - \hat{V}$, i.e., in terms of the nodes labeled with input variables. The following theorem characterizes ψ over $V - \hat{V}$ in terms of ψ over $\hat{V}$.

We require first a few additional definitions. A *use-def path* is a path p in GVG traversing only nodes linked by use-def edges. A use-def path is *maximal* if the last node of p has no departing use-def edges. For any node $v \in V$ labeled with an input variable, let $H(V)$ be the set of nodes in V lying at the end of a maximal use-def path from v. Note that $H(v)$ is a subset of $\hat{V}$. Call two paths *disjoint* if they have only their initial node in common.

THEOREM 2.2. *If v is labeled with an input variable, then either*

(a) $\psi(v) = \psi(u)$ *for all* $u \in H(v)$, *or*

(b) $\psi(v) = L(u)$, *where u is the unique node such that*

(i) *u lies on all maximal use-def paths from v but*

(ii) *there are disjoint maximal use-def paths from u to nodes* $u_1, u_2 \in H(v)$ *such that* $\psi(u_1) \neq \psi(u_2)$. *(See Fig. 7.)*

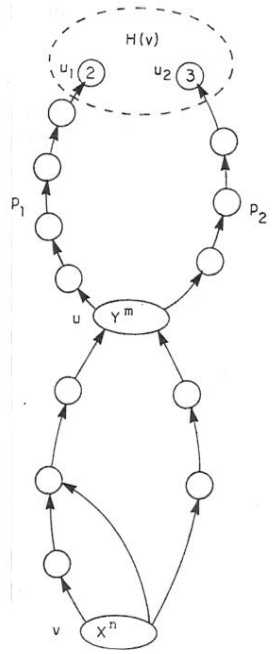

FIG. 7. Case (b) of Theorem 2.2: All maximal use-def paths from v contain u and p_1, p_2 are disjoint maximal use-def paths from u to u_1, $u_2 \in H(v)$.

Proof. Suppose $\psi(v)$ is *not* an input variable, so there exists a maximal use-def path p from v to some $u_1 \in H(v)$ such that $\psi(v) = \psi(u_1)$. Assume there exists another maximal use-def path p' from v to some $u_2 \in H(v)$ such that $\psi(v) \neq \psi(u_2)$. Let z be the first element of p' such that $\psi(z) \neq \psi(u_2)$ and let z' be the immediate predecessor of z in p', so $\psi(z') = \psi(v)$. Then by definition of Γ_{GVG}, $\psi(v) = \psi(z') = L(z')$ is an input variable. We use a proof by contradiction.

Suppose $\psi(v)$ is an input variable, so $\psi(v) = L(u)$ for some $u \in V$. For any maximal use-def path p from v, let z be the first element of p such that $\psi(z) \neq L(u)$ and let z' be the immediate predecessor of z in p. Then by definition of Γ_{GVG}, $\psi(z') = L(z') = L(u)$ so $z' = u$ is contained on p. Now suppose that there is a node $w \in V$ distinct from u and contained on all maximal use-def paths from u.

Consider any control path q from the start block s to block $\text{loc}(u)$. By Lemma 2.3, we can construct a maximal use-def path $(u = w_1, \ldots, w_k)$ such that $\text{loc}(w_1), \ldots, \text{loc}(w_k)$ are distinct blocks in q. Hence, $\text{loc}(w)$ properly dominates $\text{loc}(u)$.

Let ψ' be the mapping from V to EXP such that for all $v' \in V$, $\psi'(v')$ is derived from $\psi(v')$ by substituting $L(w)$ for each input variable laeling a node from which all maximal use-def paths contain w. Then $\psi' \in \Gamma_{\text{GVG}}$. But $\text{origin}(\psi'(v)) = \text{loc}(w)$ properly dominates $\text{loc}(u) = \text{origin}(\psi(v))$, contradicting our assumption that ψ is minimal over Γ_{GVG}. ∎

Theorem 2.2 suggests a procedure for calculating ψ, but there is an implicit circularity since the calculation (using Theorem 2.2) of $\psi(v)$ for $v \in V - \hat{V}$ requires the determination (using the definition of Γ_{GVG}) of $\psi(u)$ for $u \in H(v)$; but since $u \in \hat{V}$, the calculation of $\psi(u)$ may require the determination of $\psi(w)$ for some other $w \in V - \hat{V}$. The way out is by the rank decomposition discussed in the next section. There will remain the problem of finding disjoint paths, which we consider in Section 2.5. This allows us to apply Theorem 2.2 without circularity.

2.4. *Rank Decomposition of a Reduced GVG*

This section describes a decomposition of the nodes of a reduced GVG = (V, E, L) into sets for which we may completely characterize the minimal $\psi \in \Gamma_{\text{GVG}}$. This leads to an algorithm for the construction of ψ.

Fong, Kam, and Ullman [FKU] describe the rank decomposition of a dag; this provides a topological ordering of a dag from leaves to roots over which the dag may be efficiently reduced. Here we generalize the rank decomposition to a possibly cyclic GVG; this provides us a method of partitioning V into sets of text expressions over which ψ may have the same value; it also allows us to apply Theorem 2.2 without circularity, characterizing completely the minimal $\psi \in \Gamma_{\text{GVG}}$. In Section 2.5 we apply the rank decomposition to implement our direct method for symbolic evaluation.

The *rank* of a node $v \in V$ is defined (see Fig. 8)

$$\text{rank}(v) = 0 \quad \text{if } v \text{ is labeled with a constant symbol or an input variable at the start block } s.$$

$$= 1 + \text{MAX}\{\text{rank}(u) \mid (v, u) \in E\} \quad \text{for } v \text{ labeled with a function symbol.}$$

$$= \text{MIN}\{\text{rank}(u) \mid u \in H(v)\} \quad \text{for } v \text{ labeled with an input variable.}$$

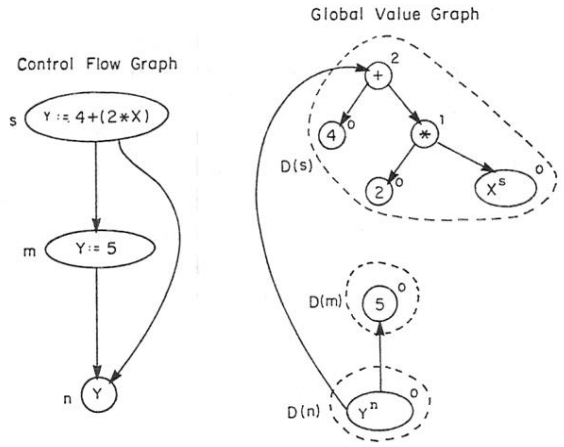

FIG. 8. Rank decomposition of a global value graph. The integer on the upper right-hand side of each node is its rank.

Observe that in the very simple case where the program contains only a single block of code at the start block s, then GVG consists of the dag $D(s)$. Hence the rank of a node $v \in V$ is one less than the length of a maximal path from v to a leaf of the dag $D(s)$.

LEMMA 2.3. $\psi(v) = \psi(v')$ *implies* $\text{rank}(v) = \text{rank}(v')$.

