CHAPTER 4

SYMBOLIC PROGRAM ANALYSIS IN ALMOST LINEAR TIME

4.0 Summary

We continue to assume a global flow model in which the flow of control is represented by a digraph called the control flow graph. We present an algorithm for symbolic evaluation requiring $O(\ell+a\alpha(a))$ bit vector operations on all flow graphs, where a is the number of edges of the control flow graph, ℓ is the length of the text of the program, and α is Tarjan's function. This algorithm is based on a static analysis of the program and yields a cover called the simple cover which is somewhat weaker than the the covers obtainable by Kildall's algorithm, but still quite useful. Our algorithm may be used to obtain good, approximate birth points for code motion and in addition, this simple cover may be used to speed up the algorithm for computing the more powerful cover of Chapter 2.

4.1 Introduction

As usual, the flow of control through the 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 between n and 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, $n \neq m$).

Let $\Sigma = \{X,Y,Z,\ldots\}$ be the set of program variables occurring globally within P. A program variable $X \in \Sigma$ is defined at 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 a set of expressions built from input variables and fixed sets of constant signs C and k-adic function signs θ. For each $n \in N$ and program variable $X \in \Sigma$ defined at n, let the output expression $X^{n\rightarrow}$ be an expression in EXP for the value of X on exit to 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 occurs within the right hand side of an assignment statement of n before being defined at n. The weak environment is a partial mapping W from input variables to N; for each input variable $X^{\to n}$ such that $X \in IN(n)$, $W(X^{\to n})$ is the earliest (i.e., closest to the start node s) dominator of n such that X is definition-free between $W(X^{\to n})$ and n. We now discuss various applications of the weak environment.

(1) For each text expression t located at $n \in N$, the birthpoint of t is the earliest dominator of n to which the the computation associated with t may be moved. Code motion is the process of moving code as far as possible out of control cycles, into new locations where the code in used less frequenty. This code improvement requires birthpoints, as well as other knowledge including of 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, AU2, E, G] and the next Chapter for further discussion of code motion optimizations.) In Chapter 1 we showed that in the arithmetic domain, the problem of determining birthpoints is recursively unsolvable. We now use the weak environment W to define 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 state node s) node in $\{W(X^{\to n} \mid X^{\to n}$ occurs in n$\}$, relative to the dominator relation.

(2) An expression $\alpha \in$ EXP <u>covers</u> text expression t if α represents the value of t over all executions of the program. The <u>origin</u> of α is the earliest node in the dominator chain occuring within α (i.e., in the super scripts of the input variables contained in α). A <u>cover</u> is a mapping from text expressions to covering expressions, and is <u>minimal</u> if the origin of the covering expressions in its range are as early in the dominator ordering (i.e., as close as possible to the start node s) as possible. Note that the origin of the minimal cover of a text expression is the birthpoint of that text expression. From the weak environment W we can compute the <u>simple cover</u> which is a cover ψ 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.

