# SYMBOLIC PROGRAM ANALYSIS IN ALMOST LINEAR TIME

John H. Reif
Department of Computer Science
The University of Rochester

Abstract

A global flow model is assumed; as usual, the flow of control is represented by a digraph called the control flow graph. The objective of our program analysis is the construction of a mapping (a cover) from program text expressions to symbolic expressions for their value holding over all executions of the program. The particular cover constructed by our methods is in general weaker than the covers obtainable by the methods of [Ki, FKU, R1], but our method has the advantage of being very efficient; requiring $O(\ell + a\alpha(a))$ extended bit vector operations (a logical operation or a shift to the first nonzero bit) on all control flow graphs (whether reducible or not), where a is the number of edges of the control flow graph, $\ell$ is the length of the text of the program, and $\alpha$ is Tarjan's function (an extremely slowly growing function).

## 1. Introduction

The flow of control through a program P is represented by the control flow graph $F = (N,A,s)$ where each node $n \in N$ is a block of assignment statements and each edge $(m,n) \in A$ specifies possible flow of control from n to m, and all flow of control begins at the start block $s \in N$. A path in F is a sequence traversing nodes in N linked by edges in A. We assume that for each $n \in N-\{s\}$, there is at least one path from s to n. For m, $n \in N$, m dominates n if all paths from s to n contain m (m properly dominates n if in addition, $n \neq m$). The dominator relation may be represented by a dominator tree such that m dominates n iff m is an ancestor of n. The father of n is the immediate dominator of n.

Let $\Sigma = \{X,Y,Z,\ldots\}$ be the set of program variables occurring globally within P. A program variable $X \in \Sigma$ is defined in some node $n \in N$ if X occurs on the left hand side of an assignment statement of n. For each $n \in N-\{s\}$ and program variable $X \in \Sigma$, we have an input variable $X^{\rightarrow n}$ to denote the value of X on entrance to n. Let EXP be the set of expressions built from input variables and fixed sets of contant signs C and k-adic function signs $\Theta$. For each $n \in N$ and program variable $X \in \Sigma$ defined in n, let the output expression $X^{n\rightarrow}$ be an expression in EXP for the value of X on exit from n in terms of the input variables in block n. A text expression is an output expression or a subexpression of an output expression.

For each $m \in N$ such that n dominates m, program variable X is defined between nodes n and m if X is output on some n-avoiding path from an immediate successor of n to an immediate predecessor of m (otherwise, X is definition-free between n and m). For each $n \in N-\{s\}$, let IN(n) be the set of program variables $X \in \Sigma$ such that $X^{\rightarrow n}$ is a text expression (i.e., $X^{\rightarrow n}$ appears within an output expression of n). The weak environment is a partial mapping W from input variables to N; for each input variable $X^{\rightarrow n}$ such that $X \in IN(n)$, $W(X^{\rightarrow n})$ is the earliest (i.e., closest to the start node s) dominator of n such that X is definition-free between $W(X^{\rightarrow n})$ and n.

We now discuss various applications of the weak environment. For each text expression t located at $n \in N$, the birthpoint of t is the earliest dominator of n to which the computation associated with t may be moved. Code motion is the process of moving code out of control cycles, into new locations where the code is used less frequently. This code improvement requires approximate knowledge of birthpoints, as well as other knowledge including the cycle structure of the control flow graph. (We may not wish to move code as far as the birthpoint since the birthpoint may be contained in control cycles avoiding n; see [CA, AU, E, G, R2] for further discussion of code motion optimizations.) In [R3] it is shown that in the arithmetic domain, the problem of determining birthpoints is recursively unsolvable. [Ki, FKU, R1] present algorithms which may be used to compute approximate birthpoints (i.e., nodes which are dominated by the true birthpoint); however, the time cost of the best of these algorithms is lower-bounded by $C(|\Sigma||A|+\ell)$. We may use the weak environment W to construct a function BIRTHPT mapping text expressions to approximations of their respective birthpoints. For each text expression t, BIRTHPT(t) is the latest (as far as possible from the start node s) node in $\{W(X^{\rightarrow n}) | X^{\rightarrow n}$ is a text expression of $t\}$, relative to the dominator relation. Thus, for each text expression, birthpoint(t) dominates BIRTHPT(t).

An expression $e \in EXP$ covers text expression t if e represents the value of t over all executions of the program. The origin of e is the latest in the chain of nodes on the dominator tree occurring within e (i.e., in the superscripts of the input

variables contained in e). A cover is a mapping from text expressions to covering expressions, and is minimal if the origin of the covering expressions in its range are earliest in the dominator ordering (i.e., as close as possible to the start node s). Note that for each text expression t, the origin of the minimal cover of t is the birthpoint of t. From the weak environment W we can compute the simple cover which is a cover $\psi$ such that for each text expression t, $\psi(t)$ is derived from t by substituting $\psi(X^{m\to})$ for each input variable $X^{\to n}$ such that $m = W(X^{\to n})$ properly dominates n. (Note that this definition requires that X be defined at m; if not, we add at block m the dummy assignment $X := X$ so that $X^{m\to} = X^{\to m}$ is a new text expression. At most $O(\ell)$ dummy assignments must be so added.) See Figure 1 for an example of a simple cover.