Proof. The result follows easily from the assumption ψ is minimal cover of Γ_{GVG}. We will proceed by induction of rank of v using the definition of the GVG at each stage of the induction.

Basis step. Suppose v is of rank 0, so $\psi(v) = \psi(v')$ is a constant symbol or input variable at the start block s. But since GVG is reduced, $L(v') = L(v)$ and v' is also of rank 0.

Inductive step. Suppose for some $r > 0$, $\text{rank}(w) = \text{rank}(w')$ for all w, $w' \in V$ such that $\text{rank}(w) < r$ and $\psi(w) = \psi(w')$. Consider some v, $v' \in V$ such that $\text{rank}(v) = r$.

Case a. Suppose $\psi(v) = \psi(v')$ is the function application $(\theta \mathscr{E}_1 \cdots \mathscr{E}_k)$. Then by Theorem 2.2, $\psi(v) = \psi(u)$ for all $u \in H(v)$, and similarly, $\psi(v') = \psi(u')$ for all $u' \in H(v')$. Fix some $u \in H(v)$ and $u' \in H(u')$. By definition of Γ_{GVG}, $L(u) = L(u') = \theta$ and if $w_1, \ldots, w_k$ are the immediate successors of u and $w_1', \ldots, w_k'$ are the immediate successors of u', then $\mathscr{E}_i = \psi(w_i) = \psi(w_i')$ for $i = 1, \ldots, k$. By the induction hypothesis, $\text{rank}(w_i) = \text{rank}(w_i')$ for $i = 1, \ldots, k$. Hence,

$$\begin{aligned} \text{rank}(v) &= \text{rank}(u) \\ &= 1 + \text{MAX}\{\text{rank}(w_1), \ldots, \text{rank}(w_k)\} \\ &= 1 + \text{MAX}\{\text{rank}(w_1'), \ldots, \text{rank}(w_k')\} \\ &= \text{rank}(u') \\ &= \text{rank}(v'). \end{aligned}$$

Case b. Suppose $\psi(v) = \psi(v')$ is an input variable. By Theorem 2.2, $\psi(v) = \psi(v') = L(u)$ and $H(u) = H(v)$ for some $u \in V$ contained on all use-def paths from v and v'. Hence, $\text{rank}(v) = \text{rank}(v') = \text{rank}(u)$. ∎

To compute the rank of all nodes in GVG 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 B

Input: GVG $= (V, E, L)$

output: RANK

```
begin
    declare RANK := an array of integers of length |V|;
    for all v ∈ V do
        RANK(v) := −outdegree(v) od;
    r := 0;
    Q' := {v | L(v) is a constant symbol};
    until Q' = the empty set { } do
        Q := Q'; Q' := the empty set { };
        comment Q = V̂_r;
    L: until Q = the empty set { } do
        delete v from Q;
        for each immediate predecessor u of v do
            if L(u) is a function symbol then
                if RANK(u) = −1 then
                    comment u ∈ V̂_{r+1};
                    RANK(u) := r + 1;
                    add u to Q'
                else RANK(u) := RANK(u) + 1 fi;
            else if RANK(u) < 0 then
                comment u ∈ V_r − V̂_r;
                RANK(u) := r;
                add u to Q fi;
            fi;
        fi;
      od;
    r := r + 1;
  od;
end.
```

THEOREM 2.3. *Algorithm* B *is correct and has time cost linear in* $|V| + |E|$.

Proof (by induction on r). *Basis step.* Initially, RANK(v) is set to $-$(outdegree of v) for each $v \in V$. So if $L(v)$ is labeled with a constant symbol then RANK(v) is set to 0. Also, Q is initially set to $\hat{V}_0$ just before label L.

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

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

(2) For each $v \in V$, RANK(v) = rank(v) if rank(v) $< r$ or $v \in \hat{V}_r$, and RANK(v) = $-$(number of successors of v with rank $> r$) if rank(v) $> r$ or $v \in V_r - \hat{V}_r$.

In the inner loop we add to Q exactly the nodes $V_r - \hat{V}_r = \{v \in V - \hat{V} \mid$ some element of $\hat{V}_r$ is reachable by a use-def path from $v\}$. For each such $v \in V_r - \hat{V}_r$ added to Q, RANK(v) is set to r. Also, for each $v \in \hat{V}$, if rank(v) $> r + 1$ then

RANK(v) is incremented by 1 for each immediate successor of v of rank r; if rank(v) $= r+1$ then all immediate successors of v are of rank $\leqslant r$ so RANK(v) is set to $r+1$ and v is added to Q. Thus, (1) and (2) are satisfied entering the loop on the $r+1$ time.

Now we show that Algorithm B 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 $O(1 + \text{indegree}(v))$. Since each node is added to Q' exactly once, the total time cost is linear in $|V| + |E|$. ∎

This suffices for the construction of ψ; $\psi(v)$ 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 the definition of Γ_{GVG} and Theorem 2.2.

Using this method could be inefficient, since theorem 2.2 could be expensive to apply and the representation 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 Section 2.5. The second problem is solved by constructing a special labeled dag; the construction of this dag and the final algorithm are given in Section 2.6.

2.5. *P-Graph Completion and Decomposition*

Let GVG $= (V, E, L)$ be a reduced global value graph. This section presents an efficient method for applying Theorem 2.2 to nodes in $V_r - \hat{V}_r$ (i.e., nodes of rank r labeled with input variables). Now to compute ψ^*, the minimal element of Γ_{GVG}, it suffices to find the partitioning of V such that $\psi^*(v) = \psi^*(u)$ iff v, u are in the same component of the partition. To represent such a partitioning, we distinguish one node of each component of the partitioning to be the *value source* of all other nodes of that block. We require that if $v \in V - \hat{V}$ (i.e., v is labeled with an input variable) then $\psi^*(v) = L(v)$ iff v is a value source. Let V^* be the set of value sources and let VS be a mapping from nodes in V to their value sources. Hence the fixed points of VS are the value sources and $VS^{-1}[V^*]$ is a partitioning of V. Note that, in general, the definition of "value source" is not uniquely determined, so the definition of V^* and VS depends on our particular choice of value sources. We shall find value sources by reducing this problem 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. (S will be easy to construct in all applications.)

P-GRAPH COMPLETION PROBLEM. Find

$$S^+ = S \cup \{v \in V_G \mid \text{there are at least two paths from distinct elements of } S \text{ to } v \text{ not containing any other element of } S\}.$$

This form of the problem is due to Karr [Ka], 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 $v \in V_G - S$ there is exactly one element $w \in S^+$ from which v is reachable by a path avoiding all other vertices of S^+ (and his proof extends directly to our slightly more general problem).

P-GRAPH DECOMPOSITION PROBLEM. Given G and S^+, find, for each $v \in V_G - S$, the unique $w \in S^+$ from which v is reachable by a path avoiding all other vertices of S^+.

We first show the *P*-Graph completion and decomposition problems can be solved efficiently. Shapiro and Saint give an $O(|V_G||E_G|)$ 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 2.2 (This construction was discovered independently by Tarjan [T2].)

