SYMBOLIC PROGRAM ANALYSIS IN ALMOST-LINEAR TIME

JOHN H. REIF† AND ROBERT E. TARJAN‡

Abstract. This paper describes an algorithm to construct, 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. We call such a mapping from text expressions to symbolic expressions a *cover.* Covers are useful in such program optimization techniques as constant propagation and code motion. The particular cover constructed by our methods is in general weaker than the covers obtainable by the methods of [Ki], [FKU], [RL], [R2] but our method has the advantage of being very efficient. It requires $O(m\alpha(m, n)+l)$ operations if extended bit vector operations have unit cost, where n is the number of vertices in the control flow graph of the program, m is the number of edges, l is the length of the program text, and α is related to a functional inverse of Ackermann's function [T2]. Our method does not require that the program be well-structured nor that the flow graph be reducible.

Key words. code movement, code optimization, constant propagation, data flow analysis, symbolic evaluation.

1. Introduction.

Let $\mathscr{E}$ be an expression which appears somewhere in a computer program. If $\mathscr{E}$ evaluates to a constant independent of the particular execution of the program, then the program can be improved by substituting the appropriate constant for $\mathscr{E}$ in the program text. The systematic application of this technique is called constant propagation. Another kind of improvement is possible if $\mathscr{E}$ occurs within a loop but has the same value for every execution of the loop; in this case the program may be improved by moving the computation of $\mathscr{E}$ outside the loop. (Note that this is not an improvement if the loop is executed less than twice.) Constant propagation and code motion require for their application a mapping from text expressions to symbolic expressions such that in any program execution every symbolic expression has the same value as its corresponding text expression. We call such a mapping a *cover.* We desire a cover which is as simple as possible in some appropriately defined sense, but even determining whether a given text expression always evaluates to a constant is an undecidable problem. In this paper we describe an algorithm for efficiently computing a reasonably good cover.

In order to address this problem, we need some definitions. We represent the flow of control through a program π by a flow graph[1] $G=(V, E, r)$ where each vertex v represents a consecutive block of assignment statements and each edge $(u, v)\in E$ specifies a possible flow of control caused by a branch from a test statement. An execution of π induces a path in G beginning at the *start vertex r.* We shall denote the number of vertices in G by n and the number of edges in G by m.

Let $\Sigma=\{X, Y, Z, \cdots\}$ be the set of program variables occurring within π. A program variable $X\in\Sigma$ is *defined* at $v\in V$ if X occurs on the left-hand side of an assignment statment of v. For each program variable $X\in\Sigma$ and vertex $v\in V$, we let the *entry variable* X^v denote the value of X on entry to v.

---

† Aiken Computer Laboratory, Harvard University, Cambridge, Massachusetts 02138. The work of this author was supported by Naval Electronics System Command contract N00039-76-C-0168 and Rome Air Development Center contract F30602-76-C-0032.

‡ Department of Computer Science, Stanford University, Stanford, California 94305. Present address: Bell Laboratories, 600 Mountain Ave., Murray Hill, New Jersey 07974. The work of this author was supported by National Science Foundation grant MCS-75-22870 A02, by Office of Naval Research contract N00014-76-C-0330, by a Guggenheim fellowship, and by Bell Laboratories.

[1] The appendix contains the graph-theoretic terminology we employ.

Let θ be the set of function signs occurring in the program. For simplicity, we assume a domain D such that every k-ary function represented by a sign in θ has the same domain D^k. Let C be a set of constant signs containing a unique sign for every element in D. Let EXP be the set of expressions built from entry variables, constant signs in C, and function signs in θ. To each expression $\mathscr{E} \in \mathrm{EXP}$ corresponds a unique *reduced expression* $\mathscr{E}_R$ formed by repeatedly substituting the appropriate constant sign for each subexpression of $\mathscr{E}$ consisting of a function sign applied to constant signs.

For any expression $\mathscr{E} \in \mathrm{EXP}$ and any execution of the program π, the *value* of $\mathscr{E}$ on exit from a vertex v is defined as follows: If $\mathscr{E}$ contains an entry variable X^u such that control has never entered u, then the value of $\mathscr{E}$ is undefined. Otherwise the value of $\mathscr{E}$ is computed by substituting for each entry variable X^u the value of X when control last entered u, and evaluating the resulting expression.

For each vertex $v \in V$ and program variable $X \in \Sigma$ defined at v, the *exit expression* $\mathscr{E}(X, v) \in \mathrm{EXP}$ is formed as follows. Begin by letting the expression $\mathscr{E}$ be X. Process each assignment statement of v, starting from the last assignment defining X and working backwards to the first assignment in v. To process an assignment $Y := \mathscr{E}'$, replace each occurrence of Y in $\mathscr{E}$ by $\mathscr{E}'$. After all assignments are processed, reduce $\mathscr{E}$ and replace each occurrence of a variable Y by the corresponding entry variable Y^v. The resulting exit expression $\mathscr{E}(X, v)$ represents the value of X on exit from v in terms of constants and values of variables on entry to v. For example, $\mathscr{E}(Z, v_2) = Z^{v_2} + (X^{v_2} * Y^{v_2})$ represents the value of Z on exit from vertex v_2 in the flow graph of Fig. 1.

