Chapter 5

CODE MOTION

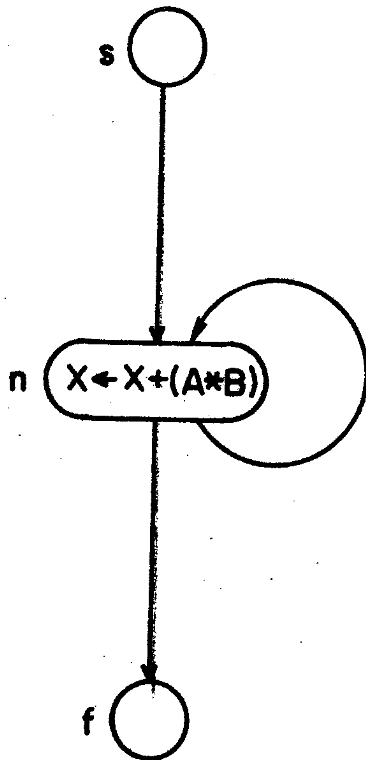
5.0 Summary

Code motion is the program optimization concerned with the movement of code as far as possible out of control cycles into new locations where the code may be executed less frequently. Methods are discussed for approximating certain functions used to ensure that the relocated code may be computed properly and safely, inducing no errors of computation.

The effectiveness of code motion depends on the goodness of our approximation to these functions, as well as on tradeoffs between (1) the primary goal of moving code out of control cycles and (2) the secondary goal of providing that the values resulting from the execution of relocated code are utilized.

Two versions of code motion are formulated: the first emphasizes the primary goal, whereas the other insures that the second goal is not compromised. Almost linear time (in bit vector operations) algorithms are presented for both these formulations of code motion; the algorithm for the first version of code motion is restricted to reducible flow graphs, but the other runs efficiently on _all_ flow graphs.

Previous algorithms for similar formulations of code motion have time cost lower bounded in the worst case by the length of the program text times the number of nodes of the control flow graph.



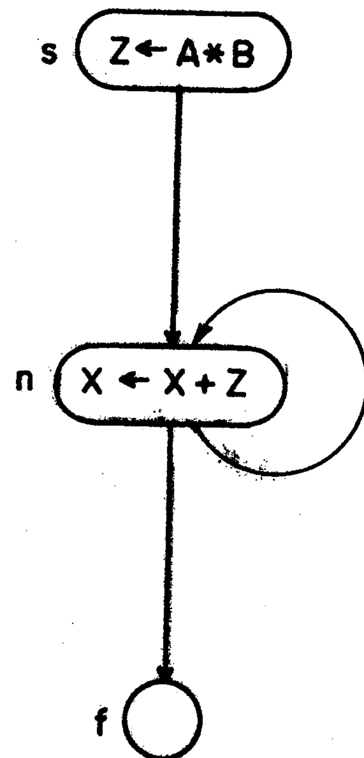


Figure 5.1. A simple example of code motion.

5.1 Introduction

We assume here the global flow model described in Chapter 1. Let $F = (N, A, s)$ be the *control flow graph* of program P to which we wish to apply code motion. Nodes in the set N correspond to linear blocks of code, edges in A specify possible control flow immediately between these blocks, and all flow of control begins at the start node s. A *control path* (*cycle*) is a path (cycle) in F. Every execution of the program P corresponds to a control path, though some control paths may not correspond to possible executions of P. The essential parameters of the model are $n = |N|$, $a = |A|$, and ℓ = the length of the program text (each block in N is assumed to contain at least one text expression, so $n \leq \ell$). We assume bit vectors of length ℓ 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 shifts a 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 steps.) An algorithm runs in *almost linear* number of steps relative to this model if it requires $O((a+\ell)\alpha(a+\ell))$ bit vector and elementary operations, where α is the extremely slow-growing function of [T3] (α is related to a functional inverse of Ackermann's function).

Consider a text expression t located at node loc(t) in N. To effect code motion (see also [CA,AU2,E,G] for descriptions of code motion optimizations) on the computation associated with t, we relocate the computation to a node movept(t) by deleting t from the text of node loc(t) and installing an appropriate text expression t' (not necessarily lexically identical to the string t) at movept(t). On execution of the modified program P' the result of the computation at movept(t) might be stored in a special register or memory location to be retrieved when the execution reaches node loc(t).

To insure that P' is semantically equivalent to the original program P, we require that if node w is the movept of t, then:

R1 All control paths from the start node s to loc(t) contain node w.

R2 The computation is possible at node w; i.e., all quantities required for the computation must be defined at node w.

R3 The computation must be *safe* at w; thus if an error occurs in a particular execution of P', an error must also have occurred in the corresponding execution of the original program P.

Observe that the nodes satifying R1 form a chain, called a *dominator chain*, from s to loc(t). The *birth point* of text expression t is the first node on this chain that satisfies R2. We show in Chapter 1 that if the program P is interpreted within the arithmetic domain, the problem of

computing birth points exactly is recursively unsolvable, so we must be content with computable approximations. Chapters 2 and 4 give algorithms for computing such approximations. The approximation of Chapter 2 is somewhat weaker than that of Chapter 4, but may be computed very swiftly; in fact the algorithm of Chapter 4 requires an almost linear number of bit vector operations for all control flow graphs to compute an approximation BIRTHPT to the true birth point.

The first node on the dominator chain from the birth point of t to loc(t) which satisfies restriction R3 is called the _safe point_ of t. Section 5.3 discusses an approximation to the safe point, called SAFEPT, which may be computed in an almost linear number of bit vector operations, given an efficient test for safety of code motion (we rely on a global flow algorithm by Tarjan[T5] for this). Unfortunately, known algorithms (including Tarjan's) for testing safety of code motion are efficient only on a restricted class of flow graphs which are called _reducible_ (see [HU1]).

Let us continue the formulation of the code motion problem. We add a further restriction:

_R4_ the movept of t may not be contained on a control cycle avoiding loc(t).

Let M_1 consist of all nodes occurring on the dominator chain from SAFEPT(t) to loc(t) that satisfy R4. We choose

movept(t) from the nodes in M_1 based on the following goals:

<u>G1</u> movept(t) is to be located on as few control cycles as possible.

<u>G2</u> As few control paths as possible may contain movept(t) and reach the final node f in N without passing through loc(t) (we assume f is reachable from all nodes).

The above goals conflict, for to satisfy G1 we would choose movept(t) earlier in the dominator ordering than we would if we were to also satisfy goal G2.

We consider two formulations of code motion. In the first formulation we stress G1 and in the other we stress G2. Let M_2 be the set of nodes in M_1 which also satisfy the restriction:

<u>R5</u> All control paths from the movept of t to the final node f must contain loc(t).

For i = 1,2 let

M_i' = those nodes in M_i which satisfy R4 and are contained in the minimum number of control cycles

and let $movept_i(t)$ be the first node in M_i' relative to the dominator ordering of F.

More general formulations of code motion have been described by Geschke[G], including the movement of code to several nodes (rather than to a single node), and also the movement of code to nodes occurring after (rather than before) loc(t) in the dominator ordering of F. Previous formulations of code motion [E,G,CA] similar to ours require $\Omega(\ell)$ (the "big omega" notation denotes a lower bound in the

worst case; see [Kn2]) operations per node in the flow graph, or a total worst-case time cost of $\Omega(\ell \cdot n)$.

The next Section defines the relevant digraph terminology. Section 5.3 presents an algorithm for computing SAFEPT, using Tarjan's algorithm for testing safety of code movement. Section 5.4 reduces the first version of code motion to the computation of SAFEPT and a pair of functions C1 and C2 related to the cycle structure of flow graphs. We show that the function C1 suffices to solve the second type of code motion; in this formulation we avoid testing for safety of code motion. Sections 5.5 and 5.6 present algorithms for computing the functions C1 and C2 over certain domains in an almost linear number of bit vector operations. The algorithm for computing C2 requires a special function DDP; in Section 5.7 an algorithm, restricted to reducible flow graphs, is presented which computes DDP in $O(|A|\alpha(|A|))$ bit vector steps. We conclude in Section 5.8 with a graph transformation (similar to those described in [E,AU2]) which improves the results obtained from the two versions of code motion and in certain cases simplifies our algorithms for computing C1 and C2.

5.2 Graph Theoretic Notions

A *digraph* $G = (V, E)$ consists of a set V of elements called *nodes* and a set E of ordered pairs of nodes called *edges*. The edge (u,v) *departs from* u and *enters* v. We say u is 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 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 path is *simple* if it contains no cycles.