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 2.4. *The members of S^+ are the sons of h in T.*

Proof. 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 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 (i.e., the only node on the paths that dominates v is h) 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 disjoint paths in G' from distinct members of S to v not containing any other element of S, and hence $v \in S^+$.

THEOREM 2.4. *For each $v \in V_G - S$, the unique node $w \in S^+$ from which v is reachable in G by a path avoiding vertices of $S^+ - \{w\}$, is the unique node which is a son of h and ancestor of v in T.*

Proof. Let w be that ancestor of v which is a son of h in T. By Lemma 2.4, $w \in S^+$, and clearly v is reachable from w in G by a path avoiding $S^+ - \{w\}$, since v

is reachable from w in T. Conversely, if v is reachable from $w \in S^+$ in G by a path avoiding $S^+ - \{w\}$ then w is a son of h in T by Lemma 2.4, and w must be an ancestor of v since otherwise v would be reachable from some other member of S^+ by a path avoiding w. ∎

Now we establish the relation of these problems to the problem of finding V^* and VS as stated above. Fix some V^* and VS by choosing one node of GVG for each value of ψ on V consistent with our definition of value sources. For each rank r, let $G_r = (V_r, E_r)$, where V_r is the set of all nodes of rank r of a reduced GVG as defined in Section 2.4 and E_r is the edge set derived from E by

(1) deleting all edges except use-def edges between nodes of rank r,

(2) for those remaining use-def edges (v, u) entering $u \in \hat{V}_r$, substituting instead the edge $(v, VS(u))$,

(3) finally reversing all edges.

Note that any edge of GVG departing from a member of $\hat{V}_r$ enters a node of rank $\leqslant r-1$. Let S_r be the set of all value sources of $\hat{V}_r$ plus all nodes of rank r labeled with input variables which have a departing use-def edge entering a node of rank greater than r. Note that for each node v of G_r, there is a node in S_r from which v is reachable in G_r. Finally, let S_r^+ be defined from S_r as in the statement of the P-graph completion problem.

LEMMA 2.5. *The members of S_r^+ are the value sources of rank r.*

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

Case 1. By definition, all elements of $\{VS(v) | v \in \hat{V}_r\}$ are value sources. Hence we need only consider the case where v is a node of rank r labled with an input variable which has a departing use-def edge (v, z) entering a node z of rank greater than r. Since v is of rank r, v must also have a departing use-def edge (v, u) leading to a node of rank r. By Lemma 2.3, $\psi(z) \neq \psi(u)$, so by the definition of Γ_{GVG}, $\psi(v) = L(v)$ and v is a value source.

Case 2. Suppose there are in G_r disjoint paths $(x_1, x_2, \ldots, x_j)$ and $(y_1, y_2, \ldots, y_k)$ in G_r from distinct $x_1, y_1 \in S_r$ to v. By construction of G_r, there exist distinct $\bar{x}_1$, $\bar{y}_1 \in H(v)$ such that $VS(\bar{x}_1) = x_1$, $VS(\bar{y}_1) = y_1$, and $(x_2, \bar{x}_1)$ and $(y_2, \bar{y}_1)$ are use-def edges, and so $p_1 = (v = x_j, x_{j-1}, \ldots, x_2, \bar{x}_1)$ and $p_2 = (v = y_k, y_{k-1}, \ldots, y_2, \bar{y}_1)$ are disjoint paths. Now suppose v is not a value source. Applying Theorem 2.2, there is a value source u (distinct from v) such that $\psi(v) = \psi(u) = L(u)$. Since p_1 and p_2 are disjoint they cannot both contain u. Suppose, without loss of generality, that p_1 avoids u. Then all maximal use-def paths from $\bar{x}_1$ contain u. Also, by definition of S_r, $\bar{x}_1 = x_1$ and there is a use-def edge (v, z) such that z is not of rank r. Since any maximal use-def path from z must contain u, $\text{rank}(z) = \text{rank}(u)$ implying that u is not of rank r. But, by hypothesis, all maximal use-def paths from v contain u, so $\text{rank}(v) = \text{rank}(u)$. This implies that v is not of rank r, contradicting our assumptions. ∎

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

THEOREM 2.5. *For all nodes $v \in V$ of rank r and labeled with an input variable, $vs(v)$ is the unique value source contained on all use-def paths in G_r from elements of S_r to v.*

Thus the problem of computing VS 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 B of Section 2.4, the latter in almost linear time by [LT].

2.6. *Our Algorithm for Symbolic Program Analysis*

In this section we pull together the various pieces developed in Sections 2.1–2.5 to give a unified presentation of our algorithm computing a minimal fixed point case. Instead of using the GVG directly to represent ψ^*, as suggested in the beginning of Section 2.5, we more economically represent ψ^* by a dag D^* derived from GVG by collapsing nodes into their value sources; more precisely $D^* = (V^*, E^*, L^*)$, where

$$V^* = \{VS(v) | v \in V\} = \text{the set of value sources,}$$

$$E^* = \{(VS(v), VS(u)) | (v, u) \in E \quad \text{and} \quad L(v) \text{ is a function symbol}\}$$

$$L^* \quad \text{is the restriction of } L \text{ to } V^*.$$

Recall from Section 2.1 that rooted dags may be used to represent expressions in EXP.

LEMMA 2.6. *For each node $v \in V$, $(D^*, VS(v))$ represents $\psi(v)$.*

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

$$\psi^*(VS(v)) = \psi^*(v)$$

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

$$(D^*, v) \text{ represents } \psi^*(v).$$

We proceed by induction on a topological ordering of D^*, from leaves to roots.

Basis step. If v is a leaf of D^*, then (D^*, v) represents the constant symbol $L(v) = \psi(v)$.

Induction step. Suppose v is in the interior of D^* and (D^*, u) represents $\psi^*(u)$ for all children u of v. Thus v must be labeled in L with a functon symbol θ and have immediate successors $u_1, \ldots, u_2$ in GVG. Then $VS(u_1), \ldots, VS(u_k)$ are the children of v in D^* and for $i = 1, \ldots, k$ by the induction hypothesis $(D^*, VS(u_i))$ represents $\psi^*(VS(u_i)) = \psi(u_i)$. Thus (D^*, v) represents $(\theta\psi^*(u_1) \cdots \psi^*(u_k)) = \psi^*(v)$ by definition of Γ_{GVG}. ∎

Our algorithm is given below. As in Section 2.4, we compute ψ^* and VS in the order of the rank of nodes in V. The array COLOR is used to discover nodes with the same ψ^*.