A *text expression* is any subexpression of an exit expression $\mathscr{E}(X, v)$ (including the expression itself); we say the text expression *occurs at* v. An expression $\mathscr{E} \in \mathrm{EXP}$ *covers* a text expression t occurring at v if for any execution of program π, $\mathscr{E}$ and t have the

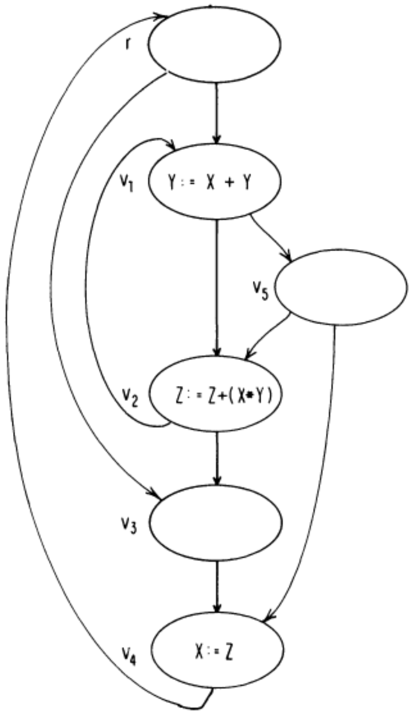

FIG. 1. *A program flow graph.*

same value on any exit from v. See Fig. 1. This definition implies that if X^u appears in $\mathscr{E}$ then u dominates v. Thus there is a unique vertex v which is minimal (i.e., closest to the start vertex) with respect to the dominator relation and such that for all entry variables X^u in $\mathscr{E}$, u dominates v. We call such a vertex the *origin* of $\mathscr{E}$; it is the earliest point in the program at which $\mathscr{E}$ can be computed.

A cover of π is a mapping Ψ from all text expressions to reduced expressions in EXP, such that, for each text expression t, $\Psi(t)$ covers t. We would like to construct a cover whose origins are minimal with respect to the dominator relation. We can use such a cover for constant propagation: if a constant sign c covers a text expression t, we may substitute c in line in the program text for the computation associated with c.

We can also use a cover in code motion. If we define the *birthpoint* of a text expression t to be the minimal vertex to which the computation of t may be moved, then the birthpoint of t is precisely the origin of a minimal cover of t. For example, in Fig. 1 the birthpoint of text expression $t = X^{v_2} * Y^{v_2}$ is v_1; $X^r * (X^r + Y^{v_1})$ covers t. Code

Text expression	Covering expressions
X^{v_2}	X^r
Y^{v_2}	$X^r + Y^{v_1}$
Z^{v_2}	Z^{v_1}
$\mathscr{E}(Z, v_2) = Z^{v_2} + (X^{v_2} * Y^{v_2})$	$Z^{v_1} + (X^r * (X^r + Y^{v_1}))$

FIG. 2. *Symbolic analysis of the program in Fig.* 1.

motion requires approximations to birthpoints (i.e., vertices which are dominated by the true birthpoints) and other knowledge including knowledge of the cycle structure of the flow graph of π. (We may not wish to move code as far as the birthpoint since the birthpoint may be contained in control cycles avoiding the original location of the code.) [R1] presents efficient algorithms which utilize approximate birthpoints for code motion optimization. See [AU], [CA], [E], [G] for further discussion of code motion optimizations. Other practical uses of covers have been made by [FK] in their optimizing Pascal compiler.

Unfortunately, for programs which manipulate the natural numbers using ordinary arithmetic the problem of computing a minimal cover is recursively unsolvable [R2]. The usual approach in program optimization is to trade accuracy for speed; [FKU], [Ki], [RL], [R2] present fast algorithms which compute reasonably good covers whose origins yield approximate birthpoints. The fastest of these [RL], [R2] has a time bound almost linear in $m \cdot |\Sigma| + l$, where l is the length of the program text.

In this paper we describe a very fast algorithm for computing a rather weak cover. This *simple cover* can be used directly for code optimization, or it can serve as input to a more powerful method for symbolic evaluation presented in [RL], [R2]. From a data structure called a *global value graph* (which is related to the use-definition chains of [AU], [Sc] used to represent the flow of values through a program), the algorithm of [RL], [R2] constructs a cover which yields better approximate birthpoints than does the simple cover. This algorithm runs in time almost linear in the size of the input global value graph, which is very compact when constructed from the simple cover [RL], [R2].

In order to define the simple cover we need one more concept. A variable X is *definition-free* between distinct vertices u and v if no u-avoiding path from a successor of u to a predecessor of v contains a definition of X. By convention any program variable X is definition-free between v and v for any vertex v. For any entry variable X^v which is a text expression, the *simple origin* of X^v is the minimal vertex u (with respect to the dominator relation) such that X is definition-free between u and v. In the example of Fig. 1, X^{v_2} has a simple origin r, and Y^{v_2} and Z^{v_2} have simple origin v_1. If X^v has simple origin $u \neq v$, then on any execution of π the program variable X has the same value on entry to v as it did after the most recent execution of u; we take the simple origin as an approximation to the birthpoint of X^v.

We recursively define the simple cover Ψ using simple origins. If t contains no entry variables then $\Psi(t) = t$. Otherwise we form $\Psi(t)$ from t by applying the following transformation.

(i) Repeat the following step for all entry variables X^v occurring in t: Let u be the simple origin of X^v. If $u = v$ do nothing. Otherwise replace X^v in t by $\Psi(\mathscr{E}(X, u))$ if X is defined at u or by X^u if X is not defined at u.

(ii) Reduce the resulting expression.

Our algorithm for computing the simple cover consists of three parts, described in §§ 2–4 of this paper. First, we determine for each vertex v the set of program variables defined between the immediate dominator of v and v itself. We call this set of variables idef (v). The idef computation can be regarded as a path problem of the kind studied in [GW], [T3], but another approach is more fruitful: a straightforward modification of the dominator-finding algorithm of [LT] computes idef in $O(m\alpha(m, n) + l)$ time, assuming that logical bit vector operations on vectors of length $|\Sigma|$ have unit cost, where l is the length of the program text and α is related to an inverse of Ackermann's function [T2].