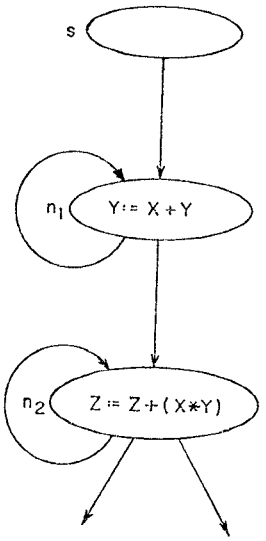


Figure 1:

| Text Expression | Simple Cover |
|---|---|
| $X^{\to n_2}$ | $X^{\to s}$ |
| $Y^{\to n_2}$ | $X^{\to s} + Y^{\to n_1}$ |
| $Z^{\to n_2}$ | $Z^{\to n_2}$ |
| $Z^{n_2\to}$ | $Z^{\to n_2} + (X^{\to s} * (X^{\to s} + Y^{\to n_1}))$ |

A further application of the weak environment involves the global value graphs of [Sc, R1] to represent the flow of values through the program. For certain special global value graphs, the algorithm of [R1] constructs a cover in time almost linear in the size of the global value graph. By a simple, but somewhat inefficient method, we can construct such a special global value graph of size $O(|\Sigma||A|+\ell)$. However, by another method which utilizes the weak environment we can construct a global value graph of size $O(d|A|+\ell)$, where d is a parameter of the program P which may be as large as $|\Sigma|$ but is often constant for block-structured programs. Hence, the very efficient (but weak) symbolic evaluation of this section may serve as a preprocessing step, to speed up a more powerful method for symbolic evaluation presented in [R1].

The organization of this paper is as follows: In the next section we define the relevant graph terminology. In Section 3 we describe an algorithm which constructs a function IDEF giving those program variables defined between nodes and their immediate dominators. The IDEF computation is of a class of path problems that may be efficiently solved by an algorithm due to Tarjan [T4] on reducible flow graphs; we extend his algorithm so as to compute IDEF efficiently on all flow graphs. Section 4 presents an algorithm for constructing the weak environment; this algorithm requires the previously computed function IDEF and contains an interesting data structure for efficiently maintaining multiple symbolic environments. Section 5 concludes the paper with the construction of the simple cover from the weak environment. As in [R1], we collapse the dags (labeled, acyclic digraphs), representing linear blocks of code, into a global dag representing the simple cover.

## 2. Graph Theoretic Notions

A digraph G = (V,E) consists of a set V of elements called nodes and a set E of ordered pairs of nodes called edges. The edge (u,v) departs from u and enters v. We way 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 $p = (u=v_1,v_2,\ldots,v_k=w)$ where $(v_i,v_{i+1}) \in E$ for all i, $1 \le i < k$. The length of the path p is k-1.

The path p may be built by composing subpaths:

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

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

A node u is reachable from a node v if either u = v or there is a path from v to u.

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 digraph is labeled if it is augmented with a mapping whose domain is the vertex set. An oriented digraph is a digraph augmented with an ordering of the edges departing from each node.

A digraph G is acyclic if G contains no cycles. 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 preordering 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 flow graph $G=(V,E,r)$ 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.

## 3. The IDEF Computation

We describe here an efficient algorithm for computing a function IDEF, giving those program variables defined between nodes and their immediate dominators. The IDEF computation is of a class of path problems that may be efficiently solved by an algorithm due to Tarjan [T2] on reducible flow graphs; we extend his algorithm so as to compute IDEF efficiently on all flow graphs. The essence of Tarjan's algorithm is to partition the nodes in N so that for each n, $m \in N$, n and m are in the same dominator strongly connected component (DSCC) iff n and m have the same immediate dominator w and there exists a w-avoiding control cycle containing both m and n. In the case where each DSCC contains but a single node, then by [HU1] the flow graph is reducible, and Tarjan's algorithm runs in time $O(|A|\alpha(|A|))$. Otherwise Tarjan's algorithm must fall back on the less efficient node listing techniques requiring time in the worst case quadratic in $|N|$. For our special problem (the computation of IDEF), the DSCCs may be solved efficiently, yielding an algorithm of cost $O(|A|\alpha(|A|))$ bit vector operations.