ALGORITHM C

input: GVG $= (V, E, L)$
output: VS and $D^* = (V^*, E^*, L^*)$.

```
begin
initialize:
    declare VS, COLOR to be arrays of length |V|;
    procedure COLLAPSE(S, u):
        for all v ∈ S do
            VS(v) := u;
            if u ≠ v then
                for each edge (w, v) entering v do substitute (w, u) od;
                for each edge (v, w) departing from v do substitute (u, w) od;
                delete v from the vertex set;                    fi;
        od;
Compute new labeling L' of V by Algorithm A
    and reduce GVG as described in Section 2.2;
Compute rank of nodes in V by Algorithm B of Section 2.3;

for r := 0 to {MAX rank(v) | v ∈ V} do
    Let V_r, V̂_r be the nodes in V, V̂ of rank r;
    for all v ∈ V̂_r do
        if r = 0 then COLOR(v) := L'(v)
        else COLOR(v) := ⟨L(v), u_1,..., u_k⟩ where
        u_1,..., u_k are the current immediate successors of v; fi; od;
    radix sort nodes in V̂_r by their COLOR;
    for each maximal set S ⊆ V̂_r containing nodes with the same COLOR do
            choose some u ∈ S;
            comment u is made a value source;
            COLLAPSE(S, u);
        od;
    Let h be some node not in V_r:
    E_r := S_r := the empty set { };
    for all v ∈ V̂_r do add VS(v) to S_r; od;
    for all v ∈ V_r − V̂_r do
        for each node u which is currently an immediate successor of v do
            if u is of rank r then add (u, v) to E_r;
            else add u to S_r; fi; od;
    Let T_r be the dominator tree of G_r = (V_r ∪ {h}, E_r ∪ {(h, v) | v ∈ S_r}, h);
    for all sons u of h in T_r do
```

comment by Theorem 2.4 and Lemma 2.5, u is a value source;
COLLAPSE ($\{$the descendents of u in $T_r\}, u$);
delete all edges departing from u;
od;
od;

Let V^*, E^* be the node set and edge list derived from V, E by the above collapses;
for all $v \in V^*$ **do** $L^*(v) := L'(v)$;
end.

THEOREM 2.6. *Algorithm* C *is correct and can be implemented in almost linear time.*

Proof. The correctness of Algorithm C follows directly from Theorems 2.4, 2.5 and Lemmas 2.5, 2.6.

In addition, we must show that Algorithm C can be implemented in almost linear time. The storage cost of GVG is linear in $|V|+|E|$. The initialization of Algorithm C costs time linear in $|N|+|A|$. Algorithms A and B cost linear time by Theorems 2.1 and 2.3, respectively. The time cost of the r'th execution of the main loop, exclusive of the computation of T_r, is linear in $|V_r|+|E_r|$, plus the sum of the outdegree of all $v \in V_r - \hat{V}_r$. (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, so a radix sort can be used to partition $\hat{V}_r$ by COLOR.) The computation of the dominator tree T_r requires by [LT] 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 for computing a minimal fixed point case ψ^*.

3. FURTHER WORK

3.1. *Improving the Efficiency of Our Algorithm for Symbolic Program Analysis*

The primary goal of this paper was to construct the minimal fixed point ψ^* of the functional μ. Actually, μ was defined relative to a program derived from the original program by adding a dummy assignment of the form $X := X$ at every block where program variable $X \in \Sigma$ is not assigned. This does not change the semantics of the program but requires the addition of $O(|\Sigma||N|)$ text expressions whose covers we are not actually concerned with. In practice we need the covers given by ψ^* only over the domain of the text expressions of the original program.

The algorithms of Section 2 allow us to construct, for any global value graph GVG, the unique minimal element of Γ_{GVG} in space linear in the size of GVG and

time almost linear in the size of GVG. Section 2.1 defines the standard global value graph GVG_0 which has size $O(|\Sigma||A|+l)$ and with the property that ψ^* is the minimal element of Γ_{GVG_0}. We describe here how we may construct a global value graph GVG_1 of size $O(d|A|+l)$, where d is a parameter of the program which is often of order 1 for block-structured programs but may grow to $|\Sigma|$. The construction of GVG_1 can be done by a preprocessing stage of [RT] costing a number of bit vector steps almost linear in $|A|+l$. Thus this preprocessing stage offers no theoretical advantage but in practice may often lead to a global value graph of size linear in the program and flow graph. The construction of GVG^+ can be done by a preprocessing stage of [RT] costing a number of bit vector steps almost linear in $|A|+l$. Appendix III shows GVG_1 has the property that the minimal element of Γ_{GVG_1} is the minimal fixed point of the functional Ψ defined in Section 2.4. In contrast to the iterative method, which for a large class of programs has storage cost $\Omega(l|N|)$ and time cost $\Omega(l|N|^2)$, our direct method has storage cost linear in the size of GVG_1 and time cost almost linear in the size of GVG_1.

A path is *m-avoiding* if the path does not contain node m. Consider blocks m, n in the control flow graph such that m dominates n. A program variable $X \in \Sigma$ is *definition-free between m and n*, if (1) $m=n$ or (2) m properly dominates n and X is not assigned a value on any m-avoiding control path from an immediate successor of m to an immediate predecessor of n (otherwise X is *defined* between m and n). We define a function W from text expressions which are input variables to blocks of the control flow graph. For each input variable X^n, $W(X^n)=m$, where m is the first block on the dominator chain of the control flow graph from the start block s to n such that X is definition-free between m and n. An algorithm in [RT] computes W in a number of bit vector steps almost linear in $|N|+l$.

It will be convenient to assume that for each text expression which is an input variable X^n such that $W(X^n)=n$, X is assigned a value at each block m immediately preceding n. We must add $O(d|N|)$ dummy assignments to accomplish this; d is often constant for block structured programs but may grow to $|\Sigma|$. Let $\mathrm{GVG}_0=(V, E, L)$ be the standard global value graph defined in Section 2.1. Let E_1 be the set of pairs of vertices $(u, v) \in V^2$ such that

(1) v is labeled with an input vriable X^n

(2) t' represents an output expression $\mathscr{E}(X, m)$

(3) either (a) $W(X^n)=n$ and m is an immediate predecessor of n in F, or (b) $W(X^n)=m$ properly dominates n.

Note that E_1 contains $O(d|A|+l)$ edges. Let E_{UD} be the use-def edges of GVG_0. Let GVG_1 be the global value graph with vertices V, labeling L, and use-def edges $E \cup E_1 - E_{\mathrm{UD}}$. Let $d=|E_1|/|A|$ and observe that $d \leqslant |\Sigma|$. Then $|E_1|=O(d|A|)$ and so GVG_1 is of size $O(|E_1|+l)=O(d|A|+l)$.

Appendix III proves Γ_{GVG_1} has a minimal fixed point which contains in its domain the minimal fixed point cover Ψ^*. Thus our algorithm given in Section 2 can be used to construct Ψ^* in time almost linear in the size of GVG_1.

3.2. *Improved Covers for Restricted Domains*

We show in Appendix I that there is no finite algorithm for computing minimal covers in the arithmetic domains. However, the minimal fixed point covers computed by our algorithm in Section 2 can be improved by use of domain-specific identities.

In [R1] our methods for computing covers are extended to programs which operate on records in a language such as PASCAL or LISP 1.0. There we use the domain specific fact that selections (such as *car* or *cdr* is LISP) on structures yield subcomponents for which we can derive covering expressions.