Second, we use idef to compute the simple origins of all entry variables appearing as text expressions. This computation requires a variable-length shift operation on bit vectors (shift left to the first nonzero bit) and requires $O(n + l)$ time. Third, we construct a directed acyclic graph representing the simple cover (we save space by combining common subexpressions). This algorithm also requires $O(n + l)$ time but uses no bit vector operations. The total running time of our algorithm is thus $O(m\alpha(m, n) + l)$ if extended bit vector operations require constant time.

2. An algorithm for computing idef based on finding dominators.

In this section we shall describe an algorithm for computing idef (v) for all vertices $v \in V$ in the flow graph $G = (V, E, r)$ of a computer program. We obtain the algorithm by adding appropriate extra steps to the dominators algorithm of [LT], and we shall assume that the reader is familiar with [LT]. Our algorithm requires def $(w) = \{X | X$ is defined at $w\}$ for each vertex $w \in V$ as input and uses set union as a basic operation. If each subset of Σ is represented as a bit vector of length $|\Sigma|$, then a set union is equivalent to an "or" operation on bit vectors; we shall assume each set union requires constant time. Construction of def (w) for all vertices w is easy and requires time proportional to the length of the program text.

Properties of idef. For any vertex $w \neq r$, let idom (w) be the immediate dominator of w in G. For $w \neq r$, we define idef$(w) = \bigcup$ {def $(v)|$ there is a nonempty path from v to w which avoids idom (w)}. Note that def (w) is a term in the union defining idef (w) if and only if there is a cycle containing w but avoiding idom(w). To compute idom and idef, we first perform a depth-first search on G, starting from vertex r and numbering the vertices from 1 to n as they are reached during the search. The search generates a spanning tree T rooted at r, with vertices numbered in preorder [T1]. For convenience in stating our results, we shall assume in this subsection that all vertices are identified by number, and we shall use $\rightarrow$, $\overset{*}{\rightarrow}$, $\overset{+}{\rightarrow}$ to denote ancestor-descendant relationships in T (see the appendix).

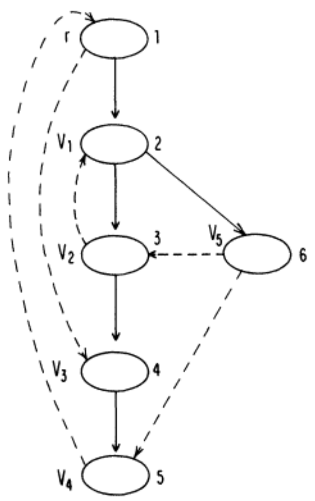

FIG. 3. *Depth-first search of the flow graph given in Fig. 1. Solid edges denote tree edges and dotted edges denote nontree edges. The depth-first search number is given to the right of each vertex.*

vertex	number	idom	sdom	def	idef	sdef
r	1	—	—	$\varnothing$	—	—
v_1	2	r	r	$\{Y\}$	$\{Y, Z\}$	$\{Y, Z\}$
v_2	3	v_1	v_1	$\{Z\}$	$\varnothing$	$\varnothing$
v_3	4	r	r	$\varnothing$	$\{Y, Z\}$	$\varnothing$
v_4	5	r	v_1	$\{X\}$	$\{Y, Z\}$	$\varnothing$
v_5	6	v_1	v_1	$\varnothing$	$\varnothing$	$\varnothing$

FIG. 4. *Tabulation of information calculated for the program flow graph given in Fig.* 1.

The following *paths lemma* is an important property of depth-first search and is crucial to the correctness of our algorithm.

LEMMA 2.1 [T1]. *If v and w are vertices of G such that $v \leqq w$, then any path from v to w must contain a common ancestor of v and w in T.*

As an intermediate step, the dominators algorithm computes a value for each vertex $w \neq r$ called its *semi-dominator*, denoted sdom (w) and defined by

$$\text{sdom}(w) = \min\{v \mid \text{there is a path } v = v_0, v_1, \cdots, v_k = w \text{ such that } v_i > w \text{ for } 1 \leqq i < k\}. \tag{1}$$

We shall in addition compute a value sdef (w) for each vertex $w \neq r$ defined by

$$\text{sdef}(w) = \bigcup \{\text{def}(v) \mid \text{there is a nonempty path } v = v_0, v_1, \cdots, v_k = w \text{ such that } v_i \geqq w \text{ for } 0 \leqq i \leqq k\}. \tag{2}$$

The following properties of semi-dominators and dominators justify the dominators algorithm.

LEMMA 2.2 [LT]. *Let $w \neq r$. Then* idom $(w) \overset{*}{\rightarrow}$ sdom $(w) \overset{+}{\rightarrow} w$.

THEOREM 2.1 [LT]. *For any vertex $w \neq r$,*

$$\text{sdom}(w) = \min(\{v \mid (v, w) \in E \textit{ and } v < w\} \cup \{\text{sdom}(u) \mid u > w \textit{ and there is an edge } (v, w) \textit{ such that } u \overset{*}{\rightarrow} v\}). \tag{3}$$

THEOREM 2.2 [LT]. *Let $w \neq r$ and let u be a vertex for which* sdom (u) *is minimum among vertices u satisfying* sdom $(w) \overset{+}{\to} u \overset{*}{\to} w$. *Then*

$$\text{idom}(w) = \begin{cases} \text{sdom}(w) & \textit{if } \text{sdom}(w) = \text{sdom}(u), \\ \text{idom}(u) & \textit{otherwise}. \end{cases} \tag{4}$$

The dominators algorithm uses Theorem 2.1 to efficiently compute semi-dominators and Theorem 2.2 to efficiently compute immediate dominators. We shall use two analogous results to efficiently compute sdef and idef.