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

A *flow graph* (V, E, r) is a triple such that (V, E) is a digraph and r is a distinguished node in V, the *root*, such that r has no predecessors and every node in V is reachable from r.

A digraph is *acyclic* if it contains no cycles. If u is reachable from v, u is a *descendant* of v and v is a *ancestor*

of u (these relations are *proper* if u ≠ v). Immediate successors are called *sons*. An acyclic flow graph T is a *tree* if every node v other than the root has a unique immediate predecessor, the *father* of v. T is *oriented* if the edges departing from each node are oriented from left to right.

The *preordering* of oriented tree T is defined by the following algorithm (see also Knuth[Kn1]).

Algorithm 5A

INPUT An oriented tree T with root r.

OUTPUT A numbering of the nodes of T.

```
begin
  procedure PREORDER(w):
    begin
      if w is unnumbered then
        begin
          Let w be numbered k := k+1;
          for all sons u of w from left to right do
            PREORDER(u);
        end;
    end;
  k := 0;
  PREORDER(r);
end;
```

Given a preordering, we can (see [T1]) test in constant time if any particular pair of nodes is in the ancestor relation. if a node is an ancestor of any other. A *postordering* is the reverse of a preordering.

Let G = (V, E, r) be an arbitary flow graph. A *spanning tree* of G is an oriented tree ST rooted at r with node set V and edge list contained in E. The edges

contained in ST are called _tree edges_, edges in E from descendants to ancestors in ST are called _cycle edges_, non-tree edges in E from ancestors to their descendants in ST are _forward edges_, and edges in E between nodes unrelated in ST are _cross edges_.

A special spanning tree of G, called a _depth-first search spanning tree_ is constructed by a linear time algorithm by Tarjan[T1] and has the property that if the nodes are preordered by the algorithm above, then for each cross edge (u,v), v is preordered before u.

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.

The cycle edges are partitioned by their relation in the dominator tree DT:

(a) A-cycle edges are cycle edges from a node to a proper dominator.

(b) B-cycle edges are cycle edges between nodes unrelated on the dominator tree.

G is _reducible_ if each cycle p of G contains a unique node dominating all other nodes in p. Programs written in a

well-structured manner are often reducible. Various characterizations of reducibility are given by Hecht and Ullman[HU1]; in particular they show that

Theorem 5.2 G is reducible iff G has no B-cycle edges.

Tarjan gives in [T2] a test for reducibility requiring an almost linear number of elementary steps.

5.3 Approximate Safe Points of Code Motion

Text expression t is safe at node w if no new errors of computation are induced when t is relocated to node w. To approximate the safe point of t we require a good method for determining if t is safe at particular nodes.

A text expression t is dependent on a program variable if that variable occurs within the text of t (this need not imply functional dependence). The text expression t is dangerous if there exists some assignment of values to the variables dependent on t which induce an error in the computation of t. For example, an expression with a division operation is dangerous, since an error occurs if the operand evaluates to zero. Following Kennedy[Ke1], we say there is an exposed instance of text expression t on a simple (acyclic) control path p if there is some text expression t' located in p, with the same text string as t, and such that no variable on which t is dependent is defined at any node in p occurring after the first node of p and before loc(t'). Let SAFE(w) consist of all text expressions which are not dangerous plus all dangerous text expressions which have an exposed instance on every simple control path from w to the final node f.

Theorem 5.3.1 (due to Kennedy[Ke1]) If w occurs on the dominator chain from BIRTHPT(t) to loc(t) and t $\in$ SAFE(w) then t is safe at node w.

Proof Let P' be the program derived from P by relocating the computation of t to node w. If there is an error resulting from the computation of t on control path p in the modified program, then since $t \in$ SAFE(w) the error would also have occurred (although somewhat later) in the execution of the original program on control path p. □

Recall the parameters $n = |N|$, $a = |A|$, and ℓ = the number of text expressions. Tarjan[T5] presents an algorithm for solving certain general path problems, and which may be used to compute SAFE in a number of bit vector operations almost linear in $a+\ell$ if the program flow graph is reducible. Also, **Graham and Wegman [GW] and** Hecht and Ullman[HU2] give algorithms for computing SAFE with time cost often linear in $\ell+a$, but with worst case time cost $\Omega(\ell+a\cdot\log(a))$ and $\Omega(\ell+n^2)$, respectively.

Let loc(t) be the node where text expression t is located. To approximate the safe point of t, we take SAFEPT(t) to be the first node w of the dominator chain from BIRTHPT(t) to loc(t) such that $t \in$ SAFE(w).

Let IDOM map from nodes in N-{s} to their immediate dominators in F. For each $w \in N$, let EARLY(w) consist of those text expressions t with BIRTHPT(t) = w plus, if $w \neq s$, all $t \in$ EARLY(IDOM(w))-SAFE(IDOM(w)). Let LATE(w) be the set of all text expressions $t \in$ SAFE(w) such that w dominates loc(t).

Lemma 5.3.1 SAFEPT(t) = w iff t $\in$ EARLY(w) $\cap$ LATE(w).

Proof. Clearly, for each node w on the dominator chain from BIRTHPOINT(t) to loc(t), t $\in$ LATE(w) iff SAFEPT(t) dominates w. Hence, for each node w on the dominator chain following BIRTHPT(t) to SAFEPT(t), if t $\in$ EARLY(IDOM(w)) then since t $\notin$ SAFE(IDOM(w)), t $\in$ EARLY(w). Also for any w on the dominator chain following SAFEPT(t) to loc(t), t $\in$ SAFE(IDOM(w)), so t $\notin$ EARLY(w). Thus w = SAFEPT(t)

iff w dominates SAFEPT(t) and SAFEPT(t) dominates w

iff t $\in$ EARLY(w) $\cap$ LATE(w). □

Lemma 5.3.1 leads to a simple algorithm for computing SAFEPT. EARLY is computed by a preorder pass through the dominator tree DT and LATE is computed by a postorder (i.e. reverse of the preorder of Section 5.2) pass through DT.

Algorithm 5B

INPUT Control flow graph F = (N, A, s), the set of text expressions TEXT, BIRTHPT, and SAFE.

OUTPUT SAFEPT.

```
begin
  declare LATE, EARLY := arrays length n = |N|;
  declare SAFEPT := array length ℓ;
  Compute the dominator tree DT of F;
  Number nodes in N by a preordering of DT;
  for w := 1 to n do
  L1: EARLY(w) := LATE(w) := the empty set {};
  for all text expressions t ε TEXT do
  L2: add t to EARLY(BIRTHPT(t)) and LATE(loc(t));
  for w := 1 to n do
  L3: EARLY(w) := EARLY(w)
                ∪ (EARLY(IDOM(w))-SAFE(IDOM(w)));
  for w := n by -1 to 1 do
    begin
      for all sons u of w in DT do
      L4: LATE(w) = LATE(w) ∪ LATE(u);
      comment Apply Lemma 5.3.1;
      for all t ε EARLY(w) ∩ LATE(w) do
      L5: SAFEPT(t) := w;
    end;
end;
```

We assume that a bit vector of length ℓ may be stored in a constant number of words and that in a constant number of bit vector operations we may determine the first nonzero element of a bit vector (this is not an unreasonable assumption since most machines have an instruction for left-justifying a word to the first nonzero bit).

<u>Theorem</u> <u>5.3.1</u> Algorithm 5B is correct and requires $O((a+\ell)\alpha(a+\ell))$ elementary and bit vector operations.

<u>Proof</u>. The correctness of Algorithm 5B follows immediately from Lemma 5.3.1. The dominator tree DT may be constructed by an algorithm by Tarjan[T4] in time almost linear in $a = |A|$, (if G is reducible, an algorithm due to Hecht and Ullman[HU2] computes DT in a linear number of bit vector operations). Steps L1, L2, L3, L4, L5 each require a constant number of elementary and bit vector operations and are executed $O(n)$, $O(\ell)$, $O(n)$, $O(n)$, $O(\ell)$ times, respectively. Since F is a flow graph, $a \geq n-1$. Hence, the total time cost of Algorithm 5B is $O((a+\ell)\alpha(a+\ell))$ bit vector operations. □

5.4 Reduction of Code Motion to Cycle Problems

For an arbitrary flow graph $G = (V, E, r)$ and $w,x \in V$ such that w dominates x in G, let $C1_G(w,x)$ be the latest node, on the dominator chain in G from w to x, which is contained on no w-avoiding cycles. Similarly, let $C2_G(w,x)$ be the first node, on this dominator chain, which is contained on no x-avoiding cycles.