Appendix I

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 an *immediate predecessor* of v and v is an *immediate successor* of u. The *outdegree* of a node v is the number of immediate successors of v and the *indegree* is the number of immediate predecessors of v.

A *path from* u *to* w in G is a sequence of nodes $p=(u=v_1, v_2,..., v_k=w)$, where $(v_i, v_{i+1}) \in E$ for all i, $1 \leqslant i < k$. The *length* of the path p is $k-1$. The path p may be built by composing subpaths

$$p=(v_1,..., v_i)\cdot(v_i,..., v_k).$$

The path p is a *cycle* if $u=w$. A *strongly connected component* of G is a maximal set of nodes such that each pair in the set is contained in a common cycle.

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*. A *flow graph* is a rooted digraph such that the root r has no predecessors and every node is reachable from r. A diagraph is *labeled* if it is augmented with a mapping whose domain is the vertex set. An *oriented digraph* is digraph augmented with an ordering of the edges departing from each node. We shall allow any given edge of an oriented graph to appear more than once in the edge list.

A digraph G is *acyclic* if G contains no cycles, *cyclic* or otherwise. Let G be acyclic. If u is reachable from v, u is a *descendant* of v and v is an *ancestor* of u (these relations are *proper* if $u \neq v$). Nodes with no proper ancestors are called *roots* and nodes with no proper descendants are *leaves*. Immediate successors are called *sons*. Any total ordering consistent with either the descendant or the ancestor relation is a *topological ordering* of G.

A flow graph T is a *tree* if every node v other than the root has a unique immediate predecessor, the *father* of v. A topological ordering of a tree is a *preor-*

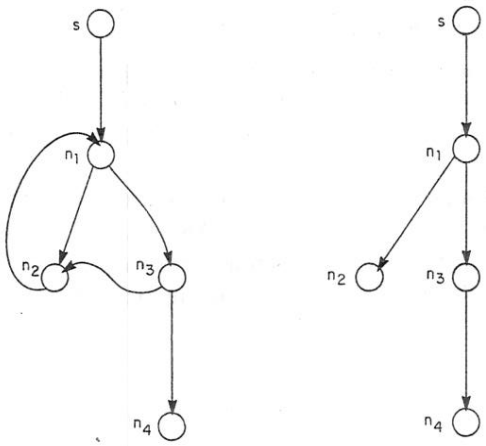

FIG. A.1. A flow graph and its dominator tree.

dering if it proceeds from the root to the leaves and is a *postordering* if it begins at the leaves and ends at the root. A *spanning tree* of a rooted digraph $G = (V, E, r)$ is a tree with node set V, an edge set contained in E, and a root r.

Let $G = (V, E, r)$ be a flow graph. A node u *dominates* a node v if every path from the root to v includes u (u *properly dominates* v if in addition, $u \neq v$). 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 is an ancestor of v in T_G. The father of a node in the dominator tree is the *immediate dominator* of that node. (See Fig. A1 for an example of a dominator tree.)

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 $O(n\alpha(n, n))$, where α is the extremely show growing function of [T3] (α is related to a functional inverse of Ackermann's function). Using adjacency lists, a digraph $G = (V, E)$ may be represented in space $O(|V| + |E|)$. Knuth [Kn1] gives a linear time algorithm for computing a topological ordering of an acyclic digraph. Lengauer and Tarjan [LT] present linear time algorithms for computing the strongly connected components of a diagraph and a spanning tree and an almost linear time algorithm for computing the dominator tree of a flow graph.

APPENDIX II

Undecidability of Various Code Improvements

The Introduction listed a number of code improvements which are related to the problem of determining minimal covers of text expressions. Here we show that even constant propagation, the most fundamental of these improvements, is recursively

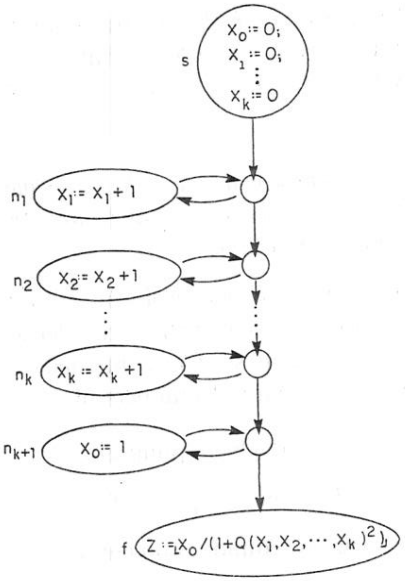

FIG. A.2. The control flow graph F_Q.

undecidable for programs evaluated within the arithmetic domain. This rules out the possibility of finding minimal covers even in simple domains. Previously, Kam and Ullman [KU2] have shown related global flow problems to be undecidable in an abstract, nonarithmetic domain.

THEOREM A1. *In the arithmetic domain, it is an undecidable problem to discover if a text expression is covered by a constant symbol.*

Proof. The method of proof will be to reduce this problem to that of the discovery of text expressions covered by constant symbols within the arithmetic domain (Z, I_Z).

Let $\{X_0, X_1, X_2, ..., X_k\}$ be a set of variables, where $k > 5$. Matijasevic [M] has shown that the problem of determining if a polynomial $Q(X_1, X_2, ..., X_k)$ has a root in the natural numbers (Hilbert's 10th problem) is recursively unsolvable.

Consider the flow graph F_Q of Fig. A.2. Let t be the text expression $\lfloor X_0^f/(1 + Q(X_1^f, ..., X_k^f)^2) \rfloor$ located at block f. We show t is covered by a constant symbol iff Q has *no* root in the natural numbers.

For any control path p from the start block s to the final block f and for $i = 0, 1, ..., k$, let $X_i(p) = I(\text{VALUE}(X_i f, p)) =$ the value of X_i just on entry to f relative to p. Also, let $X(p) = (X_1(p), ..., X_k(p))$. Observe that for any k-tuple of natural numbers z, there is a control path p from s to f such that $z = X(p)$.

(if) Suppose Q has no root in the natural numbers. Then for each control path p from s to f, $Q(X_1(p), ..., X_k(p)) \neq 0$, so $\text{VALUE}(t, p) = 0$. Thus, t is covered by the constant 0.

(only if) Suppose Q has a root z in the natural numbers. Then it is possible to find execution paths p and q from s to f such that $z = X(p)$ and $X(p) = 0$. Hence VALUE$(t, p) = 0$ and VALUE$(t, q) = 1$, so t is *not* covered by a constant symbol. ∎

COROLLARY A.1. *In the arithmetic domain, the following global flow problems are undecidable: discovery of minimal covers, birth and safe points of code motion, redundant text expressions, and loop invariants.*

Proof. It is easy to show that the problem of discovery of constant text expressions recursively reduces to each of these problems. Add the edge (f, n_1) to the control flow graph F of Fig. 4, so t is contained as a cycle of F. Then by Theorem 4, Q has *no* root in the natural numbers iff t is covered by 0

iff s is the birth point of t;

iff s is the safe point of t;

iff t is redundant on entry to f;

iff t is a constant loop invariant.