THEOREM 2.3. *Let $w \neq r$ and let*

$$\text{adef}(w) = \{\text{def}(u) \cup \text{sdef}(u) \mid u > w \\ \textit{and there is an edge } (v, w) \textit{ such that } \overset{*}{\to} v\}.$$

Then

$$\text{sdef}(w) = \begin{cases} \text{def}(w) \cup \text{adef}(w) & \textit{if there is an edge } (v, w) \textit{ such that } w \overset{*}{\to} v, \\ \text{adef}(w) & \textit{otherwise}. \end{cases} \tag{5}$$

Proof. First we show that the right side of (5) contains sdef (w). Let $v = v_0, v_1, \cdots, v_k = w$ be a nonempty path such that $v_i \geqq w$ for $0 \leqq i \leqq k$. We can assume without loss of generality that the path $v_0, v_1, \cdots, v_{k-1}$ is simple and $v_i \neq w$ for $1 \leqq i \leqq k-1$. Let j be minimum such that $v_j \overset{*}{\to} v_{k-1}$. By Lemma 2.1, $v_i > v_j$ for $0 \leqq i \leqq j-1$. We consider three cases. If $j \neq 0$, then $v_j \neq w$, and def $(v) \subseteq$ sdef $(v_j) \subseteq$ adef (w). If $j = 0$ and $v \neq w$, then def $(v) \subseteq$ adef (w). If $j = 0$ and $v = w$, then the edge (v_{k-1}, w) must satisfy $w \overset{*}{\to} v_{k-1}$, and the right side of (5) explicitly contains def (v). Thus in any case the right side of (5) contains def (v). Since this is true for any appropriate v, the right side of (5) contains sdef (w).

Now we show that sdef (w) contains the right side of (5). Suppose there is an edge (v, w) such that $w \overset{*}{\to} v$. Then the path consisting of the tree path from w to v followed by the edge (v, w) contains no vertices smaller than w, and def $(w) \subseteq$ sdef (w). Let u be a vertex such that $u > w$ and there is an edge (v, w) such that $u \overset{*}{\to} v$. Let x be any vertex for which there is a nonempty path $x = v_0, v_1, \cdots, v_k = u$ such that $v_i \geqq u$ for $0 \leqq i \leqq k$. Then this path, followed by the tree path from u to v, followed by the edge (v, w), contains no vertices smaller than w. Thus def $(x) \subseteq$ sdef (w). Since this is true for any such x, sdef $(u) \subseteq$ sdef (w). Furthermore def $(u) \subseteq$ sdef (w). It follows that adef $(w) \subseteq$ sdef (w), and the theorem is true. □

THEOREM 2.4. *Let $w \neq r$. Let u be a vertex for which* sdom (u) *is minimum among vertices u satisfying* sdom $(w) \overset{+}{\to} u \overset{*}{\to} w$. *Let* tdef $(w) = \bigcup\{\text{def}(x) \cup \text{sdef}(x) \mid \text{sdom}(w) \overset{+}{\to} x \overset{+}{\to} w\}$. *Then*

$$\text{idef}(w) = \begin{cases} \text{tdef}(w) \cup \text{sdef}(w) & \textit{if } \text{sdom}(w) = \text{sdom}(u), \\ \text{idef}(u) \cup \text{tdef}(w) \cup \text{sdef}(w) & \textit{otherwise}. \end{cases} \tag{6}$$

Proof. First we show that the right side of (6) contains idef (w). Let $v = v_0, v_1, \cdots, v_k = w$ be a nonempty path which avoids idom (w). Let v_i be the minimum vertex on this path such that $i \leqq k-1$. If $v_i \geqq w$, then def $(v) \subseteq$ sdef (w) by the definition of sdef.

Suppose on the other hand that $v_i < w$. By Lemma 2.1, there is some j in the range $i \leqq j \leqq k$ such that v_j is an ancestor of both v_i and w. This means $v_j \leqq v_i$. But by the definition of v_i, $v_i \leqq v_j$. Thus, $v_i = v_j$ and $v_i \overset{+}{\to} w$. We must consider three cases.

(i) Suppose sdom $(w) \overset{+}{\to} v_i$ and $i = 0$. Then def $(v) =$ def $(v_i) \subseteq$ tdef.

(ii) Suppose sdom $(w) \overset{+}{\to} v_i$ and $i \neq 0$. Then def $(v) \subseteq$ sdef $(v_i) \subseteq$ tdef.

(iii) Suppose $v_i \overset{*}{\to} \mathrm{sdom}(w)$. The path from r to w consisting of the tree path from r to v_i followed by the path $v_i, v_{i+1}, \cdots, w$ must contain $\mathrm{idom}(w)$; thus $\mathrm{idom}(w) \overset{+}{\to} v_i$. By Theorem 2.2, $\mathrm{sdom}(w) \neq \mathrm{sdom}(u)$ (which means the second half of (6) applies) and $\mathrm{idom}(w) = \mathrm{idom}(u)$. The path from v to u consisting of $v = v_0, v_1, \cdots, v_i$ followed by the tree path from v_i to u avoids $\mathrm{idom}(u)$, which means $\mathrm{def}(v) \subseteq \mathrm{idef}(u)$.

In all cases $\mathrm{def}(v)$ is contained in the right side of (6); since this is true for any appropriate v, $\mathrm{idef}(w)$ is contained in the right side of (6) by the definition of idef.

It remains to show that $\mathrm{idef}(w)$ contains the right side of (6). Let x be any vertex such that $\mathrm{sdom}(w) \overset{+}{\to} x \overset{+}{\to} w$, and let $v = v_0, v_1, \cdots, v_k = x$ be any path such that $v_i \geqq x$ for $0 \leqq i \leqq k$. Since $\mathrm{idom}(w) \overset{*}{\to} \mathrm{sdom}(w)$, the path from v to w consisting of the path $v = v_0, v_1, \cdots, v_k = x$ followed by the tree path from x to w avoids $\mathrm{idom}(w)$. It follows that $\mathrm{tdef} \subseteq \mathrm{idef}(w)$. Since $\mathrm{idom}(w) < w$, it is immediate that $\mathrm{sdef}(w) \subseteq \mathrm{idef}(w)$. If $\mathrm{sdom}(w) \neq \mathrm{sdom}(u)$, then $\mathrm{idom}(w) = \mathrm{idom}(u) \overset{+}{\to} u$, and any $\mathrm{idom}(u)$-avoiding path to u can be extended to an $\mathrm{idom}(w)$-avoiding path to w by adding the tree path from u to w. Thus in this case $\mathrm{idef}(u) \subseteq \mathrm{idef}(w)$ $\square$

Details of the algorithm. The algorithm for computing immediate dominators and idef consists of the following four steps.