Lemma 5.4.1. For nodes $x,y \in V$ such that y dominates x, let M be the list of nodes on the dominator chain from y to x and contained on no x-avoiding cycles, let w be the first element of M, and let M' = those nodes in M contained in a minimal number of cycles. Then $C1_G(w,x)$ is the first node in M' relative to the dominator ordering of G.

Proof Observe that $C1_G(w,x) \in M$; for otherwise $C1_G(w,x)$ is contained on a x-avoiding cycle which also contains w, a contradiction with the assumption that $w \in M$ is contained on no x-avoiding cycles.

Suppose p is a cycle containing $C1_G(w,x)$ and avoiding some $y \in M-\{C1_G(w,x)\}$. If y properly dominates $C1_G(w,x)$ then since w dominates y, p is w-avoiding, a contradiction with the assumption that $C1_G(w,x)$ is contained on no w-avoiding cycles. Otherwise, if y is properly dominated by $C1_G(w,x)$, then since y dominates w, p is x-avoiding, contradicting the assumption that $C1_G(w,x) \in M$.

Suppose some $z \in M'$ properly dominates $C1_G(w,x)$. If z is contained on no w-avoiding cycles, then $C1_G(w,x)$ dominates z, contradiction. If z is contained on a w-avoiding cycle, then so is $C1_G(w,x)$, a contradiction. □

Let $F = (N, A, s)$ be the control flow graph. Our first variation of code movement, $movept_1$, may be described in terms of $C1_F$, $C2_F$, and SAFEPT.

<u>Theorem 5.4.1</u> For each text expression t,

$$movept_1(t) = C1_F(C2_F(SAFEPT(t),loc(t)),loc(t)).$$

<u>Proof</u>. Clearly, any node on the dominator chain from SAFEPT(t) to loc(t) satisfies R1-R3. Recall that M_1 consists of those nodes on the dominator chain from SAFEPT(t) to loc(t) which satisfy R4; i.e., they are contained on no control cycles avoiding loc(t). By definition of $C2_F$, $w = C2_F(SAFEPT(t),loc(t))$ is the first node in M_1 relative to the domination ordering in F. Hence by Lemma 5.4.1, $movept_1(t) = C1_G(w,loc(t))$ is the first node of M_1' relative to the domination ordering. □

From the control flow graph $F = (N, A, s)$ we derive the <u>reverse control flow graph</u> $R = (N, A_R, f)$ which is a digraph rooted at the <u>final</u> node $f \in N$ and with edge set A_R derived from A by reversing all edges. R is assumed to be a flow graph; so every node is reachable in R from f.

<u>Lemma 5.4.2</u> If x dominates y in F, y dominates z in F, and z dominates x in R, then y dominates x in R and z dominates y

in R.

Proof by contradiction. Suppose there is a y-avoiding path p_1 in R from f to x. Since z dominates x in R, p_1 must contain z. The reverse of p_1, p_1^R, is a path in F. Since x dominates y in F, there must be a y-avoiding path p_2 in F from s to x. Composing p_2 and p_1^R, we have a path in F from s to f which contains z but avoids y. But this contradicts our assumption that y dominates z in F. Hence, y dominates x in R. Similarly, we may easily show that z dominates y in R. □

Theorem 5.4.2 If w dominates x in F and x dominates w in R, then $C2_F(w,x) = C1_R(x,w)$.

Proof. It is sufficient to observe by Lemma 5.4.2 that the dominator chain from w to x in F is the reverse of the dominator chain from x to w in R. The symmetries in the definition of $C2_F$ and $C1_F$ then give the result. □

Let HPT(t) be the first node on the dominator chain of F from BIRTHPT(t) to loc(t) which is dominated by loc(t) in the reverse flow graph R. Also, for each $w \in N$ let H(w) be the first node, on the dominator chain in F from the start node s to w, which is dominated in R by w. H may be computed by a swift scan of the nodes in N, in preorder of the dominator tree of F by the following rule:

H(w) = H(x) if w dominates x in R, where x is the immediate dominator of w in F, and otherwise H(w) = w.

HPT is given from H by the following lemma, which is trivial to prove.

Lemma 5.4.3 HPT(t) = H(loc(t)) if BIRTHPT(t) dominates H(loc(t)) in F and otherwise HPT(t) = BIRTHPT(t).

The following Theorem expresses $movept_2$ in terms of C1 and HPT.

Theorem 5.4.3 For all text expressions t,

$$movept_2(t) = C1_F(C1_R(loc(t),HPT(t)),loc(t)).$$

Proof. Recall that M_2 is the set of nodes $v \in M_1$ which satisfy restriction R5: that all control paths from v to f contain loc(t).

We claim that w = $C2_F$(HPT(t),loc(t)) is the first node in M_1 relative to the dominator ordering of F. Since SAFEPT(t) dominates HPT(t) in F, w is clearly an element of M_1. If there exists some $w' \in M_2$ which properly dominates w, then since w' satisfies restriction R5, loc(t) is contained on all paths from w' to f, which implies that HPT(t) dominates w', a contradiction.

By Theorem 5.4.2, w = $C2_F$(HPT(t),loc(t)) = $C1_R$(loc(t),HPT(t)). Hence, $movept_2(t)$ = $C1_F$(w,loc(t)) is the first node in M_2' relative to the domination ordering. □

The next two sections describe how to compute C1 and C2 efficiently.

5.5 The Computation of C1

Let G = (V, E, r) be an arbitary flow graph with the nodes of V numbered from 1 to n = |V| by a preordering of some depth-first search spanning tree ST of G (see Section 5.2 for definitions of depth-first spanning trees and preorderings). For certain w,x ε V such that w dominates x in G, we wish to compute $C1_G(w,x)$; recall from Section 5.4 that this is the last node on the dominator chain from w to x which is contained on no w-avoiding cycles.

For w = n,n-1,...,2 let I(w) be the set of all x ε V contained on a cycle of G consisting only of descendants of w in ST, and such that x is not contained in any I(u)-{u} for u > w. The sets I(n), I(n-1),...,I(2) are related to the *intervals* of G (see Allen[A]) and may be computed in almost linear time by an algorithm of Tarjan[T2].

Let IDOM(x) give the immediate dominator of node x ε V-{r}.

Lemma 5.5.1 (due to Tarjan[T2]) For each w ε V-{r} and x ε I(w), IDOM(w) properly dominates x.

Proof by contradiction. Suppose the lemma does not hold; so there exists a IDOM(w)-avoiding path p from the root r to x. But by definition of I(w), there exists a cycle q, avoiding all proper ancestors of w in ST and containing both w and x. Since IDOM(w) is a proper ancestor of w in ST, q avoids IDOM(w). Hence, we can construct from p and q a

IDOM(w)-avoiding path from r to w, which is impossible. □

Our algorithm for computing C1 will construct, for each $w \in V$, a partition PV(w) of the node set V. Initially, for $w = n$, PV(w) consists of all singleton sets named for the nodes which they contain. For $w = n, n-1, \ldots, 2$ let J(w) consist of I(w) plus all nodes in V contained on a w-avoiding cycle and immediately dominated by some element of I(w). Then PV(w-1) is derived from PV(w) by collapsing into w all sets with at least one element contained in J(w)-{w}.

For $w,x \in V$ such that w dominates x, let g(w,x) be the name of the set of PV(w) in which x is contained.

Lemma 5.5.2 g(w,x) is an ancestor of x in ST and if $w > 1$, IDOM(g(w,x)) properly dominates x.

Proof by induction on w.

Basis step. For $w = n$, $g(w,x) = x$ and so IDOM(g(w,x)) = IDOM(x) properly dominates x.

Inductive step. Suppose, for some $w > 1$, the Lemma holds for all $w' \geq w$. Consider some $x \in V$ such that w dominates x.

Case 1. If $g(w-1,x) = g(w,x)$ then the Lemma holds by the induction hypothesis.

Case 2. If $g(w-1,x) = w$ then in PV(w), g(w,x) contains some $y \in J(w)-\{w\}$. First we show that w is an ancestor of y in ST and IDOM(w) properly dominates y. If $y \in I(w)-\{w\}$, then

w is an ancestor of y in ST by definition of I(w), and IDOM(w) dominates y by Lemma 5.5.1. Otherwise, suppose y ϵ (J(w)-I(w))-{w} so y is immediately dominated by some y' ϵ I(w). Hence y' is a proper ancestor of y in ST and by definition of I(w), w is an ancestor of y', so w is an ancestor of y' in ST. By Lemma 5.5.1, IDOM(w) properly dominates y', and hence IDOM(w) also properly dominates y.