We now define the problem more formally. Let $F=(N,A,s)$ be the control flow graph and let DT be the dominator tree of F. For each node $n \in N$ let OUT(n) be the set of program variables defined at n and for each node m properly dominated by n, let DEF(n,m) be the set of program variables defined between n and m. Also, for each $n \in N-\{s\}$ let IDOM(n) be the immediate dominator of n, and let IDEF(N) = DEF(IDOM(n),n), i.e., the set of program variables defined between IDOM(n) and n. The above equation may be inverted as follows:

$$DEF(n,m) = \left(\bigcup_{i=2}^{k} IDEF(z_i)\right) \cup \bigcup_{i=2}^{k-1} OUT(z_i),$$

where $(n=z_1,z_2,\ldots,z_k=m)$ is the dominator chain from n to m. Thus, given the dominator tree DT, DEF and IDEF can be computed from each other.

The algorithm for computing IDEF proceeds in a postorder (leaves to root) scan of the dominator tree DT of F. We compute in one pass IDEF(n) for all sons n of a fixed node w. Clearly this is trivial if w is a leaf of the dominator tree ("son" and "father" refer to the dominator tree DT). Otherwise, a digraph is formed by connecting together those sons of w in DT that are connected in F by paths that avoid w (such paths pass through proper descendants in DT of w only). The strongly connected components of this digraph are DSCCs and may then be processed in topological order; as each is processed, it is identified with the parent node w itself. Thus when all sons of w have been processed, all have been collapsed into w, and the procedure may be repeated on the sons of some other node w'.

To be precise, a set of nodes $S \subseteq N$ is condensed by the following process:

1) Delete the nodes in S from the node set N and add in their place the set S (which is considered to be a new node).

2) Delete each edge entering a node in S and substitute a corresponding edge entering the new node S.

3) Similarly, substitute an edge departing from the new node S for any edge departing from an element of S.

4) Finally, delete any new trivial loops which both depart from and enter the new node S.

Now let $A_w$ consist of the set of edges in A departing from a node other than w and entering a son of w. Such an edge must depart from a proper descendant of w; otherwise, the node it enters would not be dominated by w. For each proper descendant m in DT of w, let H(m,w) be the unique son in DT of w on the dominator chain from w to m, i.e., w immediately dominates H(m,w) which dominates m. Let $G_w = (IDOM^{-1}[w], E_w)$ be a digraph with nodes the sons in DT of w and edges

$$E_w = \{(H(m,w),n) \mid (m,n) \in A_w\}.$$

It is easy to show that:

Lemma 3.1: For each $n,n' \in IDOM^{-1}[w]$, there exists a path in $G_w$ from n to n' iff there exists a w-avoiding path in F from n to n'.

Note that by the above lemma, each strongly connected region of $G_w$ is a DSCC.

The digraph $G_w'$, derived from $G_w$ by condensing each DCSS, is called the condensation of $G_w$ and is obviously acyclic. We shall process each DSCC of $G_w$ in topological order of $G_w'$ (from roots to leaves). In the special case where each DSCC of $G_w$ consists of the singleton set, then F is called reducible ([HU1] give various other characterizations of flow graph reducibility), and Tarjan's algorithm runs in the $O(|A|\alpha(|A|))$. However, in the case that F is nonreducible, various DSCCs will contain two or more nodes and Tarjan's algorithm becomes considerably more expensive and complex. The theorem below expresses IDEF(n) in terms of DEF on previously computed domains; this theorem holds even when the cardinality of a DSCC is greater than 1, giving an efficient method for computing IDEF for all F, both reducible and nonreducible.

Fix a topological order (from roots to leaves) of $G_w'$ and consider a DSCC of $G_w$, say S. Let $m \in N$

be a descendant of w in DT such that H(m,w) is either (1) in S or (2) in some DSCC S' of $G_w$ such that there is a path in $G_w'$ from S' to S (i.e., S' precedes S in the topological order). Then let H'(m,w,S) be H(m,w) in case (1) and w in case (2). That is, H'(m,w,S) is just H(m,w), the unique son in DT of w which is an ancestor of m in DT, unless S' contains H(m,w), in that case; H(m,w) is to be viewed as collapsed into w. The partial function H'(m,w,S) plays a critical role in the inductive correctness proof of our algorithm.

Note that for each node $n \in S$ and edge $(m,n) \in A_w$, H(m,w) satisfies either (1) or (2). We wish to compute IDEF(n). We may assume that due to previous computations, DEF(w,m) is known in case (1), and the value DEF(H(w,m),m) is known in case (2). Our immediate goal is to relate IDEF(n) to these previously computed values.

Call a DSCC S of $G_w$ trivial if S contains a single node {n} and $(n,n) \notin E_w$ and otherwise nontrivial. Now define

$$Q_S^1 = \{\ \} \text{ if S is trivial}$$

and otherwise, if S is nontrivial let

$$Q_S^1 = \bigcup_{n \in S} OUT(n).$$

Also, define

$$Q_S^2 = \bigcup_{\substack{n \in S \\ (m,n) \in A_w}} (DEF(H'(m,w,S),m) \cup OUT(m)).$$

Theorem 3.1: For each $n \in S$, $IDEF(n) = Q_S^1 \cup Q_S^2$. (Note that this characterization of IDEF(n) provides an algorithm for computing IDEF(n) for all sons n of w, by induction on the topological ordering of $G_w'$.)