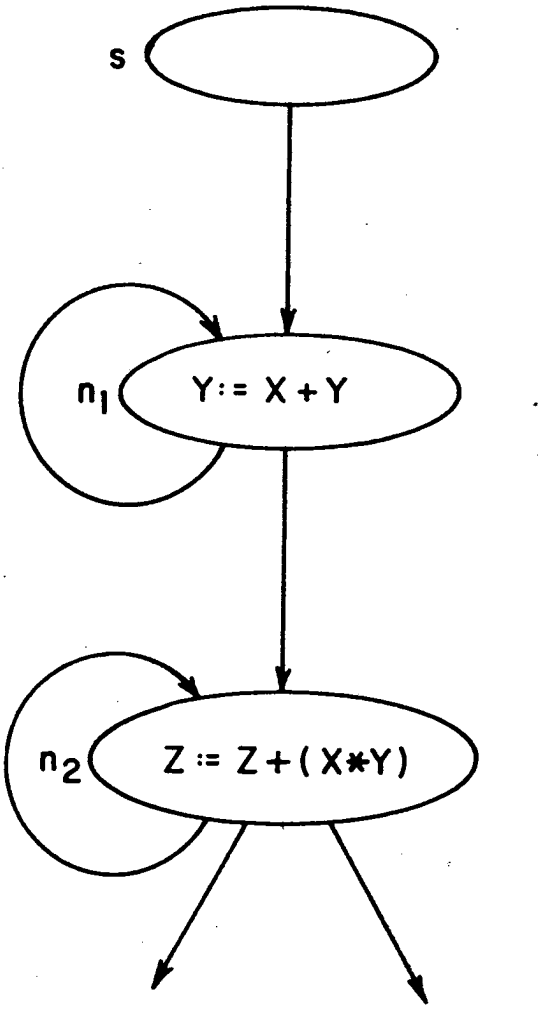


Figure 4.1 $Z^{n\rightarrow} = Z^{\rightarrow n} + (X^{\rightarrow n} * Y^{\rightarrow n})$ has simple cover

$Z^{\rightarrow n} + (X^{\rightarrow s} * (X^{\rightarrow s} + Y^{\rightarrow m}))$.

(3) A further application of the weak environment involves the global value graphs of Chapter 2 to represent the flow of values through the program. Recall that for certain special global value graphs, the algorithm of Chapter 2 constructs a cover in time almost linear in the size of GVG. By a simple, but somewhat inefficient method, we can construct such a special a global value graph GVG^* of size $O(|\Sigma||A|+\ell)$. However, by another method which utilizes the weak environment we can construct a global value graph GVG^+ of size $O(da+\ell)$, where d is a parameter of the program P which may be as large as $|\Sigma|$ but is usually 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 the more powerful method for symbolic evaluation presented in Chapter 2.

The organization of this chapter is as follows:

In the next section 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[T5] on reducible flow graphs; however, we extend his algorithm so as to compute IDEF efficiently on <u>all</u> flow graphs.

Section 4.3 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.

Finally, Section 4.4 concludes the chapter with the construction of the simple cover from the weak environment. As in Chapter 2, we use a large, global dag (labeled, acyclic digraph) to represent the simple cover.

4.2 The Computation of IDEF

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 varaibles 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.

An algorithm by Tarjan[T5] may be used to compute IDEF in a number of bit vector steps almost linear in |A| for a special class of flow graphs called reducible; but his algorithm may cost $\Omega(|N|^2)$ bit vector steps for general flow graphs. Here we extend Tarjan's algorithm so as to compute IDEF in almost linear time for <u>all</u> flow graphs.

Our algorithm for computing IDEF will proceed 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). The essence of the method is to form a digraph 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 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 <u>condensed</u> 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 4.2.1 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'.

The digraph G'_w, derived from G_w by condensing each strongly connected region, is called the condensation of G_w and is obviously acyclic. We shall process each strongly connected region S of G_w in topological order of G'_w (from roots to leaves). In the special case where each such S consists of the singleton set, then F is called reducible ([HU1] give various other characterizations of flow graph reducibility), and Tarjan[T5] provides an efficient method for computing IDEF(n). However, in the case that F is nonreducible, various such S 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 $|S| > 1$, giving an efficient method for computing IDEF for all F, both reducible and non-reducible.

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 strongly connected region S' of G_w such that there is a path in G'_w from S' to S (i.e. S' preceeds S in 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 function H'(m,w,S) plays a critical role in the inductive correctness proof of our algorithm.

Call a strongly connected region S of G_w <u>trivial</u> if S contains a single node {n} and $(n,n) \notin E_w$ and otherwise <u>nontrivial</u>.