Since the set g(w,x) of PV(w) contains y, g(w,x) = g(w,y). By the induction hypothesis, g(w,x) = g(w,y) is an ancestor of both x and y in ST. We have shown that w is an ancestor of y in ST. Since w < g(w,x), w is a proper ancestor of g(w,x) in ST, so w is also an ancestor of x in ST.

We claim that IDOM(w) properly dominates g(w,x). If not, there would exist an IDOM(w)-avoiding path p from the root r = 1 to g(w,x). IDOM(w) is an ancestor of w in ST and g(w,x) is not an ancestor of w, so g(w,x) is not an ancestor of IDOM(w) in ST. Also, since g(w,x) is an ancestor of y in ST, there is a IDOM(w)-avoiding path p' of tree edges from g(w,x) to y. Composing p and p', we have a IDOM(w)-avoiding path from r to g(w,x), which is impossible since we have previously shown that IDOM(w) properly dominates y. Hence, IDOM(w) properly dominates g(w,x). By the induction hypothesis, IDOM(g(w,x)) properly dominates x, and so IDOM(w) properly dominates x. □

<u>Theorem</u> <u>5.5.1</u>. Consider any $x, w \in V$ such that w dominates x. If x is contained in no w-avoiding cycles then $g(w,x) = x$ and otherwise $g(w,x)$ is the highest ancestor of x in ST such that IDOM($g(w,x)$) properly dominates x and all nodes, on the dominator chain following IDOM($g(w,x)$) to x, are contained in w-avoiding cycles.

<u>Proof</u> (sketch). If x is contained in no w-avoiding cycles in G then x can not be contained in $I(w')$ for $w < w' \leq x$ and so in this case $g(w,x) = x$.

Otherwise, consider the case where x is contained in some w-avoiding cycle. Suppose some node w' on the dominator chain following IDOM($g(w,x)$) to IDOM(x) is <u>not</u> contained in a w-avoiding cycle. Then the set $g(w',x)$ of $PV(w')$ is not merged into w' in $PV(w'-1)$, so $g(w',x) = g(w'-1,x) \neq w'$. Furthermore we can show that for $y = w', w'-1, \ldots, g(w,x)+1$; $g(w',x) \notin J(y)$ so $g(w',x) = g(y,x) \neq y$. Hence $g(w',x) = g(g(w,x),x) \neq g(w,x)$. Since $g(w,x)$ is the name of a set of $PV(w)$, $g(w,x)$ is not merged into any other set of $PV(g(w,x)), PV(g(w,x)-1), \ldots, PV(w)$, so $g(g(w,x),x) = g(w,x)$, and we have a contradiction.

Finally, suppose IDOM($g(w,x)$) is contained in some w-avoiding cycle p. Each such path p must contain a unique node w_p which dominates IDOM($g(w,x)$) and no node in p properly dominates w_p. Choose some such p with w_p as late as possible in the dominator ordering; i.e., as close as

possible to IDOM(g(w,x)). Then we can show that g(w,x) ϵ $J(w_p)-\{w_p\}$ and so g(w,x) is merged into w_p in $PV(w_p-1)$, which is impossible (since g(w,x) is the name of a set in PV(w)). □

Corollary 5.5.1 Let w,x ϵ V such that w dominates x in G. If x is contained in no w-avoiding cycles then $C1_G(w,x)$ = x. Otherwise, $C1_G(w,x)$ = IDOM(g(w,x)).

Proof. If x is contained in no w-avoiding cycles then, by definition, $C1_G(w,x)$ = x. Otherwise, suppose x is contained in some w-avoiding cycle. By Theorem 5.5.1, all nodes in the dominator chain following IDOM(g(w,x)) to x are contained in w-avoiding cycles, so $C1_G(w,x)$ properly dominates g(w,x). Hence, IDOM(g(w,x)) is the last node in the dominator chain from w to x which is contained in a w-avoiding cycle and we conclude that $C1_G(w,x)$ = IDOM(g(w,x)). □

We require the disjoint set operations:

(1) FIND(x) gives the name of the set currently containing node x.

(2) UNION(x,y): merge the set named x into the set named y.

The algorithm for computing $C1_G$ is given below.

Algorithm 5C

INPUT Flow graph G = (V, E, r) and ordered pairs $(w_1,x_1),\ldots,(w_\ell,x_\ell)$ such that each w_i dominates x_i.

OUTPUT $C1_G(w_1,x_1),\ldots,C1_G(w_\ell,x_\ell)$.

```
begin
  declare SET, BUCKET, FLAG := arrays length n = |V|;
  Compute the depth-first spanning tree ST of G;
  Number the nodes in V by preorder in ST;
  Computer the dominator tree DT;
  for x := 1 to n do
    begin
      SET(x) := {x};
      BUCKET(x) := the empty set {};
      FLAG(x) := FALSE;
    end;
  for i := 1 to ℓ do add xi to BUCKET(wi);
  for w := n by -1 to 1 do
    begin
      for all x ε BUCKET(w) do
        begin
          if FLAG(x) then
            C1G(w,x) := the father of FIND(x) in DT;
          else C1G(w,x) := x;
      if w > 1 then
        begin
          Compute I(w) by the Algorithm of [T2];
          if I(w) is not empty then
            begin
              for all y ε I(w) do
                begin
                  z := FIND(y);
                  if NOT FLAG(z) then
                    begin
                  D:  for all x ε IDOM-1(z) do
                        if FLAG(x) then
                          UNION(FIND(x),w);
                      FLAG(z) := TRUE;
                    end;
                  if z ≠ w do UNION(z,w);
                end;
            end;
        end;
    end;
end;
```

<u>Theorem 5.5.2</u> Algorithm 5C correctly computes $C1_G(w_1,x_1),\ldots,C1_G(w_\ell,x_\ell)$ in time almost linear in $a+\ell$.

<u>Proof</u> (Sketch). We may show by an inductive argument that on entering the main loop on the (n+1-w)'th iteration:

(1) FIND(x) gives g(w,x),

(2) FLAG(x) iff x is contained in a w-avoiding cycle,

and then apply Corollary 5.5.1 to show the correctness of Algorithm 5C.

ST, DT, and I(n), I(n-1),...,I(2) may be computed by the algorithms of [T1,T4,T2] in time almost linear in $a = |A|$. The other steps of Algorithm 5C clearly require a linear number of elementary and disjoint set operations. These set operations may be implemented in almost linear time by an algorithm analyzed by Tarjan[T3]. □

5.6 The Computation of C2

The first formulation of code motion was shown to reduce to a number of subproblems including the calculation of the function C2; recall that for flow graph $G = (V, E, r)$ and each $w,x \in V$ such that w dominates x, $C2_G(w,x)$ is the first node on the dominator chain from w to x which is not contained on any x-avoiding cycles. For such w,x let a path from x to w, which avoids all proper dominators of x other than w, and which is either a simple (acyclic) path or a simple cycle (a cycle containing no other cycles as proper subsequences), be called a *dominator disjoint* (DD) *path*. Let DT be the dominator tree of G and for each $x \in V-\{r\}$, let IDOM(x) be the father of node x.

Our algorithm for computing $C2_G$ will require a function DDP such that for each $x \in V$, DDP(x) = x if x = r or there is no DD path from IDOM(x), and otherwise DDP(x) is the first node y on the dominator chain from the root r to x such that there exists an x-avoiding DD path from IDOM(x) to y.

Lemma 5.6.1. If DDP(x) properly dominates x then all nodes on the dominator ordering from DDP(x) to IDOM(x) are contained on an x-avoiding cycle. Otherwise, DDP(x) = x and IDOM(x) is contained on no x-avoiding cycles.

Proof. If DDP(x) properly dominates x, then let p be a DD path from IDOM(x) to DDP(x). Since DDP(x) dominates

$IDOM(x)$, there is an x-avoiding path p' from $DDP(x)$ to $IDOM(x)$. Hence $p \cdot p'$ is the required x-avoiding cycle.

On the other hand, suppose $DDP(x) = x \neq r$ and $IDOM(x)$ is contained on an x-avoiding cycle q. Let q' be the subsequence of q from $IDOM(x)$ to some node z immediately dominating x, and containing no other proper dominators on x. Then q' is a DD path, so $DDP(x)$ properly dominates z, implying that $DDP(x) \neq x$, contradiction. □

<u>Lemma 5.6.2</u> Let $z \in V$ have at least two sons and be contained on a cycle avoiding some son of z in DT. Let x_1 (x_2) be a son of z with DDP value earliest (latest) in the dominator ordering. Then for each y which is properly dominated by z, $DDP(x_1)$ is a dominator of $DDP(y)$; furthermore, if $y \neq x_2$ and y is a son of z then $DDP(y) = DDP(x_1)$.