Thus, the problem of discovery of whether text expression t is covered by a constant reduces to each of the above global flow problems. (Note that the problem of safety of code motion is also hard for other reasons; if we add the text expression $t' = \lfloor 1/Q(x_1 f, \ldots, X_k f) \rfloor$ to block f then Q has no root in the natural numbers iff t' is safe at f.) ∎

APPENDIX III

Fixed Points of Γ_{GVG}

We define a partial mapping **min**: $\text{EXP}^2 \to \text{EXP}$ such that for all $\mathscr{E}, \mathscr{E}' \in \text{EXP}$,

$$\begin{aligned} \mathscr{E} \textbf{ min } \mathscr{E}' &= \mathscr{E} \quad \text{if origin}(\mathscr{E}) \text{ properly dominates origin}(\mathscr{E}') \\ &= \mathscr{E}' \quad \text{if origin}(\mathscr{E}') \text{ properly dominates origin}(\mathscr{E}) \end{aligned}$$

or if origin$(\mathscr{E})$ = origin$(\mathscr{E}')$ and

(i) if $\mathscr{E} = \mathscr{E}'$ then $\mathscr{E}$ **min** $\mathscr{E}' = \mathscr{E} = \mathscr{E}'$, or

(ii) if $\mathscr{E}$ is a constant symbol and $\mathscr{E}'$ is a function application, then $\mathscr{E}$ min $\mathscr{E}' = \mathscr{E}'$ **min** $\mathscr{E} = \mathscr{E}$, or

(iii) if $\mathscr{E}$, $\mathscr{E}'$ are function applications $(\theta\ \mathscr{E}_1 \cdots \mathscr{E}_k)$, $(\theta\ \mathscr{E}_1' \cdots \mathscr{E}_k')$, respectively, and $\overline{\mathscr{E}}_i = \mathscr{E}_i$ **min** $\mathscr{E}_i'$ is defined for $i = 1, \ldots, k$ then $\mathscr{E}$ **min** $\mathscr{E}' = (\theta \overline{\mathscr{E}}_1 \cdots \overline{\mathscr{E}}_k)$, and otherwise, $\mathscr{E}$ **min** $\mathscr{E}'$ is undefined.

We extend **min** to the partial mapping from pairs of elements of Γ_{GVG} to Γ_{GVG}

defined thus: for ψ, $\psi' \in \Gamma_{GVG}$, if for all $v \in V$, $\psi(v)$ **min** $\psi'(v) = \bar{\psi}(v)$ is defined then ψ **min** $\psi' = \bar{\psi}$ and otherwise ψ **min** ψ' is undefined.

Let GVG be as an arbitrary global value graph. We show that Γ_{GVG} is a semilattice. We require two technical lemmas:

LEMMA A1. *For any $v \in V$ labeled with an input variable and any control path p from the start block s to* loc(v), *there is a maximal use-def path q from v such that all the nodes in q have distinct* loc *values in p.*

Proof. We consider (t) to be a trivial use-def path. Suppose we have constructed a use-def path $(v = u_1, ..., u_i)$ such that $\text{loc}(u_i)$, $\text{loc}(u_{i-1})$,..., $\text{loc}(u_1)$ are distinct blocks occurring in this order in p. If u_i is not labeled with an input variable (and thus has no departing value edges) then $(t = u_1, ..., u_i)$ is a maximal use-def path. Otherwise, let p_i be the subpath of p from s to the first occurrence of block $\text{loc}(u_i)$ and let (u_i, u_{i+1}) be a use-def edge such that $\text{loc}(u_{i+1})$ occurs strictly before $\text{loc}(u_i)$ in p. Then $(t = u_1, ..., u_i, u_{i+1})$ is a use-def path and $\text{loc}(u_{i+1})$ is distinct from blocks $\text{loc}(u_1)$,..., $\text{loc}(u_i)$. The result thus follows from induction on the length of p. ∎

LEMMA A2. *For any $\psi \in \Gamma_{GVG}$ and $v \in V$,* origin($\psi(v)$) *dominates* loc(v).

Proof (by contradiction). Suppose for some $v \in V$, origin($\psi(v)$) does not dominate loc(v). Hence, there must be an input variable X^n occurring in $\psi(v)$ such that n does not dominate loc(v), and so there is an n-avoiding path p from the start block s to loc(v). Also, there must exist some $u \in V$ labeled with an input variable and also located at block n, such that $\psi(u) = X^n$. By Lemma A.1, we can construct a maximal use-def path $(u = u_1, ..., u_k)$ such that $\text{loc}(u_1)$,..., $\text{loc}(u_k)$ are distinct blocks in p. Let j be the maximal integer $\leqslant k$ such that $\psi(u_1) = \cdots = \psi(u_j)$. If $L(u_j)$ is an input variable, then $\psi(u_1) = L(u_j) = X^n$, so $\text{loc}(u_j) = n$ is contained in p, contradicting the assumption that p contains n. Otherwise, if $L(u_j)$ is not an input variable then neither is $\psi(v) = \psi(u_j)$, a contradiction with the assumption that $\psi(u) = X^n$. ∎

THEOREM A2. *Γ_{GVG} is a semilattice.*

Proof. It is sufficient to show **min** is well defined over Γ_{GVG}. We proceed by induction. Suppose for ψ, $\psi' \in \Gamma_{GVG}$ and some $\mathscr{E}$ in the domain of Ψ, $\psi(u)$ **min** $\psi'(u)$ is defined for all $u \in V$ such that $\psi(u)$ is a proper subexpression of $\mathscr{E}$. Consider some text expression v such that $\psi(v) = \mathscr{E}$. By Lemma 2.2, both origin($\psi(v)$) and origin($\psi'(v)$) are contained on all control paths from the start block s to loc(v), so we may assume without loss of generality that origin($\psi(v)$) dominates origin($\psi'(v)$). Observe that $\psi(v)$ **min** $\psi'(v) = \psi(v)$ if origin($\psi(v)$) properly dominates origin($\psi'(v)$) so we further assume that origin($\psi(v)$) = origin($\psi'(v)$).

Case 1. If $L(v)$ is a constant symbol c then $\psi(v) = \psi'(v) = c$ so $\psi(v) =$ **min** $\psi'(v) = c$.

Case 2. Suppose $L(v)$ is a function symbol θ and v has immediate successors $u_1,..., u_k$. By the induction hypothesis $\mathscr{E}_i' = \psi(u_i)$ **min** $\psi'(u_i)$ is defined for $i = 1,..., k$. Hence $\psi(v)$ **min** $\psi'(v)$ is the reduced expression derived from $(\theta\, \mathscr{E}_1' \cdots \mathscr{E}_k')$.

Case 3. Otherwise, suppose $L(v)$ is an input variable. Let p be a control path from the start block s to $\mathrm{loc}(v)$. By Lemma 2.1, we can construct a maximal use-def path $(v = u_1,..., u_k)$ such that for $i = 1,..., k$ each $\mathrm{loc}(u_i)$ is contained in p. Let j be the maximal integer such hat $\psi(u_1) = \cdots = \psi(u_j)$.