Now define

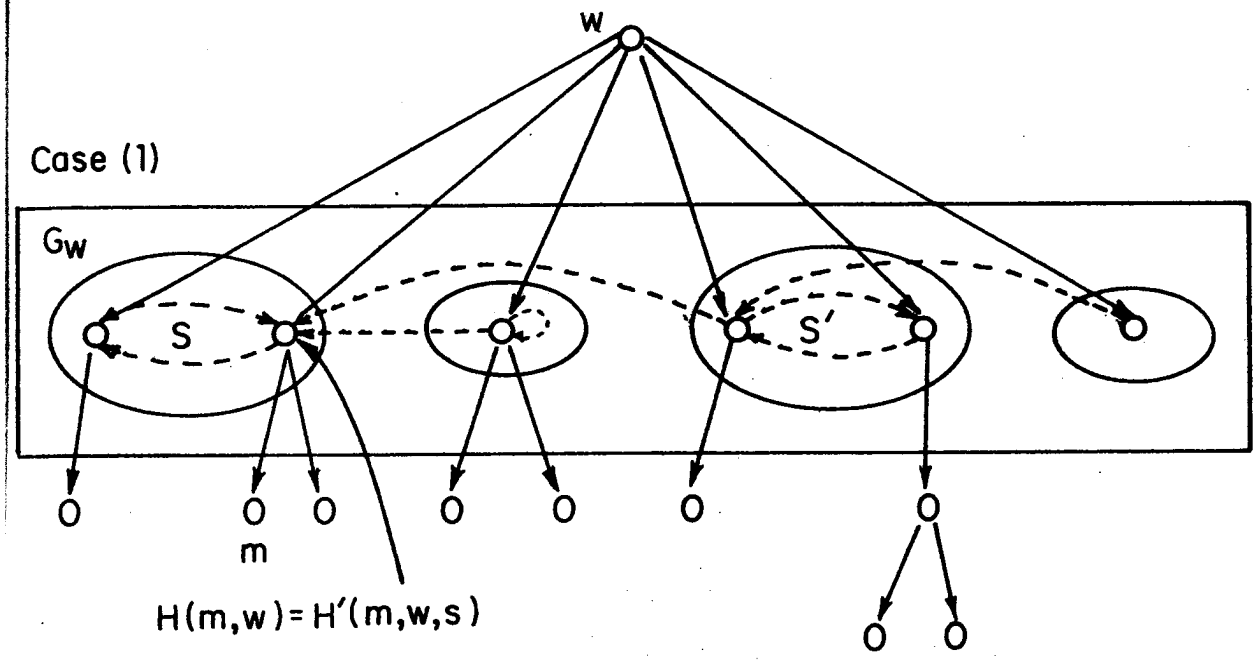
$$Q1_S = \{\} \text{ if S is trivial}$$

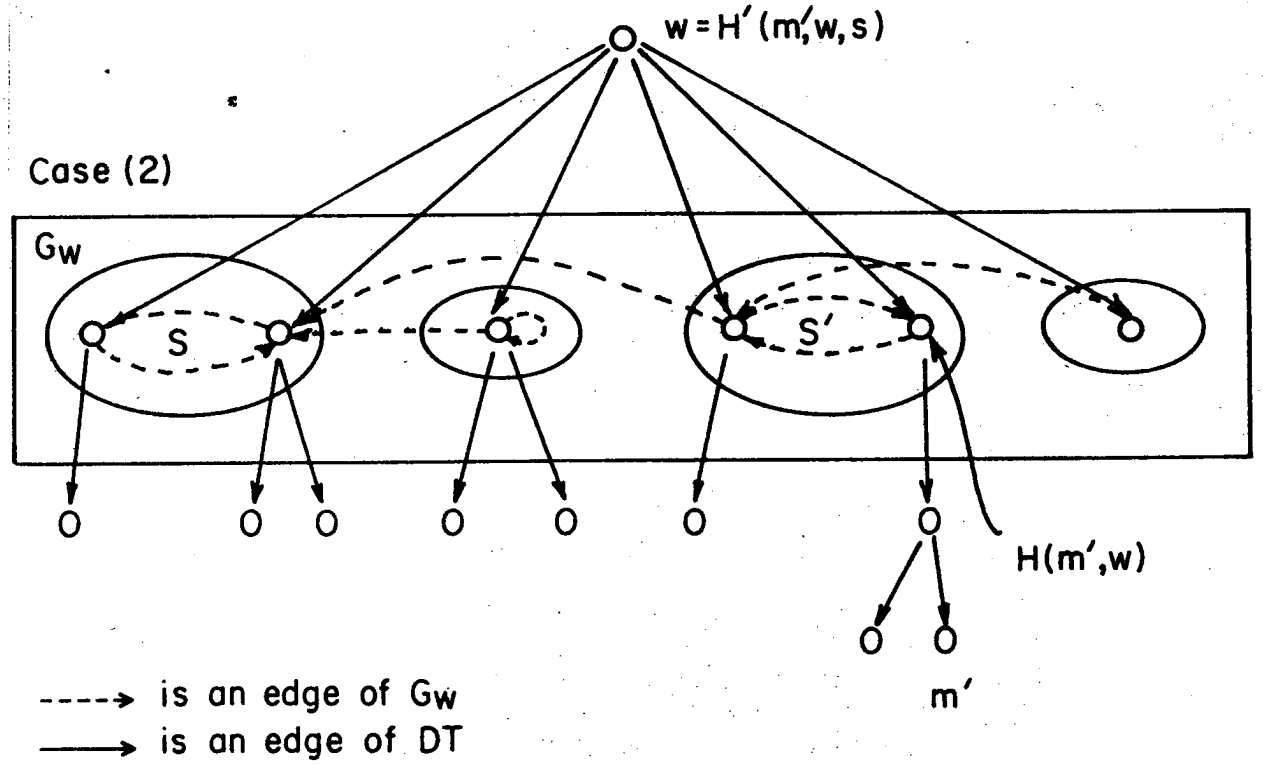
and otherwise, if S is nontrivial let

$$Q1_S = \bigcup_{n \in S} OUT(n).$$

Also, define

$$Q2_S = \bigcup_{\substack{n \in S \\ (m,n) \in A_w}} (DEF(H'(m,w,S),m) \cup OUT(m)).$$





----→ is an edge of Gw

——→ is an edge of DT

Figure 4.2.

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

Theorem 4.2.1 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 strongly connected region of G_w, there must be a path in G_w from n' to n. So by Lemma 4.2.1, 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 $X \in Q^2_S$ 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) $\subseteq$ DEF(w,n) = DEF(w,n') and we conclude that $X \in$ IDEF(n). □

Now we use the techniques of Tarjan[T4] to implement our algorithm based on Theorem 4.2.1. 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[T3] 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[T4] gives an almost linear time algorithm (again utilizing the method of 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 [T5], 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 4A

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 region 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 4.2.1;
            IDEF(n) := Q_S;
          end;
      end;
    end;
end;
```

Theorem 4.2.2 Algorithm 4A 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 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 strongly connected region in the topological ordering.

We first set Q_S to Q_S^1

$= \{\}$ if S is trivial

$= \bigcup_{n \in S} OUT(n)$ 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 4.2.1, for each n ϵ N, IDEF(n) is correctly set to $Q_S = Q_S^1 \cup Q_S^2$.

Let S' be the strongly connected region 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) ϵ 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 4.2.3. Algorithm 4A 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[T4].

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 regions of G_w using an algorithm due to Tarjan[T1] plus time linear