<u>Proof</u> Suppose z is a proper dominator of y, but $DDP(y)$ is a proper dominator of $DDP(x_1)$. Then $DDP(y) \neq y$ so there is a DD path p from $IDOM(y)$ to $DDP(y)$. Let x' be a son of z which is not a dominator of y. Let p' be a simple x'-avoiding path from z to y. Composing p' and p, we have an x'-avoiding DD path from z to $DDP(y)$. But this implies that $DDP(x')$ is a proper dominator of $DDP(x_1)$, contradicting the assumption that x_1 has DDP value earliest in the dominator ordering. Hence, $DDP(y)$ is dominated by $DDP(x_1)$.

Suppose $y \neq x_2$ and y is a son of z. Since z is

contained on a cycle avoiding some son of z, there must be a DD path $\bar{p}$ from z to DDP(x_1). If $\bar{p}$ avoids all sons of z in DT, then we have our result; DDP(y) = DDP(x_1). Otherwise, let $\bar{x}$ be the last node in $\bar{p}$ which is a son of z. Let $\bar{p}_1$ be the subsequence of $\bar{p}$ from $\bar{x}$ to z. For any x' ϵ V-{$\bar{x}$}, let p_2 be a x'-avoiding simple path from z to $\bar{x}$. Composing $\bar{p}_1$ and p_2, we have a x'-avoiding DD path from z to DDP(x_1). Hence, DDP(x') = DDP(x_1). If $\bar{x} = x_2$ then $y \neq \bar{x}$ so we have DDP(y) = DDP(x_1). On the other hand, if $\bar{x} \neq x_2$ then DDP(x_2) = DDP(x_1). Since DDP(y) dominates DDP(x_2), we again have DDP(y) = DDP(x_1). □

Let DT be the dominator tree of G with the edges oriented so that for each node z ϵ V contained on a cycle avoiding some node immediately dominated by z, the left-most son of z in DT has DDP value at least as late in the dominator ordering as the other sons of z (by Lemma 5.6.2, the remaining sons have the same DDP), and number V by a preordering of DT.

For each x ϵ V-{r}, let K(x) consist of (1) the set of nodes contained on the dominator chain from DDP(x) to IDOM(x) plus (2) the immediate dominator of DDP(x) if it is contained on a DDP(x)-avoiding cycle.

Let PV'(1),PV'(2),...,PV'(n) be a sequence of partitions of V such that:

(a) PV'(1) partitions V into unit sets, each set named for

the node which it contains.

(b) For $x = 2,\ldots,n$ let $PV'(x) = PV'(x-1)$ if $DDP(x) = x$. Otherwise, let $PV'(x)$ be derived from $PV'(x-1)$ by collapsing each set containing an element of $K(x)-\{IDOM(x)\}$ into the set containing $IDOM(x)$ in $PV'(x-1)$ and then renaming this set to $IDOM(x)$.

For $w,x \in V$ such that w dominates x, let $h(w,x)$ be the name of the set containing w in $PV'(x)$.

<u>Theorem 5.6.1</u> If w is contained in no x-avoiding cycles, then $h(w,x) = w$ and otherwise $h(w,x)$ is the last node on the dominator chain from w to x such that all nodes occurring up to and including $h(w,x)$ on this chain are contained on x-avoiding cycles.

<u>Proof</u> Let $(w=y_1,\ldots,y_k=x)$ be the dominator chain from w to x.

Suppose w is <u>not</u> contained on an x-avoiding cycle. Consider some node y_i on this dominator chain following w. If $DDP(y_i)$ dominates w then by Lemma 5.6.1, w is contained in an x-avoiding cycle, a contradiction. Thus $w \notin K(y_i)-\{y_{i-1}\}$ and w is <u>not</u> collapsed into y_{i-1}, so $w = h(w,y_1) = \ldots = h(w,y_k) = h(w,x)$.

Otherwise, suppose w is contained on some x-avoiding cycle. Assume there is a node y_i, on the dominator chain following w to $h(w,x)$, which is <u>not</u> contained on an x-avoiding cycle. By Lemma 5.6.1, $DDP(y_i) = y_i$. Then

$h(w,y_i)$ properly dominates y_i, so there is some $y_{j-1} = h(w,y_j)$ on the dominator chain from y_i to w such that $DDP(y_j)$ dominates $h(w,y_i)$. By Lemma 5.6.1, y_i is contained on an x-avoiding cycle, a contradiction.

Finally, assume $h(w,x) \neq w$ and let y_i be the first node following $h(w,x)$ on the dominator chain from w to x. Suppose y_i is contained on an x-avoiding cycle. Then by Lemma 5.6.1, $DDP(y_i)$ properly dominates y_i. Since $h(w,x) \neq w$, $h(w,x)$ is contained on an x-avoiding cycle, so $h(w,x) \in K(y_{i+1})-\{y_i\}$ and hence $h(w,x)$ is merged into y_i, contradicting our assumption that $h(w,x)$ is the name of a set in $PV'(x)$. □

<u>Corollary 5.6.1</u> For $w,x \in V$ such that w dominates x, if w is contained on no x-avoiding cycles then $C2_G(w,x) = w$ and otherwise, $C2_G(w,x)$ is the unique node dominating x and immediately dominated by $h(w,x)$.

<u>Proof</u> follows directly from Theorem 5.6.1.

Our algorithm for computing C2 will require the usual disjoint set operations UNION and FIND plus the operation RENAME(x,y) which renames the set x to y.

Algorithm 5D

INPUT Flow graph $G = (V, E, r)$, DDP, and ordered pairs $(w_1,x_1),\ldots,(w_\ell,x_\ell)$ such that each w_i dominates x_i.

OUTPUT $C2_G(w_1,x_1),\ldots,C2_G(w_\ell,x_\ell)$.

```
begin
  declare SET, FLAG, BUCKET := arrays length n = |V|;
  Compute the dominator tree DT of G;
  for all z ε V such that z has a son x in DT with
  DDP(x) dominating z do
    begin
      let x' be the son of z which has DDP(x')
      latest in the dominator ordering;
      install x' as the left-most son of z;
    end;
  Number the nodes of V by the preordering of the
  resulting oriented tree;
  for x := 1 to n do
    begin
      SET(x) := {x};
      FLAG(x) := FALSE;
      BUCKET(x) := the empty set {};
    end;
  for i := 1 to ℓ do add w_i to BUCKET(x_i);
  for x := 1 to n do
    begin
      if x > 1 and DDP(x) ≠ x then
        begin
          z := the father of x in DT;
          FLAG(z) := TRUE;
          NEXT(z) := x;
          RENAME(FIND(z),z);
          y := the father of DDP(x) in DT;
      D:  if FLAG(y) and y ≠ z do
            UNION(y,z);
          u := FIND(DDP(x));
          till u = z do
            begin
              FLAG(u) := TRUE;
              UNION(u,z);
              u := FIND(NEXT(u));
            end;
        end;
      comment Apply Corollary 5.6.1;
      for all w ε BUCKET(x) do
        if FLAG(w) then C2_G(w,x) := NEXT(FIND(w))
        else C2_G(w,x) := w;
    end;
end;
```

Theorem 5.6.2 Algorithm 5D correctly computes $C2_G(w_1,x_1),\ldots,C2_G(w_\ell,x_\ell)$ in time almost linear in $a+\ell$.

Proof (Sketch). It is possible to establish that for all $w \in V$ after the x'th iteration of the main loop:

(1) NEXT(IDOM(w)) = w for $w \neq r$ and w properly dominates x.

(2) The sets are just as in PV'(x), with h(w,x) the name of the set containing w.

(3) FLAG(w) = TRUE iff w is not contained in a x-avoiding cycle.

Then the correctness follows from Corollary 5.6.1.

We compute DT by the algorithm of [T4] in time almost linear in $a+\ell$. The other steps of Algorithm 5D may easily be shown to require a linear number of elementary and disjoint set operations. Hence, by the results of [T3], the total cost in elementary operations is almost linear in $a+\ell$. □

5.7 Computing DDP on Reducible Flow Graphs

This section is concerned with the function DDP required by Algorithm 5D to compute C2. Unfortunately, we know of no algorithm which computes DDP efficiently for G nonreducible. We assume henceforth that G is reducible, so by the results of Hecht and Ullman [HU1], all cycle edges of G are A-cycle edges (they lead from nodes to their proper dominators). Let ST' be the spanning tree derived from the depth first search spanning tree ST of G by reversing the edge list. The nodes of G are numbered by a preordering of ST'.