Case 3a. If $\psi'(v) = \psi(u_1) = \cdots = \psi(u_i) \neq \psi'(u_{i+1})$ for some i, $1 \leqslant i < j$, then by the definition of Γ_{GVG}, $\psi(v) = \psi'(u_i) = L(u_i)$. Hence $\mathrm{origin}(\psi'(v)) = n_i \neq n_j = \mathrm{origin}(\psi(v))$, contradicting our assumption that $\mathrm{origin}(\psi'(v)) = \mathrm{origin}(\psi(v))$.

Case 3b. Otherwise, suppose $\psi'(v) = \psi'(u_1) = \cdots = \psi'(u_j)$ so we have $\psi(v) = \psi(u_j)$ and $\psi'(v) = \psi'(u_j)$. Applying Cases 1 and 2, $\psi(v)$ **min** $\psi'(v) = \psi(u_j)$ **min** $\psi'(u_j)$ is defined if $L(u_j)$ is either a constant symbol or function symbol, so we assume $L(u_j)$ is an input variable. Since j is maximal, $\psi(v) = \psi(u_j) = L(u_j)$. If $\psi'(v) = \psi'(u_j) = L(u_j)$ then $\psi(v)$ **min** $\psi'(v) = L(u_j)$. Otherwise, suppose $\psi'(u_j) \neq L(u_j)$. For each use-def edge (u_j, v'), by the definition of Γ_{GVG}, $\psi'(u_j) = \psi'(v')$ and by Lemma 2.2, $\mathrm{origin}(\psi'(v'))$ dominates $\mathrm{loc}(v')$. Hence $\mathrm{origin}(\psi'(v)) = \mathrm{origin}(\psi'(u_j))$ is distinct from $\mathrm{origin}(\psi(v))$, contradicting our assumption that $\mathrm{origin}(\psi'(v)) = \mathrm{origin}(\psi(v))$. ∎

Theorem A.2 immediately implies that

COROLLARY A.2. *Γ_{GVG} has an unique minimal element* **min**(Γ_{GVG}).

Let GVG_0 be the standard global value graph defined in Section 2.1. We have shown that Γ_{GVG_0} is a finite semilattice and hence has a minimal element. We now show that this minimal element is the unique minimal fixed point of μ as defined in Section 1.4.

THEOREM A.3. *ψ^*, the minimal fixed point of μ, is identical to the unique minimal element of Γ_{GVG_0}.*

Proof. Observe that any fixed point of μ is an element of Γ_{GVG_0}. By Corollary 2.1, Γ_{GVG_0} has a unique minimal element $\hat{\psi} = \mathbf{min}(\Gamma_{\mathrm{GVG}_0})$. Suppose $\hat{\psi}$ is not a fixed point of μ. Observe that since $\hat{\psi} \in \Gamma_{\mathrm{GVG}_0}$, for each input variable X^n, if $\psi(X^n) \neq X^n$ then $\Psi(\hat{\psi})(X^n) = \hat{\psi}(X^n)$. Hence there is an input variable X^n such that $\hat{\psi}(X^n) = X^n$ but $\Psi(\hat{\psi})(X^n) = \mathscr{E}$, where $\mathscr{E} = \mathscr{E}(\hat{\psi}(X, m))$ for all blocks m immediately preceding block n in the control flow graph F.

We are going to construct a mapping $\psi \in \Gamma_{\mathrm{GVG}_0}$ distinct from $\hat{\psi}$ such that $\psi \leqslant \hat{\psi}$. This will contradict our assumption that $\hat{\psi}$ is the minimal element of Γ_{GVG_0}. For each text expression t, let $\psi(t)$ be derived from $\hat{\psi}(t)$ by substituting $\mathscr{E}$ for each occurrence of X^n, and then reducing the resulting expression. We now show $\hat{\psi} \in \Gamma_{\mathrm{GVG}_0}$. Consider any input variable $Y^{n'}$.

Case a. Suppose $\hat{\psi}(Y^{n'}) = Y^{n'}$. If $Y^{n'} \neq X^n$ then $\psi(Y^{n'}) = Y^{n'}$. Otherwise, if $Y^{n'} =$

(X, n) then for each block m immediately preceding block $n' = n$, $\psi(Y^{n'}) = \hat{\psi}(\mathscr{E}(Y, m)) = \mathscr{E}$, and since X^n is not contained in $\mathscr{E}$, $\psi(Y^{n'}) = \psi(\mathscr{E}(Y, m)) = \mathscr{E}$.

Case b. If $\hat{\psi}(Y^{n'}) \neq Y^n$ then for each block m immediately preceding n' in F, $\hat{\psi}(Y^{n'}) = \hat{\psi}(\mathscr{E}(Y, m))$, so $\psi(Y^{n'}) = \psi(\mathscr{E}(Y, m))$. Thus $\psi \in \Gamma_{\mathrm{GVG}_0}$. For each block m immediately preceding n in F, $\mathscr{E} = \psi(X^n) = \psi(\mathscr{E}(X, m))$, so

$$\begin{aligned} \mathrm{origin}(\psi(X^n) &= \mathrm{origin}(\psi(\mathscr{E}(X, m))) \qquad \text{dominates } \mathrm{loc}(\mathscr{E}(X, m)), \text{ by Lemma A.2} \\ &= m, \end{aligned}$$

and hence $\mathrm{origin}(\psi(X^n))$ properly dominates $\mathrm{origin}(\hat{\psi}(X^n))$. This implies that $\hat{\psi}$ is not the minimal element of Γ_{GVG_0}, a contradiction. ∎

Let GVG_1 be the global value graph defined in Section 3.1. Let ψ^+ be the minimal fixed point of Γ_{GVG_1}. By Theorem A.3, ψ^* is the minimal fixed point of Γ_{GVG_0}. As in Section 3.1, we assume that for each text expression which is input variable X^n such that X is not assigned at block n, then X is assigned to at each block immediately preceding n. Thus ψ^+ and ψ^* have the same domain.

THEOREM A.4. $\psi^+ = \psi^*$.

Proof. Clearly $\psi^+ \in \Gamma_{\mathrm{GVG}_0}$. Suppose, however that $\psi^+ \neq \psi^*$. Then since ψ^* is the unique minimal fixed point of Γ_{GVG_0}, there is some v such that $\mathrm{origin}(\psi^*(v))$ properly dominates $\mathrm{origin}(\psi^+(v))$. Choose v so that $\psi^+(v)$ has minimal rank and $\mathrm{origin}(\psi^+(v))$ is also minimal with respect to domination ordering. Now v is certainly not a constant. If v is of the form $(u_1, \ldots, u_k)$ then $\psi^*(u_i) \neq \psi^+(u_i)$ for some i, such that $\mathrm{rank}(u_i) < \mathrm{rank}(v)$, a contradiction with the assumption that v has minimal rank. Otherwise, suppose v is an input variable X^n. Since $\mathrm{origin}(\psi^+(v))$ is also minimal, we can assume that $\psi^+(v) = v$. Then X cannot be definition-free from $\mathrm{origin}(\psi^*(v))$ to n, and there must be use-def edges (v, u_1), (v, u_2) such that $\psi^+(u_1) \neq \psi^+(u_2)$. But this implies also that $\psi^*(v) = v$, a contradiction. ∎

REFERENCES

[A] F. E. ALLEN, Control flow analysis, *SIGPLAN Notices* **5**, No. 7 (1970), 1–19.