in r_w to condense each stongly connected region in 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[Kn1], 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 4A requires $O(|A|)$ LINK and FIND operations implementable in time almost linear in a by the method analyzed in [T3]. With the symbolic sequence of EVAL, LINK, and FIND operations now determined, the second (primary) execution of Algorithm 4A requires $O(|A|\alpha(|A|))$ bit vector operations by the method of [T4]. □

4.3 The Weak Environment.

We present here an algorithm for computing the weak environment. 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.

For each $n \in N$, let IN(n) be the set of program variables input at n. An array of stacks WS will be used to implement W; so that just before processing node n

$$W(X^{\rightarrow n}) = TOP(WS(X))$$

for all $X \in IN(n)$. This stack implementation of W allows us to maintain W over input variables at n efficiently while visiting proper descendants of n in DT. Note that

$$W(X^{\rightarrow n}) = W(X^{\rightarrow n'})$$

for all n' on the domination chain following $W(X^{\rightarrow n})$ to n. To assure the WS is not modified needlessly, we compute R(n) = those program variables X such that $X^{\rightarrow m}$ 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 4.3.1

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

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 4.3.2 There exists some m such that $W(X^{\rightarrow m}) = n$ 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 $W(X^{\rightarrow m}) = n$ 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 $W(X^{\rightarrow m})$ properly dominates n, which contradicts our assumption that $W(X^{\rightarrow m}) = n$. Hence, $x \in R(n) \subseteq IDEF(n)$. □

Algorithm 4B

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

OUTPUT the weak environment W.

```
begin
  Compute IDEF by Algorithm 4A (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 4.3.1 Algorithm 4B 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 seperate 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 4.3.2, 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 4.3.2. Algorithm 4B costs $O(\ell+|A|\alpha(|A|))$ bit vector

operations.