Lemma 5.7.1 If $x > y$ and both x and y are unrelated in DT, then any path p from x to y contains a dominator of x.

Proof It is sufficient to assume that p is simple (acyclic). Let (u,v) be the first edge through which p passes such that $v \leq y < u$. Observe that the only edges of G in decreasing preorder are A-cycle edges, so (u,v) is an A-cycle edge and v dominates u. We claim also that v dominates x. Suppose not, so there is a v-avoiding path p' from the root r to x. Composing p' with the subsequence of p from x to u, we have a v-avoiding path from r to u, which contradicts the fact that v dominates u. Hence, v dominates x. □

We now show that in the reducible flow graph G, DD paths have a very special structure. Let p = $(x=y_0,\ldots,y_k=w)$ be a DD path from x to w passing through

edges $e_1,\ldots,e_k$, where $e_i = (y_{i-1},y_i)$.

Theorem 5.7.1 e_k is an A-cycle edge and $e_1,\ldots,e_{k-1}$ are not.

Proof. Since p can not contain any dominators of x other than w, y_{k-1} and x are unrelated in DT. Assume $e_k = (y_{k-1},w)$ is not an A-cycle. Hence, $x > w > y_{k-1}$ and applying Lemma 5.7.1, $(x=y_0,\ldots,y_{k-1})$ must contain a node z which is a proper dominator of x, contradicting our assumption that p is DD.

Consider any $e_i = (y_{i-1},y_i)$ for $1 \leq i < k$. Since p is DD, y_i does not dominate x. Thus, there is a y_i-avoiding path p_1 from the root r to x. Also, let p_2 be the subsequence of p from x to y_{i-1}. Composing p_1 and p_2, we have a y_i-avoiding path from the root r to y_{i-1}, which implies that y_{i-1} is not dominated by y_i. Hence, none of $e_1,\ldots,e_{k-1}$ are A-cycles. □

Theorem 5.7.2 Let p be a DD path from x to w, where w properly dominates x and let z be a immediate predecessor of x in G such that z,x are unrelated in DT. Then $p' = (z,x) \cdot p$ is a DD path avoiding all sons of z in DT.

Proof To show that p' is DD we need only demonstrate that w properly dominates z and p avoids z. Let $p = (x=y_0,\ldots,y_k=w)$. Since z,x are unrelated in DT and w properly dominates x, w is distinct from z.

We claim that w properly dominates z in G. Suppose not, then there must be a w-avoiding path p_1 from the root r

to z. But $p_1 \cdot (z,x)$ is a w-avoiding path from the root r to x, contradicting our assumption that w properly dominates x. Hence, w properly dominates z.

Suppose p contains z, so $z = y_i$ for some $1 < i < k$. Then $(z, x=y_1, \ldots, y_i=z)$ is a cycle in G and must contain an A-cycle edge. Since z,x are unrelated in DT, this implies that for some j, $1 \leq j \leq i$, (y_{j-1}, y_j) is an A-cycle edge, contradicting Theorem 5.7.1. We conclude that p avoids z.

Hence, $p' = (z,x) \cdot p$ is DD.

Now suppose p contains a node y dominated by z. Since x,z are unrelated in DT, there must be a z-avoiding path p_2 from the root r to x. Composing p_2 and the portion of p from x to y, we have a z-avoiding path from r to y, which is impossible. Hence, $p' = (z,x) \cdot p$ avoids all sons of z in DT. □

Let p be a DD path from x to w. Let the first edge (u,v) through which p passes, such that u is dominated by x but v is not properly dominated by x, be called the <u>first jump edge</u> of p.

<u>Theorem 5.7.3</u> Let x' be a proper dominator of x. If either (1) v = w dominates x' or (2) v ≠ w and IDOM(v) properly dominates x', then there exists a DD path from x' to w with first jump edge e = (u,v).

<u>Proof</u> Let p_1 be a simple path from x' to x. Suppose p_1

contains some node z not dominated by x'. Then the subsequence of p_1 from z to x must contain x'. But this implies that x' occurs twice in p_1, which is impossible. Hence, all nodes in p_1 are dominated by x' and $p_2 = p_1 \cdot p$ is a DD path. Since x' properly dominates x which dominates u, x' also dominates u. If either (1) or (2) hold, then v does not properly dominate x'. Thus, the first jump edge of p_2 is e = (u,v). □

Algorithm 5E

INPUT A reducible flow graph G = (V, E, r).
OUTPUT DDP.

```
begin
  declare SET,FLAG,DDP,SONS := arrays length n = |V|;
  procedure EXPLORE(x,w,e):
    begin
      comment there is a DD path from x to w
          and e is the first jump edge of p;
      Let e = (u,v);
      for each y ε SONS(x) such that y,u are
        unrelated in DT do
        begin
          delete y from SONS(x);
          DDP(y) := w;
        end;
        if x ≠ r and not FLAG(x) then
          begin
            FLAG(x) := TRUE;
            x' := IDOM(x);
            if FLAG(x') then
              UNION(x,FIND(x'));
            if NOT x = w then
              begin
                comment Apply Theorem 5.7.3;
                if (v=w dominates x') OR (v≠w and
                IDOM(v) properly dominates x') then
                  L1: EXPLORE(x',w,e);
                comment Apply Theorem 5.7.2;
                for all immediate predecessors z
                of x in G such that x,z are unrelated
                in DT do
                  L2: EXPLORE(z,w,(z,x));
              end;
          end;
    end;
  Compute DT, the dominator tree of G;
  Compute ST, the depth-first spanning tree of G;
  Let ST' be derived from ST by reversing the edge list;
  Number the nodes of V by preorder of ST';
  for all x := 1 to n do
    begin
      SET(x) := {x};
      FLAG(x) := FALSE;
      DDP(x) := x;
      SONS(x) := the sons of x in DT;
    end;
  for w := 1 to n do
    for all A-cycle edges (x,w) entering w do
      L3: EXPLORE(x,w,(x,w));
end;
```

Lemma 5.7.2 On each execution of EXPLORE(x,w,e), w dominates x and there is a DD path from x to w with first jump edge e.
Proof by structural induction. On each initial call to EXPLORE(x,w,e) at label L3, e is a A-cycle edge (x,w) which is clearly a DD path. Suppose on any other call to EXPLORE(x,w,e) there is a DD path from x to w with first jump edge e. By Theorems 5.7.3 and 5.7.2, the recursive calls to EXPLORE at L1 and L2, respectively, also satisfy this lemma. □

It is also easy to prove by structural induction that:
Lemma 5.7.3 On each execution of EXPLORE(x,w,e), let y be a dominator of x contained in the set named FIND(y). If y has not previously been visited then FLAG(y) = FALSE and FIND(y) = y; otherwise, FLAG(y) = TRUE and FIND(y) is the earliest node y' on the domination chain from the root r to y such that all nodes from y' to y on this chain have been previously visited.

Let p be a DD path from x to w with first jump edge e = (u,v). For k > 1, the kth jump edge of p is recursively defined to be the (k-1)th jump edge (if this is defined and is not the last edge through which p passes) of the subsequence of p from v to w.
Lemma 5.7.4 For each w,y ε V such that w properly dominates y, if there exists a y-avoiding DD path p from IDOM(y) to w, then EXPLORE(IDOM(y),w,e) is eventually called, where e =

(u,v) is the first jump edge of some such p.

Proof by induction on w. Suppose the lemma holds for all w' < w. Since e = (u,v) is the first jump edge of p, IDOM(y) dominates u. If v = w, then (u,v) is an A-cycle edge so EXPLORE(u,w,(u,w)) is executed at label L3, and by a sequence of recursive calls to EXPLORE at label L1, we finally have a call to EXPLORE(IDOM(y),w,(u,v)). Otherwise, suppose the lemma holds for all p leading to w such that p has less than k jump edges. If p has k jump edges, then by the second induction hypothesis, EXPLORE(u,w,(u,v)) is called at label L2. Again, by a sequence of recursive calls to EXPLORE at label L1, we eventually have a call to EXPLORE(IDOM(y),w,(u,v)). □

Theorem 5.7.4 Algorithm 5E correctly computes DDP for G reducible, in time almost linear in a = |A|.