Proof: Suppose $X \in IDEF(n)$, so there is a path $p = (w = u_1, \ldots, u_k = n)$ such that $X \in OUT(u_i)$ for some $1 < i < k$.

Case 1: If $u_i \in S$, then S must be nontrivial and $X \in OUT(u_i) \subseteq Q_S^1$.

Case 2: Otherwise, suppose $u_i \notin S$. Let $u_j$ be the first node occurring after $u_i$ in p such that $u_j \in S$; then $(u_{j-1}, u_j) \in A_w$.

Case 2.1: If $u_i = u_{j-1}$ then $X \in OUT(u_i) = OUT(u_{j-1}) \subseteq Q_S^2$.

Case 2.2: Otherwise, suppose $u_i \neq u_{j-1}$. Then $H'(u_i,w,S)$ is some $u_{j'}$, $1 \leq j' < i$ such that $u_i$ and $u_{j-1}$ are descendants of $u_{j'}$ in DT. Also note that $u_{j'} = H'(u_{j-1},w,S)$. Then $X \in OUT(u_i) \subseteq DEF(u_{j'},u_{j-1}) = DEF(H'(u_{j-1},w,S),u_{j-1}) \subseteq Q_S^2$.

Now we must show that $X \in Q_S^1 \cup Q_S^2$ implies $X \in IDEF(n)$ for each $n \in S$.

If $X \in Q_S^1$, then X is output from some node $n' \in S$ and S must be nontrivial. Since n' is a son in DT of w, there is a w-avoiding path in F from an immediate successor of w to n'. Also, since S is a nontrivial DSCC of $G_w$, there must be a path in $G_w$ from n' to n. So by Lemma 3.1, there is a w-avoiding path in F from n' to n. Thus, we can construct a w-avoiding path in F from an immediate successor of w to an immediate predecessor of n, and so $X \in IDEF(n)$.

On the other hand, if $X \in Q_S^2$ then $X \in DEF(H'(m,w,S),m) \cup OUT(m)$ for some $(m,n') \in A_w$ and $n' \in S$. Since w dominates H'(m,w,S), $DEF(H'(m,w,S),m) \subseteq DEF(w,m)$. Also, since there is an edge $(m,n') \in A$, $DEF(w,m) \cup OUT(m) \subseteq DEF(w,n')$. Finally, since n,n' are both in S, $IDEF(n) \cup DEF(w,m) \supseteq DEF(H'(m,w,S),m)$. Thus, $DEF(w,n) = DEF(w,n')$, and we conclude that $X \in IDEF(n)$. □

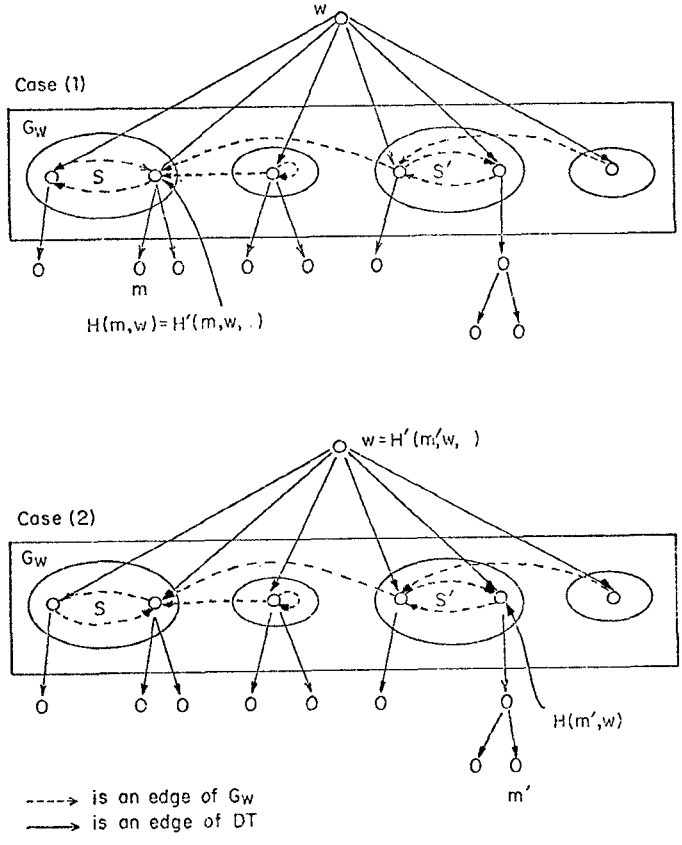


Figure 2.