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

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

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

and so the total cost of all executions of WEAKVAL $O(\ell+|A|)$ bit vector operations. By Theorem 4.2.3, t computation of IDEF by Algorithm 4A costs $O(|A|\alpha(|A|))$ b vector operations. Hence, the total cost of Algorithm 4B bit vector operations is $O(\ell+|A|\alpha(|A|))$. □

4.4 Conclusion: Computing Approximate Birthpoints and the Simple Cover

Given the weak environment constructed by Algorithm 4B, we can now easily compute approximate birthpoints and construct the simple cover.

Recall from Chapter 2 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 4.a maps from text expressions to their approxiamte birthpoints in N. Again, for each $n \in N$, we process the nodes of D(n) in topological order, for leaves to roots. Let v be 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.

As in Chapter 2, 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 4B 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 $X^{m\rightarrow}$, 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 4.3.2 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.

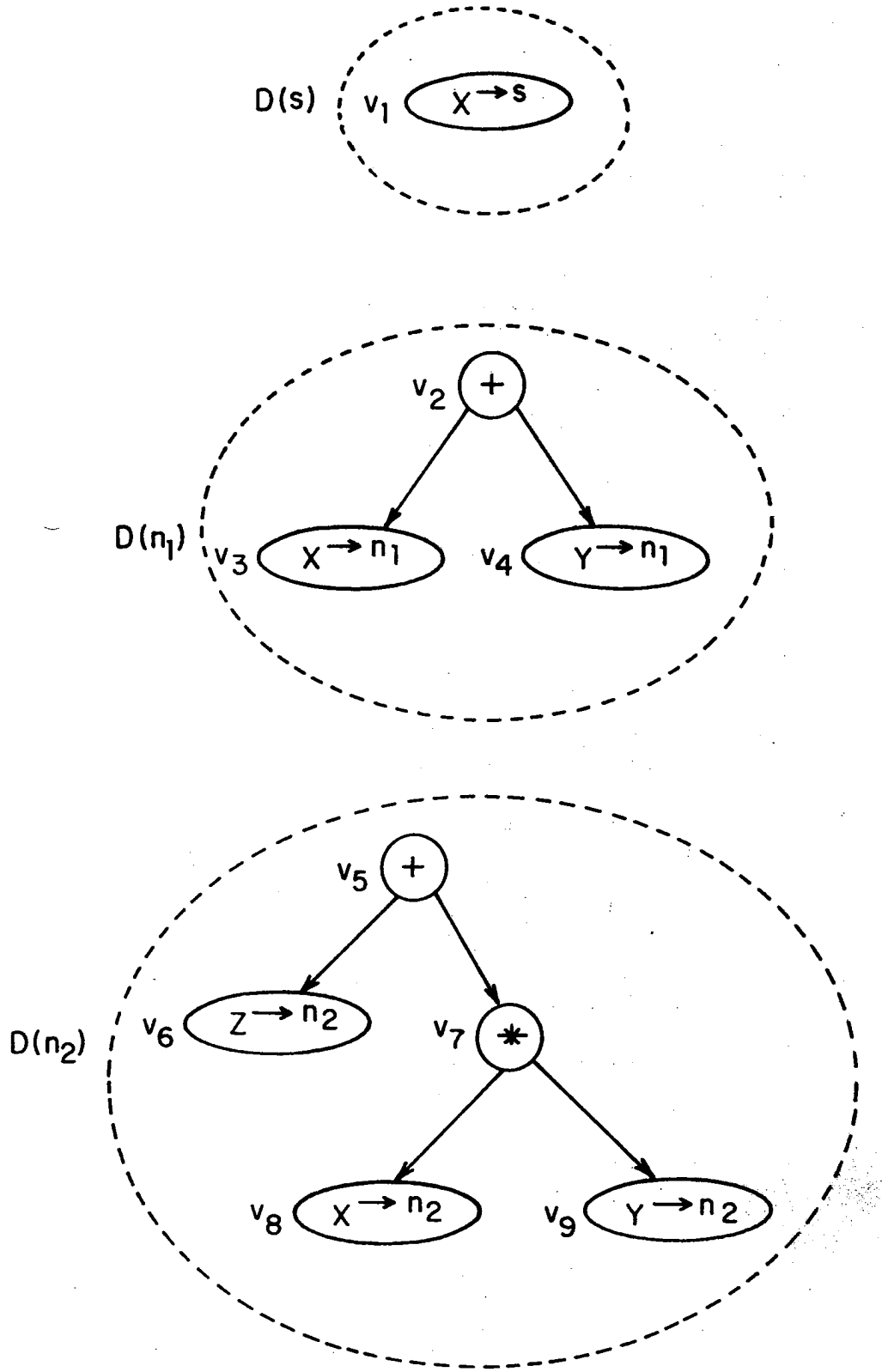


Figure 4.3 The dags of the program in Figure 4.1.