Proof The correctness of Algorithm 5E follows from Lemmas 5.7.2, 5.7.3, and 5.7.4. ST and DT may be computed (if they have not been computed previously) by the metods of [T1,T4] in almost linear time. For each x ε V, the total cost of all visits to x by EXPLORE is $|IDOM^{-1}[x]|$ + |indegree(x)| in elementary and disjoint set operations. Hence, if we use a good implementation of disjoint set operations (analyzed by Tarjan[T3]), the total cost of Algorithm 5E is almost linear in a. □

5.8 Niche Flow Graphs

Here we introduce a special class of flow graphs called niche flow graphs which in certain cases simplify the algorithms given in Sections 5.5 and 5.6 for computing C1 and C2. As we shall demonstrate, the transformation of an arbitrary flow graph to a niche flow graph can be done in almost linear time; furthermore, both versions of code motion are improved by this transformation. [E,AU2] describe a similar process, where special nodes are added to the flow graph just above intervals.

Let $G = (V, E, r)$ be an arbitrary flow graph. For any $w \in V-\{r\}$ with immediate dominator IDOM(w) in G, if IDOM(w) is contained on no w-avoiding cycles then IDOM(w) is called the *niche node of w*. Intuitively, the niche nodes lie just above cycles (relative to the dominator ordering of G) and hence are good nodes to move code into. G is a *niche flow graph* if each node $w \in V-\{r\}$, with an entering A-cycle edge but *no* entering B-cycle edge, has a niche node.

If G is not a niche flow graph, then a niche flow graph G' may be derived from G by testing for each $w \in V-\{r\}$ whether w has an entering A-cycle edge and no entering B-cycle edges. If so, then add a distinct, new node $\hat{w}$ which is to be the niche of w in G', an edge from $\hat{w}$ to w, and replace each non-cycle edge (x,w) entering w with a new edge $(x,\hat{w})$. The resulting flow graph G' has no more than $n = |V|$

additional nodes and edges. Since no B-cycle edges are added to G', by Theorem 5.2, G' is reducible if G was.

Lemma 5.8.1 If G is reducible and $y \in V-\{r\}$ is contained of an IDOM(y)-avoiding cycle q, then y has an entering A-cycle edge.

Proof Let x be the immediate predecessor of y in q. Since G is reducible, q contains a unique node z dominating all other nodes in q. But no proper dominator of y is contained in q, so z = y. Hence, y dominates x and (x,y) is an A-cycle edge. □

Let the nodes of G be numbered as in Section 5.5 by a preordering of a depth first search spanning tree of G.

Theorem 5.8.1 If G is a reducible niche flow graph, then for $w = n, n-1, \ldots, 2$ the partition PV(w-1) is derived from PV(w) by collapsing sets $I(w)-\{w\}$ into w.

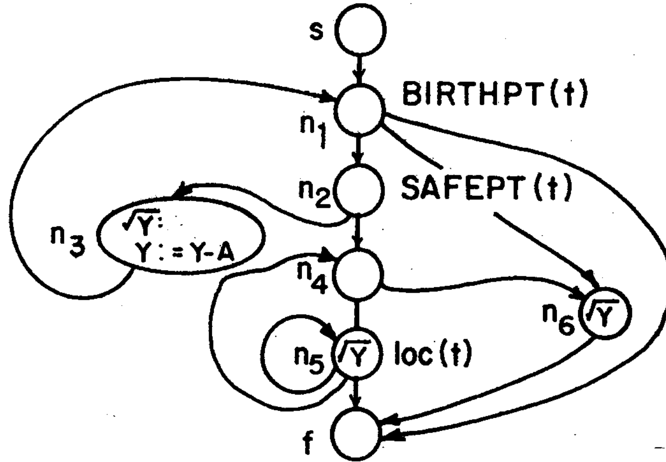
Proof Recall that PV(w-1) is defined to be derived from PV(w) by collapsing into w each set z containing at least one element $y \in J(w)-\{w\}$. Suppose there is a set $z \not\in I(w)$ in PV(w) containing some $y \in (J(w)-I(w))-\{w\}$. Then, by definition of J(w), y is contained on a w-avoiding cycle q and $IDOM(y) \in I(w)$. But since $z \not\in I(w)$, q avoids IDOM(y) and IDOM(y) is contained in a y-avoiding cycle q'. By Lemma 5.8.1, y has an entering A-cycle edge. Since G is a niche flow graph, IDOM(y) is the niche of y. But this is impossible since IDOM(y) is contained on a y-avoiding cycle q'. □

The above theorem allows us to simplify Algorithm 5D, which was used to compute $C1_G$, in the case G is a reducible niche flow graph. In particular, the statement labeled D may be deleted from Algorithm 5D. Similarly, in this case the statement labeled D may be deleted from Algorithm 5E.

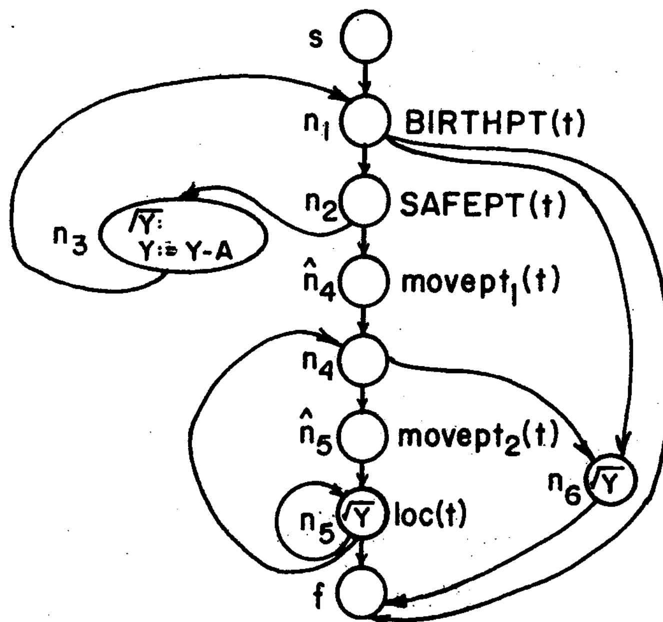
Theorem 5.8.2 If G is a reducible niche node and DDP(x) ≠ x, then K(x) = those nodes of the dominator chain from DDP(x) to IDOM(x).

Proof Suppose there exists some x ε V such that DDP(x) properly dominates x and IDOM(DDP(x)) is contained on a DDP(x)-avoiding cycle. Let p be the DDP path from x to DDP(x) and let p' be a simple path from DDP(x) to x. Composing p and p', we have a IDOM(DDP(x))-avoiding cycle containing DDP(x). Hence by Lemma 5.8.1, DDP(x) has an entering A-cycle edge. Since G is a niche flow graph, IDOM(DDP(x)) is the niche node of x. But by hypothesis, this niche node of DDP(x) is contained on a DDP(x)-avoiding cycle, which is impossible. □

Original Control Flow Graph



Niche Flow Graph



t is the text expression $\sqrt{Y}$ located at n_5

Figure 5.2. Transformation of a flow graph F into a niche flow graph F'.