Cases (1) and (2) of the definition of H'.

Now we use the techniques of Tarjan [T3] to implement our algorithm based on Theorem 3.1. We construct a forest of labeled trees, with node set N. Each edge (n,m) has a label VAL(n,m) containing a set of program variables (in our implementation, the set will be represented by a bit vector). Initially, there is a forest of |N| trees, each consisting of a single node. We shall require three types of instructions:

1) FIND(n) gives the root of the tree currently containing node n.
2) EVAL(n) gives $\bigcup_{i=2}^{k} VAL(n_i,n_{i+1})$ where $(r = n_1, n_2, \ldots, n_k = n)$ is the unique path to n from the current root r

of the tree containing n.

3) LINK(m,n,z) combines the trees rooted at n and m by adding edge (n,m), so n is made the father of m, and sets VAL(n,m) to z.

Tarjan [T2] has shown that a certain algorithm for processing a sequence of r FIND and LINK instructions costs $O((|N|+r)\alpha(|N|+r))$ elementary operations. This algorithm involves path compression on balanced trees and is frequently used in the implementation of UNION-FIND disjoint set operations. Also, Tarjan [T3] gives an almost linear time algorithm (again utilizing path compression) for processing a sequence of FIND, LINK, and EVAL instructions, given that the sequence is known beforehand, except for the values which are to label the edges in the LINK operations.

The following algorithm for computing IDEF uses, like the algorithm of [T4], a preprocessing stage that executes all FIND and LINK instructions but not EVAL instructions; this allows us in the second pass to efficiently process the EVAL as well as the FIND and LINK instructions.

Algorithm A

INPUT Program flow graph F = (N,A,s) and OUT.

OUTPUT IDEF.

```
begin
  declare IDEF: sequence of integers of length |N|;
  Compute the dominator tree DT of F;
  Number the nodes in N by a postordering of DT;
  Scan the below so as to determine the sequence of
  EVAL, FIND, and the first two arguments of the
  LINK instructions;
  for w := 1 to |N| do
    begin
  L0: E_w := A_w := the empty set { };
  L1: for all (m,n) ε A such that IDOM(n) = w and m≠w do
      begin
        add (m,n) to A_w;
        add (FIND(m),n) to E_w;
        comment FIND(m) = H(m,w);
      end;
  L2: Let G'_w be the condensation of
      G_w = (IDOM^-1[w],E_w);
  L3: for each strongly connected component S of G_w
      in topological order of G'_w do
      begin
        comment FIND(m) = H'(m,w,S);
        Q_S := the empty set { };
        comment set Q_S to Q_S^1;
        if S is nontrivial do
          for all n ε S do
            Q_S := Q_S ∪ OUT(n);
        comment add Q_S^2 to Q_S;
        for all n ε S do
          for all (m,n) ε A_w do
            Q_S := Q_S ∪ EVAL(m) ∪ OUT(m);
        for all n ε S do
          begin
        L4: LINK(n,w,Q_S);
            comment apply Theorem 3.1;
            IDEF(n) := Q_S;
          end;
      end;
    end;
end;
```

Theorem 3.2: Algorithm A correctly computes IDEF.

Proof: (Sketch.) By induction in postordering of DT. Initially, each node $n \in N$ is contained in a trivial tree with root n and EVAL(n) gives the empty set { }. Suppose, on entering the main loop at L0 on the w'th iteration, for any node m dominated by w

1) FIND(m) = H(m,w),

2) EVAL(m) = DEF(H(m,w),m).

We require a second induction, this one on the computed topological ordering of $G'_w$. We assume that just before processing the strongly connected region S in $G_w$, for each m dominated by w

1') FIND(m) = H'(m,w,S)

2') EVAL(m) = DEF(H'(m,w,S),m).

By the primary induction hypothesis, (1') and (2') clearly hold for the first DSCC of $G_w$ in the topological ordering.

We first set $Q_S$ to $Q_S^1$

$$= \{\ \} \text{ if S is trivial}$$

$$= \bigcup_{n \in S} OUT(n) \text{ if S is nontrivial}$$

and then add to $Q_S$ the set

$$\bigcup_{\substack{n \in S \\ (m,n) \in A_w}} (EVAL(m) \cup OUT(m))$$

$$= \bigcup_{\substack{n \in S \\ (m,n) \in A_w}} (DEF(H'(m,w,S),m) \cup OUT(m))$$

$$= Q_S^2.$$

Hence by Theorem 3.1, for each $n \in N$, IDEF(n) is correctly set to $Q_S = Q_S^1 \cup Q_S^2$.