Step 1. Carry out a depth-first search of the problem graph. Number the vertices from 1 to n as they are reached during the search. Initialize the variables used in succeeding steps.

Step 2. Compute the semi-dominators of all vertices by applying Theorem 2.1 and the sdef values by applying Theorem 2.3. Carry out the computation vertex-by-vertex in decreasing order by number.

Step 3. Implicitly define the immediate dominator of each vertex by applying Theorem 2.2 and partially compute idef values by applying Theorem 2.4.

Step 4. Explicitly define the immediate dominator of each vertex and finish computing idef. Carry out the computation vertex-by-vertex in increasing order by number.

The dominators algorithm used the following arrays.

Input

succ (v):	The set of vertices w such that (v, w) is an edge of the graph.

Computed

parent (w):	The vertex which is the parent of vertex w in the spanning tree generated by the search.
pred (w):	The set of vertices v such that (v, w) is an edge of the graph.
semi (w):	A number defined as follows: (i) Before vertex w is numbered, semi $(v) = 0$. (ii) After w is numbered but before its semi-dominator is computed, semi (w) is the number of w. (iii) After the semi-dominator of w is computed, semi (w) is the number of the semi-dominator of w.
vertex (i):	The vertex whose number is i.
bucket (w):	A set of vertices whose semi-dominator is w.
dom (w):	A vertex defined as follows: (i) After Step 3, if the semi-dominator of w is its immediate dominator, then dom (w) is the immediate dominator of w. Otherwise dom (w) is a vertex v whose number is smaller than that of w and whose immediate dominator is also the immediate dominator of w. (ii) After Step 4, dom (w) is the immediate dominator of w.

In addition, our algorithm uses def (w) as input and computes sdef (w) and idef (w).

Rather than converting vertex names to numbers during Step 1 and converting numbers back to names at the end of the computation, the dominators algorithm refers to vertices as much as possible by name. Arrays semi and vertex include all necessary information about vertex numbers. Array semi serves a dual purpose, representing (though not simultaneously) both the number of a vertex and its semi-dominator.

During Step 1, our algorithm initializes parent, pred, semi, vertex, and sdef. When a vertex w receives a number i, the algorithm assigns semi $(w) = i$ and vertex $(i) = w$. Step 1 also initializes sdef $(w) = \emptyset$ and updates sdef $(w) =$ def (w) if it finds an edge (v, w) such that $w \overset{*}{\rightarrow} v$. Implementation of Step 1 is straightforward, and we omit the details.

The algorithm carries out Steps 2 and 3 simultaneously, processing the vertices $w \neq r$ in decreasing order by number. During this computation the algorithm maintains an auxiliary data structure that represents a forest contained in the depth-first spanning tree. More precisely, the forest consists of vertex set V and edge set $\{(\text{parent}\,(w), w) | \text{vertex } w \text{ has been processed}\}$. The algorithm uses one procedure to construct the forest and two procedures to extract information from it.

LINK (v, w):	Add edge (v, w) to the forest.
EVAL (v):	If v is the root of a tree in the forest, return v. Otherwise, let r be the root of the tree in the forest which contains v. Return any vertex $u \neq r$ of minimum semi (u) on the path $r \overset{*}{\rightarrow} v$.
EVALDEF (v):	If v is a tree root, return $\emptyset$. Otherwise, let $r = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k = v$ be the tree path from the root of the tree containing v to v. Return $\cup\{\text{def}\,(v_i) \cup \text{sdef}\,(v_i) | 1 \leqq i \leqq k\}$.

Reference [LT] explains how to use EVAL to compute semi-dominators and dominators; we shall describe how to use EVALDEF analogously to compute sdef and idef. When a vertex w is processed, the algorithm examines each edge $(v, w) \in E$ and updates sdef by assigning sdef $(w) :=$ sdef $(w) \cup$ EVALDEF (v). After w is processed, sdef (w) has the proper value by Theorem 2.3. To verify this claim, consider any edge $(v, w) \in E$. If v is numbered no greater than w, then v is unprocessed when (v, w) is examined, which means v is the root of a tree in the forest and EVALDEF (v) returns $\emptyset$. If v is numbered greater than w, then EVALDEF returns $\cup\{\text{def}\,(u) \cup \text{sdef}\,(u) | u > w \text{ and } u \rightarrow^* v\}$. Thus the algorithm computes sdef exactly as specified in Theorem 2.3.