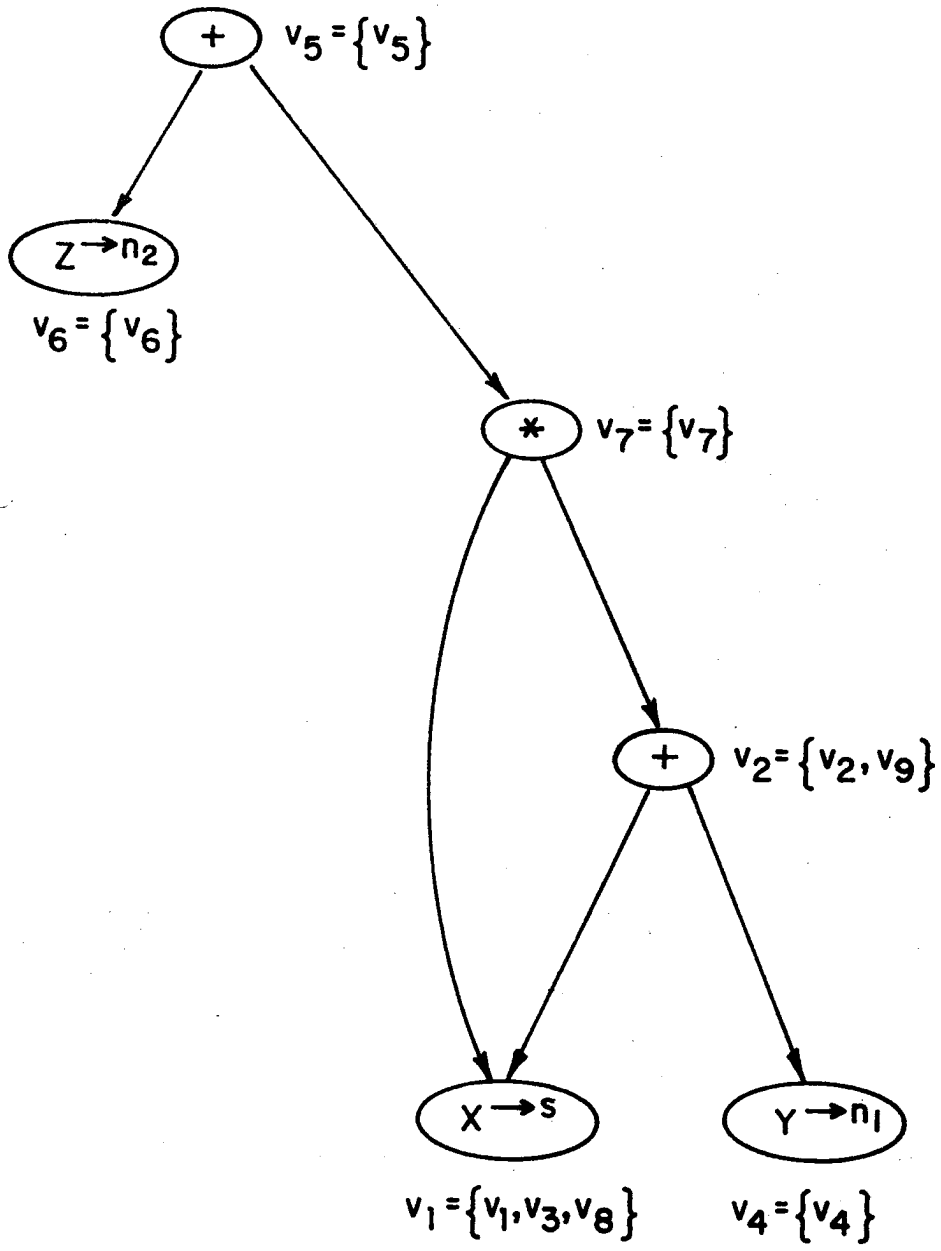


Figure 4.4. Dag representation of the simple cover.