Let S' be the DSCC immediately following S in the topological ordering. After executing LINK(n,w,$Q_S$) at L4, for each node m dominated by w such that $H(m,w) \in S$, FIND(m) now gives $w = H'(m,w,S)$ and EVAL(m) now gives $DEF(H(m,w),m) \cup Q_S$

$$= DEF(w,m)$$

$$= DEF(H'(m,w,S'),m)$$

thus completing the second induction proof. Furthermore, just before visiting node w, we have visited all the elements of $IDOM^{-1}[w]$, and so for each m properly dominated by w

1) FIND(m) = w = H(m,w),

2) EVAL(m) = DEF(w,m) = DEF(H(m,w),m)

thus completing the first induction proof. □

Theorem 3.3: Algorithm A costs an almost linear number of bit vector operations.

Proof: The dominator tree may be constructed in almost linear time by an algorithm due to Tarjan [T3].

Now consider the w'th iteration of the main loop. Let $r_w = |IDOM^{-1}[w]|+|A_w|$. Step L1 clearly costs $O(r_w)$ elementary and FIND operations. Step L2 costs $O(r_w)$ elementary steps to discover the strongly connected components of $G_w$ using an algorithm due to Tarjan [T1] plus time linear in $r_w$ to condense each strongly connected component of $G_w$. Finally, at Step L3, we require $O(r_w)$ elementary steps to topologically sort the condensed, acyclic digraph $G'_w$ by an algorithm due to Knuth [Kn], plus $O(r_w)$ bit vector, EVAL, and LINK operations in the loop at L3. The total time cost of this execution of the main loop is this $O(r_w)$ bit vector, EVAL, LINK, and FIND operations. But $2|A|+1 \geq \sum_{w \in N} r_w$. Hence, the preliminary scan of Algorithm A requires $O(|A|)$ LINK and FIND operations implementable in time almost linear in a by the method analyzed in [T2]. With the symbolic sequence of EVAL, LINK, and FIND operations now determined, the second (primary) execution of Algorithm A requires $O(|A|\alpha(|A|))$ bit vector operations by the method of [T4]. □

## 4. Computing the Weak Environment

We now present an algorithm for efficiently computing the weak environment W. For each $n \in N$, let IN(n) be the set of program variables input at n. For the moment, it is useful to represent W by the set of partial functions $\{W_n | n \in N\}$ such that for each $n \in N$, $W_n(X) = W(X^{\to n})$ for all $X \in \Sigma$.

For each $n \in N-\{s\}$ we have

$$W_n(X) = n \text{ if } X \in IDEF(n)$$

$$= W_m(X) \text{ if } X \notin IDEF(n) \text{ and } m \text{ is the immediate dominator of } n.$$

We shall process the nodes in N in preorder (from root to leaves) of the dominator tree. Note that for each $n \in N-\{s\}$, we must store $W_{m_0}$, $W_{m_1},\ldots,W_{m_k}$ where $(s = m_0, m_1, \ldots, m_k = n)$ is the chain of nodes on the dominator tree from the start node s to n. To efficiently maintain these multiple environments, we keep an array of stacks WS such that just before processing node n, the top element of WS(X) is $W_n(X)$ for all $X \in IN(n)$. A prepass is required to insure that elements are not redundantly pushed onto these stacks. The total cost of this data structure for maintaining multiple environments is $O(\ell)$ extended bit vector operations.

To assure that WS is not modified needlessly, we compute R(n) = those program variables X such that $X^{\to m}$ is an input variable for some node m properly dominated by n and such that X is definition-free from n to m. Intuitively, R(n) is a set of program variables whose value is constant on exit to n to some node properly dominated by n. We compute R by a swift postorder walk of the dominator tree DT using the rule:

Lemma 4.1: $R(n) = \bigcup_{m \in IDOM^{-1}(n)} ((IN(m) \cup R(m)) - IDEF(m))$.

The following lemma shows that to correctly maintain WS, we need add node n to the stack WS(X) just in case $X \in R(n) \cap IDEF(n)$.

Lemma 4.2: There exists some m such that $W(X^{\to m}) = n$ and X is input at node m iff $X \in R(n) \cap IDEF(n)$.

Proof: By definition of R, if $X \in R(n)$ then there exists some node $m \in N$ properly dominated by n, X is input at node m, and furthermore, X is definition-free from n to m.

Suppose $W(X^{\to m}) = n$ and X is input at node m. Then clearly X is definition-free from n to m so $X \in R(n)$. But suppose $X \notin IDEF(n)$. Then $W(X^{\to m})$ properly dominates n, which contradicts our assumption that $W(X^{\to m}) = n$. Hence, $x \in R(n) \subseteq IDEF(n)$. □

The usual stack operations will be required:

1) TOP(S) gives the top element of stack S,
2) PUSH(S,z) installs z as the top element of stack S,
3) POP(S) deletes the top element of S.

Algorithm B

INPUT Program flow graph F = (N,A,s), IN, and OUT.