After processing w to compute semi (w) and sdef (w), the algorithm adds w to bucket (vertex (semi (w))) and adds a new edge to the forest using LINK (parent (w), w). This completes Step 2 for w. The algorithm then empties bucket (parent (w)), carrying out Step 3 for each vertex v in the bucket. By applying EVAL (v), the algorithm obtains a vertex u satisfying the condition in Theorem 2.2 and 2.4. Using this u, the algorithm implicitly computes the immediate dominator of v. The algorithm also partially computes idef (v) by assigning idef (v):=sdef $(v) \cup$ EVALDEF (parent (v)). (EVALDEF (parent (v)) = tdef (v) as defined in Theorem 2.4.) In Step 4, the algorithm examines vertices in increasing order by number, filling in the immediate dominators not explicitly computed by Step 3 and completing the computation of idef. Here is an Algol-like version of Steps 2–4. The bracketed statements are those added to the original dominators algorithm to compute sdef and idef.

```
        **comment** initialize variables;
        **for** $i := n$ **by** $-1$ **until** 2 **do**
            $w :=$ vertex (i);
Step 2:     **for each** $v \in$ pred $(w)$ **do**
                $u :=$ EVAL $(v)$;
                **if** semi $(u) <$ semi $(w)$ **then** semi $(w) :=$ semi $(u)$ **fi**;
                [sdef $(w) :=$ sdef $(w) \cup$ EVALDEF $(v)$] **od**;
            add $w$ to bucket (vertex (semi $(w)$));
            LINK (parent $(w)$, $w$);
Step 3:     **for each** $v \in$ bucket (parent $(w)$) **do**
                delete $v$ from bucket (parent $(w)$);
                $u :=$ EVAL $(v)$;
                dom $(v) :=$ **if** semi $(u) <$ semi $(v)$ **then** $u$
                                **else** parent $(w)$ **fi**;
                [idef $(v) :=$ sdef $(v) \cup$ EVALDEF (parent $(v)$)] **od od**;
Step 4: **for** $i := 2$ **until** $n$ **do**
            $w :=$ vertex $(i)$;
            **if** dom $(w) \neq$ vertex (semi $(w)$) **then**
                [idef $(w) :=$ idef (dom) $(w)) \cup$ idef $(w)$;]
                dom $(w) :=$ dom (dom $(w)$) **fi od**;
```

Reference [T2] offers two ways to implement LINK, EVAL, and EVALDEF. The simpler method has an $O(m \log n)$ time bound and the more complicated one has an $O(m\alpha(m, n))$ time bound. Farrow [F] provides another $O(m\alpha(m, n))$ method. If we include the $O(l)$ time required to construct def from the program text, then the entire algorithm for computing idef requires $O(m\alpha(m, n)+l)$ time, assuming that each set union requires constant time.

3. Computing simple origins. Once we know def and idef, we can employ the following theorem to compute simple origins. It is convenient for us to assume that idef $(r) = \Sigma$.

THEOREM 3.1. *Let X^v be an entry variable which is a text expression. Then*

$$\text{simple origin } (X^v) = \begin{cases} v \textit{ if } X \in \text{idef } (v), \\ u \textit{ if } X \notin \text{idef } (v) \textit{ and } u \textit{ is the maximal proper dominator of} \\ \qquad v \textit{ such that } X \in \text{def } (u) \cup \text{idef } (u). \end{cases} \tag{7}$$

Proof. Recall that X^v occurs at v. The theorem is immediate from the definitions of simple origin, def, and idef, using the fact that idef $(r) = \Sigma$. □

In order to use Theorem 3.1 efficiently, we need to compute two additional subsets of variables for each vertex. For any vertex $v \in V$, text (v) is the set of variables X such that X^v is a text expression. We can compute text in $O(l)$ time by scanning the program text. For any vertex $v \in V$, relevant (v) is the set of variables X such that, for some vertex w properly dominated by v, X^w is a text expression and X is definition-free between v and w.

THEOREM 3.2. *For any vertex v,*

$$\text{relevant } (v) = \bigcup \{(\text{text } (w) \cup \text{relevant } (w)) - \text{idef } (w) \mid w \in V \text{ and idom } (w) = v\}.$$

Proof. Immediate. □

We can compute relevant in $O(n)$ time by carrying out a depth-first traversal of the dominator tree and processing the vertices in postorder. Note that, for any vertex v, the set relevant $(v) \cap (\text{def}\,(v) \cup \text{idef}\,(v))$ contains exactly the variables X such that, for some vertex w, v is the simple origin of the text expression X^w.

Given text and relevant, we compute simple origins in another depth-first traversal of the dominator tree. During the traversal, we maintain a stack for each variable X. When the traversal reaches a vertex $v \neq r$, stack (X) contains (in dominator order) all proper dominators u of v such that $X \in \text{relevant}\ (u) \cap (\text{def}\,(u) \cup \text{idef}\,(u))$. These vertices are all the candidates (other than v) for the simple origin of X^v. If $X \in \text{idef}\,(v)$, then the simple origin of X^v is v; otherwise the simple origin of X^v is the top vertex on stack (X) when v is reached during the traversal. The following algorithm computes simple origins using this method.

```
procedure TRAVERSE ($v$);
    begin
        for each $X \in$ test($v$) do
            simple origin ($X^v$) := if $X \in$ idef ($v$) then $v$
                                     else top of stack ($X$) fi od;
        for each $X \in$ relevant ($v$) $\cap$ (def ($v$) $\cup$ (idef ($v$)) do
            push $v$ on stack ($X$) od;
        for each $w$ in $\{w | \text{idom}\,(w) = v\}$ do TRAVERSE ($w$) od;
        for each $X \in$ relevant ($v$) $\cap$ (def ($v$) $\cup$ idef ($v$)) do
            pop $v$ from stack ($X$) od
    end TRAVERSE;
for each $X \in \Sigma$ do stack ($X$) = $\varnothing$ od;
TRAVERSE ($r$);
```

The correctness of the algorithm is immediate. To get the algorithm to run fast, we need a method to convert a bit vector representing a set into a list of elements of the set. We can do this in time proportional to the size of the set if we have a variable-length shift operation which shifts a bit vector left to the first nonzero bit and returns the length of the shift. Since such an operation is required to normalize floating-point numbers, it is a machine-language instruction on many computers. Assuming that a variable-length shift requires constant time, the time required to compute simple origins is

$$O\Big(n + \sum_{v \in V} (|\text{text}\,(v)| + |\text{relevant}\,(v) \cap (\text{def}\,(v) \cup \text{idef}\,(v))|)\Big) = O(n+l)$$

since each variable $X \in \text{text}\,(v)$ corresponds to an appearance of X in the program text at vertex v, and each variable $X \in \text{relevant}\,(v) \cap (\text{def}\,(v) \cup \text{idef}\,(v))$ corresponds to a text expression X^w for which v is the simple origin.