[AU1] A. V. AHO, AND J. D. ULLMAN, "The Theory of Parsing, Translation and Compiling," II, Prentice–Hall, Englewood Cliffs, N.I., 1973.

[AU2] A. V. AHO, AND J. D. ULLMAN, "Principles of Compiler Design," Addison–Wesley, Reading, Mass., 1977.

[CHT] T. E. CHEATHAM, G. H. HOLLOWAY, AND J. A. TOWNLEY, Symbolic evalution and the analysis of programs, *IEEE Trans. Software Eng.* **SE-5**, No. 4 (1979), 402–417.

[C] J. COCKE, Global common subexpression elimination, *SIGPLAN Notices* **5**, No. 7 (1970), 20–24.

[CA] J. COCKE, AND F. E. ALLEN, "A Catalogue of Optimization Transformations, Design and Optimization of Computers" (R. Rustin, Ed.), pp. 1–30, Prentice–Hall, Englewood Cliffs, N.J., 1971.

[E] C. EARNEST Some topics in code optimization, *J. Assoc. Comput. Mach.* **21**, No. 1 (1974), 76–102.

[FKU] E. A. FONG, J. B. KAM, AND J. D. ULLMAN, Application of lattice algebra to loop optimization, *in* "Conf. Record of the 2nd ACM Sympos. on Principles of Programm. Lang.", January 1975, pp. 1–9.

[FU] E. A. FONG AND J. D. ULLMAN, Induction variables in very high level languages, *in* "Conf. Record of the 2nd ACM Sympos. on Principles of Programm. Lang.," January 1976, pp. 1–9.

[G] C. M. GESCHKE, "Global Program Optimizations," Ph. D. thesis, Carnegie–Mellon University, Dept. of Computer Science, October 1972.

[H] M. S. HECHT, "Flow Analysis of Computer Programs," North-Holland, Amsterdam, 1977.

[HK] S. L. HANTLER, AND J. C. KING, An introduction to proving the correctness of programs, *Comput. Surveys* **8** (1976), 331–353.

[HU1] M. S. HECHT, AND J. D. ULLMAN, Flow graph reducibility, *SIAM J. Comput.* **1**, No. 2 (1972), 188–202.

[HU2] M. S. HECHT, AND J. D. ULLMAN, Analysis of a simple algorithm for global flow problems, *SIAM J. Comput.* **4**, No. 4 (1975), 119–532.

[KK] R. N. KALMAN, AND A. A. KORTESOJA, An optimizing Pascal compiler, *IEEE Trans. Software Eng.* **SE**-6 (1980), 512–519.

[KU1] J. B. KAM, AND J. D. ULLMAN, Global data flow problems and iterative algorithms, *J. Assoc. Comput. Mach.* **23**, No. 1 (1976), 158–171.

[KU2] J. B. KAM, AND J. D. ULLMAN, "Monotone Data Flow Analysis Frameworks," Tech. Rep. 167, Princeton University, Comp. Science Dept., 1976.

[Ka] M. KARR, "P-graphs," CAID–7101–1111, Massachusetts Computer Associates, 1975.

[KM] S. KATZ AND Z. MANNA, Logical analysis of programs, *Commun ACM.* **19** 1976, 188–206.

[Ki] G. A. KILDALL, A unified approach to global program optimization, *in* "Proceedings, ACM Sympos. on Prin. of Prog. Lang." Boston, Mass., 1973, pp. 194–206.

[Kin1] J. C. KING, "A Program Verifier," Ph. D. thesis, Carnegie–Mellon University, Dept. Computer Science, Pittsburgh, Pa., 1969.

[Kin2] J. C. KING, Symbolic execution and program testing, *Common. ACM.* **19** 1976, 385–394.

[Kn1] D. E. KNUTH, "The Art of Computer Programming," Vol. 1, "Fundamental Algorithms," Addison–Wesley, Reading, Mass., 1968.

[Kn2] D. E. KNUTH, Big omicron and big omega and bit theta, *SIGACT News*, April–June (1976), 18–24.

[LT] R. LENGAUER, AND R. E. TARJAN, A fast algorithm for finding dominators in a flow graph, *ACM Trans. Programm. Languages and Systems* **1** 1979, 121–141.

[L] D. LOVEMAN, Program improvement by source-to-source transformations, *J. Assoc. Comput. Mach.* **24** 1977, 121–145.

[M] Y. MATIJASEVIC, Enumerable sets are diophantine, *Dokl. Akad. Nauk SSSR* **191** (1970), 279–282. [Russian]

[MJ] S. S. MUCHNICK AND N. D. JONES, "Program Flow Analysis: Theory and Applications," Prentice–Hall, Englewood Cliffs, N.J., 1981.

[NO] C. G. NELSON, AND D. C. OPEN, A simplifier based on efficient decision algorithms, *in* "Conf. Rec. 5th Annu. ACM Sympos. Prin. Prog. Lang., " Tucson, Ariz., 23–25, 1978 pp. 141–150.

[R1] J. H. REIF, "Combinatorial Aspects of Symbolic Program Analysis," Ph. D. thesis, Harvard University, Div. of Engineering and Applied Physics, 1977.

[R2] J. H. REIF, Code motion, *SIAM J. Comput.* **9** 1980, 375–395.

[RT] J. H. REIF, AND R. E. TARJAN, Symbolic program analysis in almost-linear time, *SIAM J. Comput.* **11**, No. 1 1981, 81–93.

[SG] J. T. SCHWARTZ, Optimization of very high level languages–value transmission and its corollaries, *Comput. Languages* **1**, No. 2 1975, 161–194.

[SS] R. SHAPIRO, AND H. SAINT, "The Representation of Algorithms," RADC Tech. Rep. 313, June 1972.

[SHKN] T. STANDISH, D. HARRIMAN, D. KIBLER, AND J. NEIGHBORS, "The Irvine program transformation catalogue," Dept. Information and Computer Science, University of California, Irvine, January 1976.

[T1] R. E. TARJAN, Depth-first search and linear graph algorithms, *SIAM J. Comput.* **1**, No. 2 1972, 146–160.

[T2] R. TARJAN, Personal communication to M. Karr, 1976.

[U] J. D. ULLMAN, Fast algorithms for elimination of common subexpressions, *Acta Inf.* **2**, No. 3 1974, 191–213.

[W] B. WEGBREIT, The synthesis of loop predicates, *Commun. ACM.* **17**, No. 2 1974, 102–112.

[WZ] M. N. WEGMAN, AND F. K. ZADECK, Constant propagation with conditional branches, *in* "12th Annu. ACM Sympos. Princ. of Prog. Lang.," 1985.

Printed by the St. Catherine Press Ltd., Tempelhof 41, Bruges, Belgium