OUTPUT the weak environment W.

```
begin
  Compute IDEF by Algorithm A (as a side effect,
  the dominator tree DT is constructed);
  declare WS := a vector of stacks length |Σ|;
  procedure WEAKVAL(n):
    begin
  L1: for all X ∈ IN(n) do W(X^→n) := TOP(WS(X));
      M := R(n) ∩ IDEF(n);
  L2: for all X ∈ M do PUSH(WS(X),n);
  L3: for all m ∈ IDOM^-1[n] do WEAKVAL(m);
  L4: for all X ∈ M do POP(WS(X));
    end WEAKVAL;
L5: for all n in postorder of DT do
     begin
       R(n) := { };
       for all m ∈ IDOM^-1[n] do
         R(n) := R(n) ∪ ((R(m) ∪ IN(m))-IDEF(m));
     end;
L6: for all program variables X do PUSH(WS(X),s);
L7: WEAKVAL(s);
end;
```

Theorem 4.1: Algorithm B correctly computes the weak environment.

Proof: It is sufficient to show that on each execution of WEAKVAL(n) at label L1:

(*) $W(X^{\to n}) = TOP(WS(X))$ for all $X \in IN(n)$.

This clearly holds on the execution of WEAKVAL(s) at L7, since at label L6 all program variables X

have the top of WS(X) set to s.

Suppose that (*) holds for a fixed $n \in N$. Observe that all nodes pushed in the stacks at L2 are popped out of the stacks at L4. With this observation, we may easily show by a separate induction that the state of WS on exit of any call to WEAKVAL is just as it was on entrance to the call. The state of WS on entrance to WEAKVAL(m) is the same for all $m \in IDOM^{-1}[n]$. Hence, by Lemma 4.2, the claim (*) holds for m, completing our induction proof. □

We shall assume that a single bit vector of length $|\Sigma|$ may be stored in a constant number of words, and we have the usual logical and arithmetic operations on bit vectors, as well as an operation which rotates the bit vector to the left up to the first nonzero element. This operation is generally used for normalization of floating point numbers; here it allows us to determine the position of the first nonzero element of the bit vector in a constant number of such bit vector operations.

Theorem 4.2: Algorithm B costs $O(\ell+|A|\alpha(|A|))$ bit vector operations.

Proof: Each execution of WEAKVAL(n) requires $O(|IDOM^{-1}[n]|+|R(n) \wedge IDOM(n)|)$ bit vector operations. But it is easy to show that

$$|N| < \sum_{n \in N} |IDOM^{-1}[n]|$$

and

$$\ell \leq \sum_{n \in N} |R(n) \wedge IDEF(n)|$$

and so the total cost of all executions of WEAKVAL is $O(\ell+|A|)$ bit vector operations. By Theorem 3.3, the computation of IDEF by Algorithm A costs $O(|A|\alpha(|A|))$ bit vector operations. Hence, the total cost of Algorithm B in bit vector operations is $O(\ell+|A|\alpha(|A|))$. □

## 5. Conclusion: Computing Approximate Birthpoints and the Simple Cover

Given the weak environment constructed by Algorithm B, we can now easily compute approximate birthpoints and construct the simple cover.

Recall that a dag is an acyclic, labeled digraph. Here we assume that the leaves are labeled with either constant signs or input variables. The interior nodes of a dag are labeled with k-adic function signs. For each $n \in N$, the set of text expressions located at n are represented by the dag D(n).

A dag is minimal if it has no redundant subdag and if no proper subdag may be replaced with an equivalent constant sign.

Note that nodes of the dag D(n) represents text expressions whereas the nodes of the control flow graph F represent blocks of assignment statements. Here we wish to construct the function BIRTHPT, which as defined in Section 1 maps from text expressions to their approximate birthpoints in N. Again, for each $n \in N$, we process the nodes of D(n) in topological order, from leaves to roots. Let v to a node in D(n). If v is a leaf labeled with a constant then set BIRTHPT(v) to the start node s. If v is a leaf labeled with an input variable of form $X^{\rightarrow n}$ then set BIRTHPT(v) to n. Recursively, if v is an interior node with every son u previously visited, set BIRTHPT(v) to the latest BIRTHPT(u) (relative to the dominator ordering, with the start node s first) for any such son u.

We use a large, global dag to represent the simple cover. This dag is constructed as follows:

1) First, combine the dags of all the nodes in N. Associate the singleton set {v} with each node v in the resulting dag.

2) Next, compute by Algorithm B the weak environment W. For each $n \in N$ and input variable $X^{\rightarrow n}$ such that $m = W(X^{\rightarrow n})$ properly dominates n, collapse the node corresponding to $X^{\rightarrow n}$ into the node containing $X^{\overrightarrow{m}}$, the output expression for X at m.

3) Finally, minimize the resulting dag.

The above construction takes time $O(\ell+|A|)$, except for the construction of the weak environment which by Theorem 4.2 takes $O(\ell+|A|\alpha(|A|))$ bit vector operations. Hence our method for construction of the simple cover requires $O(\ell+|A|\alpha(|A|))$ bit vector operations.