4. Computing the simple cover and approximate birthpoints. From the simple origins, it is easy to construct the simple cover Ψ and an approximate birthpoint for each text expression. We begin by constructing a directed acyclic graph (*dag*) to represent all text expressions in the program. We shall call the vertices in this dag *nodes* to distinguish them from the vertices of the control flow graph. The dag has one node representing each text expression. An expression which is a constant sign or an entry variable X^v is represented by a sink labeled by the appropriate constant sign or entry variable; an expression of the form $\theta(E_1, E_2, \cdots, E_k)$ is represented by a node labeled with θ having k (ordered) successors representing the expressions $E_1, E_2, \cdots, E_k$. An example appears in Fig. 5. See [AU], [FKU] for further discussion of this representation. It is easy to construct a dag representing the text expressions in $O(l)$ time.

We convert the dag representing the text expressions into a dag representing the simple cover as follows. We process the sinks of the dag labeled by entry variables X^v in

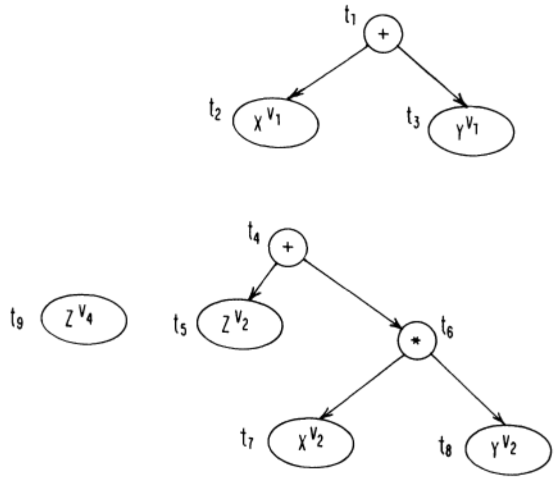

FIG. 5. *Dags representing the text expression of the program in Fig.* 1.

an order consistent with the dominator order; i.e., if v dominates w, we process sinks labeled X^v before sinks labeled X^w. We process sinks labeled X^v as follows. Let u be the simple origin of X^v. If $u = v$ we do nothing. *If* $u \neq v$ and X is defined at u, we replace all edges leading to sinks labeled X^v by edges leading to the node corresponding to exit expression $\mathscr{E}(X, u)$. (This node now represents $\Psi(\mathscr{E}(S, u))$.) If $u \neq v$ and X is not defined at u, we replace the labels X^v by labels X^u. This method requires $O(l)$ time.

We apply two more steps to simplify the resulting dag. First we replace each node all of whose successors represent constants by a sink representing an appropriate constant. We repeat this transformation until it is no longer applicable. This requires $O(l)$ time and produces a dag representing a set of reduced expressions. Next, we merge

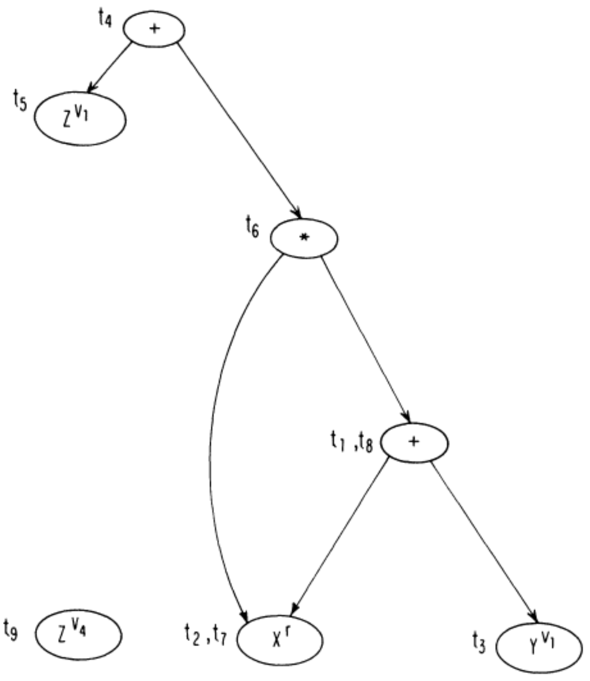

FIG. 6. *Dag representing the simple cover of the program in Fig.* 1.

all nodes representing common subexpressions. This can be done in $O(l)$ time using the acyclic congruence closure algorithm described in [DST]. The result is a dag representing the simple cover. See Fig. 6.

We can compute an approximate birthpoint for each text expression by processing the nodes of the dag representing the simple cover in reverse topological order. Each sink labeled by a constant has approximate birthpoint r. Each sink labeled X^v has approximate birthpoint v. Each node with successors has an approximate birthpoint which is the maximal vertex (with respect to the dominator relation) of the approximate birthpoints of its successors. The approximate birthpoint of a text expression is the approximate birthpoint of the corresponding node in the simple cover dag. (Thus our birthpoints are approximated in part by the simple origins which we computed in § 3.) This computation also requires $O(l)$ time, giving a total of $O(l)$ time to compute both a simple cover and approximate birthpoints.

By combining the algorithms of §§ 2, 3, and 4, we obtain a symbolic evaluation method which requires $O(m\alpha(m, n)+l)$ time if extended bit vector operations require constant time.