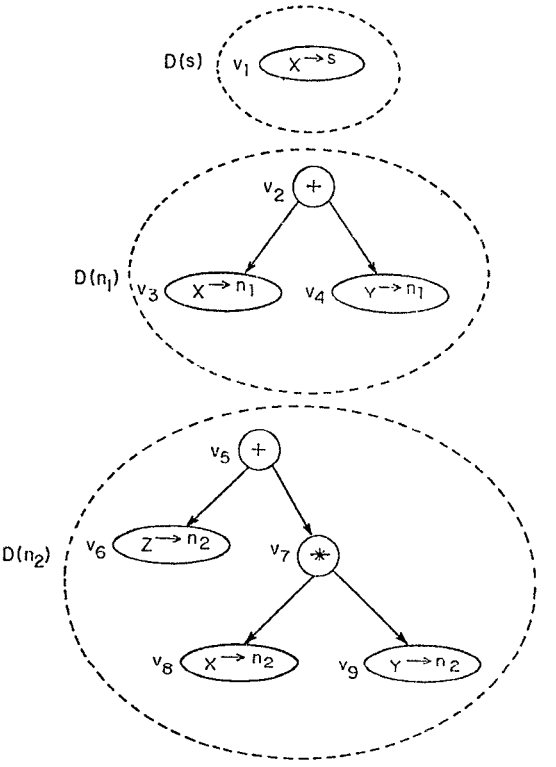


Figure 3 The dags of the program in Figure 1.

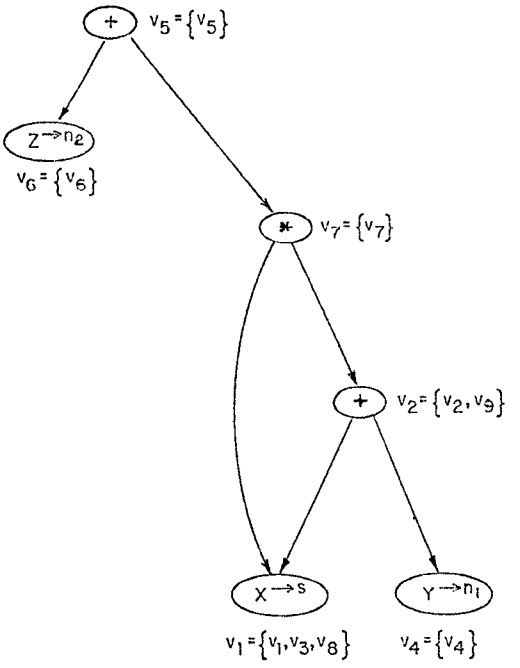


Figure 4 Dag representation of the simple cover.

## References


[AU] Aho, A.V. and Ullman, J.D., Introduction to Compiler Design, to appear.

[E] Earnest, C., Some topics in code optimization, JACM, 21, 1, (Jan. 1974), pp. 76-102.

[FKU] Fong, E.A., Kam, J.B., and Ullman, J.D., Application of lattice algebra to loop optimization, Conf. Record of the 2nd ACM Symp. on Principles of Programming Languages (Jan. 1975), pp. 1-9.

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

[GW] Graham, S., and Wegman, M., A fast and usually linear algorithm for global flow analysis, JACM, 23, No. 1 (Jan. 1976), pp. 172-202.

[HU1] Hecht, M.S. and Ullman, J.D., Flow graph reducibility, SIAM J. Computing, 1, No. 2 (June 1972), pp. 188-202.

[HU2] Hecht, M.S. and Ullman, J.D., Analysis of a simple algorithm for global flow problems, SIAM J. of Computing, 4, 4 (Dec. 1975), pp. 519-532.

[KU1] Kam, J.B. and Ullman, J.D., Global data flow problems and iterative algorithms, J. ACM, 23, 1 (Jan. 1976), pp. 158-171.

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

[Kn] Knuth, D.E., The art of computer programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, MA (1968).

[R1] Reif, J.H. and Lewis, H.R., Symbolic evaluation and the global value graph, 4th ACM Symp. on Principles of Programming Languages (Jan. 1977).

[R2] Reif, J.H., Code motion, Conf. on Theoretical Computer Science, University of Waterloo, Ontario, Canada (1977).

[R3] Reif, J.H., Combinatorial aspects of symbolic program analysis, Ph.D. Thesis, Harvard University, Division of Engineering and Applied Physics (1977).

[Sc] Schwartz, J.T., Optimization of very high level languages--value transmission and its corollaries, Computer Languages, 1, 2 (1975), pp. 161-194.

[T1] Tarjan, R.E., Depth-first search and linear graph algorithms, SIAM J. Computing, 1, 2 (June 1972), pp. 146-160.

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

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

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