Appendix. Graph-theoretic terminology. A *directed graph* $G=(V, E)$ consists of a finite set V of *vertices* and a set E of ordered pairs (v, w) of vertices, called *edges*. If (v, w) is an edge, w is a *successor* of v and v is a *predecessor* of w. A *sink* is a vertex with no successors. A graph $G_1=(V_1, E_1)$ is a *subgraph* of G if $V_1 \subseteq V$ and $E_1 \subseteq E$. A *path* p of *length* k from v to w in G is a sequence of vertices $p=(v=v_0, v_1, \cdots, v_k=w)$ such that $(v_i, v_{i+1}) \in E$ for $0 \leqq i < k$. The path is *simple* if $v_0, \cdots, v_k$ are distinct (except possibly $v_0=v_k$) and the path is a *cycle* if $v_0=v_k$. By convention there is a path of no edges from every vertex to itself but a cycle must contain at least one edge. If $p_1=(u=u_0, u_1, \cdots, u_k=v)$ is a path from u to v and $p_2=(v=v_0, v_1, \cdots, v_l=w)$ is a path from v to w, the path p_1 *followed by* p_2 is $p=(u=u_0, u_1, \cdots, u_k=v=v_0, v_1, \cdots, v_l=w)$. A directed graph is *acyclic* if it contains no cycles. A *topological order* on an acyclic graph is a total ordering of the vertices such that, for each edge (v, w), v is ordered before w.

A *flow graph* $G=(V, E, r)$ is a directed graph (V, E) with a distinguished *start vertex* r such that for any vertex $v \in V$ there is a path from r to v. A (*directed, rooted*) *tree* $T=(V, E, r)$ is a flow graph such that $|E|=|V|-1$. The start vertex r is the *root* of the tree. Any tree is acyclic, and if v is any vertex in a tree T, there is a unique path from r to v. If v and w are vertices in a tree T and there is a tree path from v to w, then v is an *ancestor* of w and w is a *descendant* of v (*denoted by* $v \overset{*}{\rightarrow} w$). If in addition $v \neq w$, then v is a *proper ancestor* of w and w is a *proper descendant* of v (denoted by $v \overset{+}{\rightarrow} w$). If $v \overset{*}{\rightarrow} w$ and (v, w) is an edge of T (denoted by $v \rightarrow w$), then v is the *parent* of w and w is a *child* of v. In a tree each vertex has a unique parent (except the root, which has no parent). If $G=(V, E)$ is a graph and $T=(V', E', r)$ is a tree such that (V', E') is a subgraph of G and $V'=V$, then T is a *spanning tree* of G.

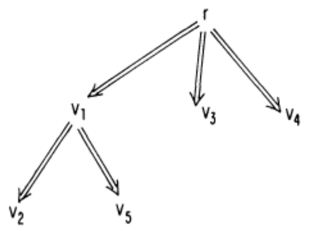

FIG. 7. *Dominator tree of the flow graph given in Fig.* 1. *The symbol* $\Rightarrow$ *leads from* idom (v) *to vertex* v.

If $G = (V, E, r)$ is a flow graph and $u, v \in V$, then *u dominates v* if all paths from r to v contain u. The dominator relation is a partial ordering with minimal element r. If u dominates v and $u \neq v$, then *u properly dominates v*. It can be shown that, for each vertex $v \neq r$, there is a unique vertex u called the *immediate dominator* of v which properly dominates v and is dominated by all other dominators of v. We denote the immediate dominator of v by idom (v). The tree $T = (V, E', r)$ with $E' = \{(\text{idom}(v), v) | v \neq r\}$ is the *dominator tree* of G.

REFERENCES

[AU] A. V. AHO AND J. D. ULLMAN, *Introduction to Compiler Design*, Addison-Wesley, Reading, MA, 1977, pp. 441–477.

[CA] J. COCKE AND F. E. ALLEN, *A catalogue of optimization transformations*, Design and Optimization of Computers, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1971, pp. 1–31.

[DST] P. J. DOWNEY, R. SETHI AND R. E. TARJAN, *Variations on the common subexpression problem*, J. Assoc. Comput. Mach., 27 (1980), pp. 758–771.

[E] C. EARNEST, *Some topics in code optimization*, J. Assoc. Comput. Mach., 21 (1974), pp. 76–102.

[F] R. FARROW, *Efficient variants of path compression in unbalanced trees*, unpublished manuscript (1978).

[FK] R. N. FAIMAN AND A. A. KORTESOJA, *An optimizing Pascal compiler*, IEEE Trans. Software Engineering, SE-6 (1980), pp. 512–519.

[FKU] E. A. FONG, J. B. KAM AND J. D. ULLMAN, *Application of lattice algebra to loop optimization*, Conf. Record Second ACM Symposium on Principles of Programming Languages, January, 1975, pp. 1–9.

[G] C. M. GESCHKE, *Global program optimizations*, Ph.D. thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA, 1972.

[GW] S. GRAHAM AND M. WEGMAN, *A fast and usually linear algorithm for global flow analysis*, J. Assoc. Comput. Mach., 23 (1976), pp. 172–202.

[HU] M. S. HECHT AND J. D. ULLMAN, *Flow graph reducibility*, this Journal, 2 (1972), pp. 188–202.

[Ki] G. A. KILDALL, *A unified approach to global program optimization*, Proc. ACM Symposium on Principles of Programming Languages, Boston, 1973, pp. 194–206.

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

[R1] J. H. REIF, *Code motion*, this Journal, 9 (1980), pp. 375–395.

[R2] ———, *Combinatorial aspects of symbolic program analysis*, Ph.D. thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, MA, 1977.

[RL] J. H. REIF AND H. R. LEWIS, *Symbolic evaluation and the global value graph*, Proc. 4th ACM Symposium on Principles of Programming Languages, 1977.

[Sc] J. T. SCHWARTZ, *Optimization of very high level languages—value transmission and its corollaries*, Computer Languages, 1 (1975), pp. 161–194.

[T1] R. E. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[T2] ———, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

[T3] ———, *A unified approach to path problems*, J. Assoc. Comput. Mach., 22 (1981), to appear.