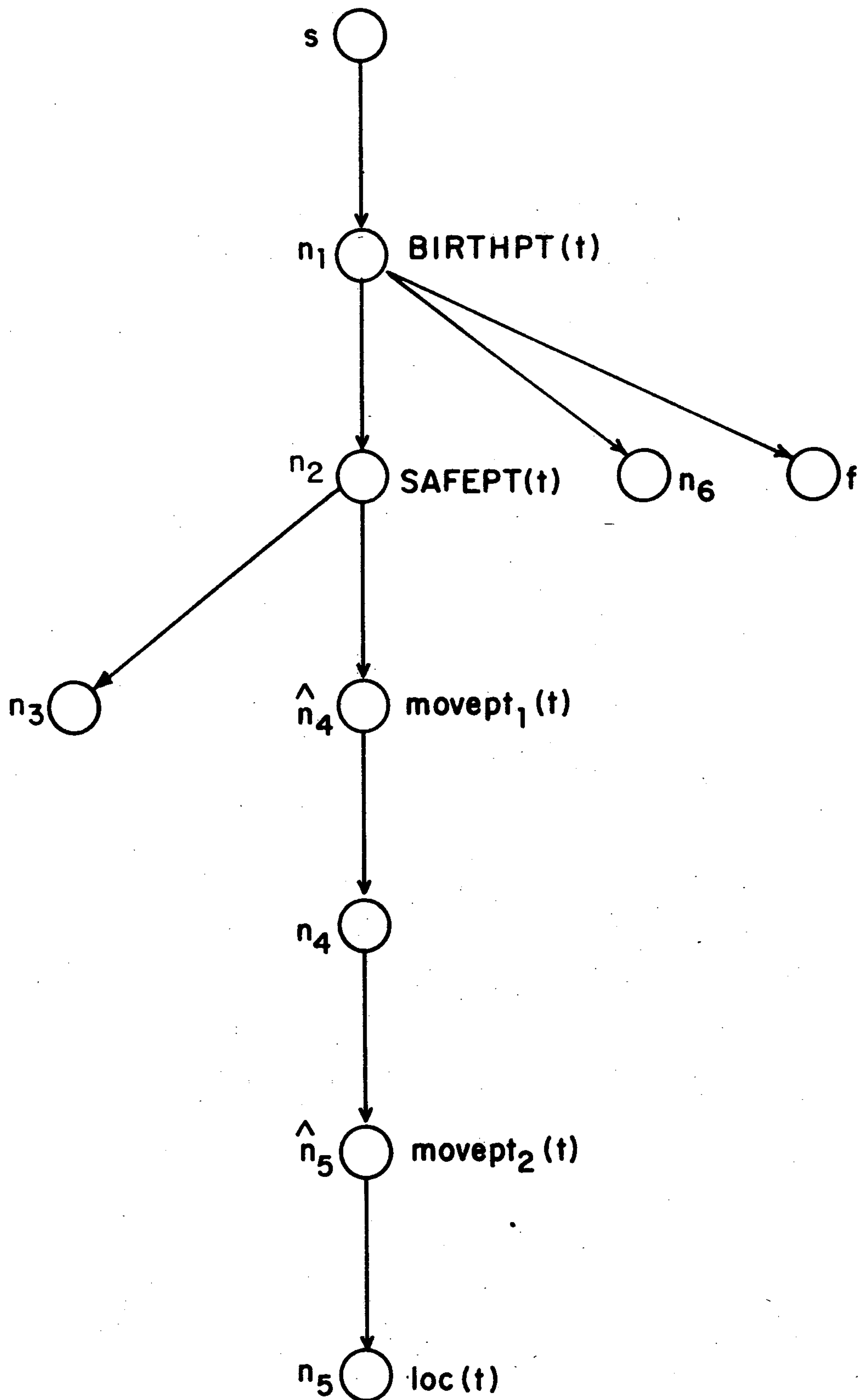


Figure 5.3. The dominator tree of the control flow graph F'.

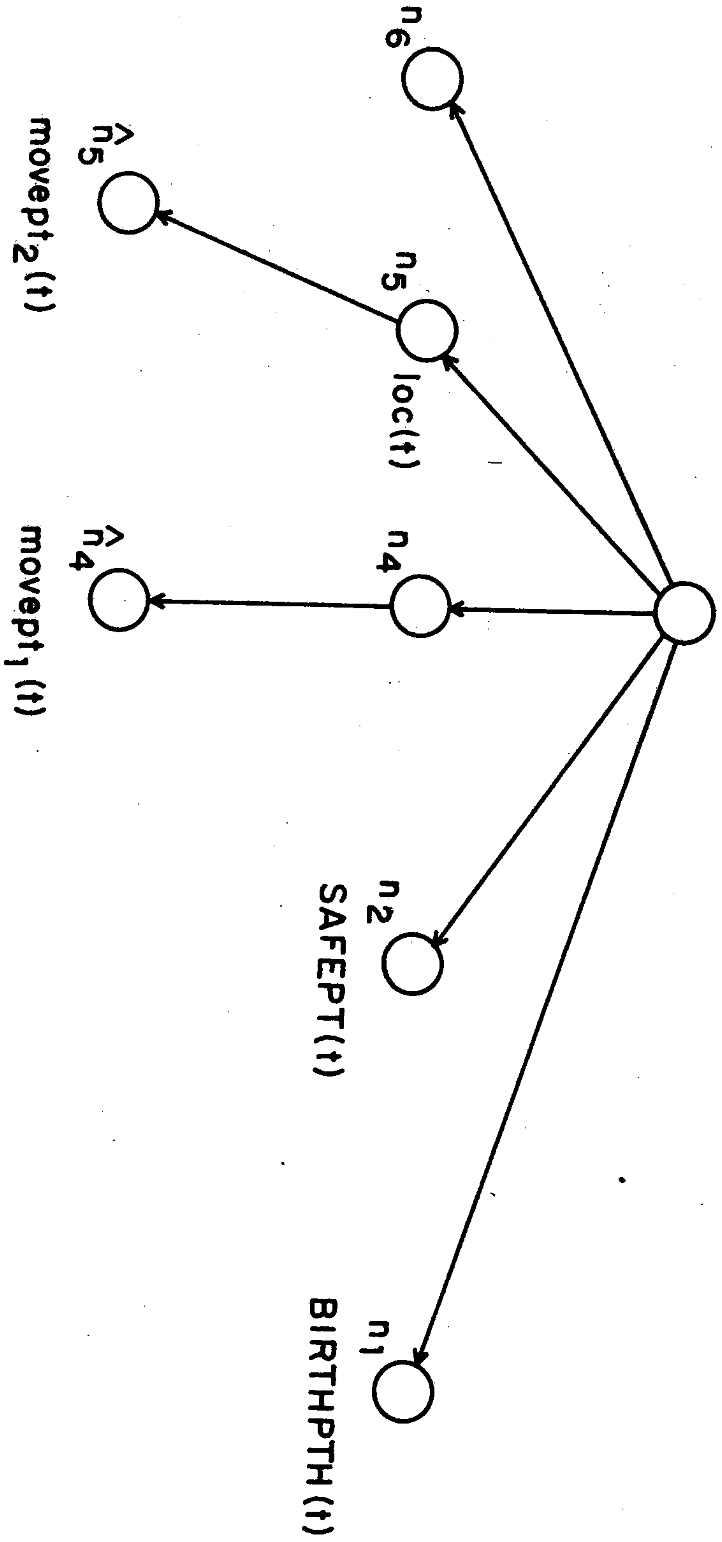


Figure 5.4. The dominator tree of the reverse of the control flow graph F'.

REFERENCES

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

[AU1] Aho, A.V. and Ullman, J.D., _The theory of parsing, translation and compiling_, Vol. II, Prentice-Hall, Englewood Cliffs, N.J., (1973).

[AU2] Aho, A.V. and Ullman, J.D., _Introduction to Compiler Design_, to appear.

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

[CA] Cocke, J. and Allen, F. E., "A catalogue of optimization transformations," _Design and Optimization of Computers_, (R. Rustin, ed.), Printice-Hall, (1971), pp 1-30.

[E] Earnest, C., "Some topics in code optimization," _JACM_, Vol. 21, Num. 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 Second ACM Symp. on Principles of Programming Languages_, (Jan. 1975), pp. 1-9.

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

[G] Geschke, C.M., "Global program optimizations," Carnegie-Mellon University, 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." _J. ACM_, Vol. 23, No. 1, (Jan. 1976), pp. 172-202.

[HU1] Hecht, M.S. and Ullman, J.D., "Flow graph reducibility," _SIAM J. Computing_, Vol. 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_, Vol. 4, Num. 4, (Dec. 1975), pp. 519-532.

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

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

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

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

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

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

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

[Kn1] Knuth, D. E., The art of computer programming, Vol 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., (1968).

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

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

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

[S] Schaefer, M., A mathematical theory of global flow analysis, Prentice-Hall, Englewood Cliffs, N.J., (1973).

[Sc1] Schwartz, J.T., "Automatic data structure choice in a language of very high level," CACM, Vol. 18, Num. 12, (Dec. 1975), pp. 722-728.

[Sc2] Schwartz, J.T., "Optimization of very high level languages -- value transmission and its corollaries," Computer Languages, V. 1, Num 2, (1975), pp. 161-194.

[Sc3] Schwartz, J.T., "Optimization of very high level languages -II. Deducing relationships of inclustion and membership," Computer Languages, V. 1, Num 3, (Sept. 1975), pp. 161-194.

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

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

[T2] Tarjan, R., "Testing flow graph reducibility," J. Comp. and Sys. Sciences, Vol. 9, (1974), pp 355-365.

[T3] Tarjan, R., "Efficiency of a good but not linear set union algorithm," J. ACM, Vol. 22, (April 1975), pp 215-225.

[T4] Tarjan, R., "Applications of path compression on balanced trees," Stanford Computer Science Dept., Technical report 512, (Aug 1975).

[T5] Tarjan, R., "Solving path problems on directed graphs," Stanford Computer Science Dept., Technical report 528, (Oct 1975).

[T6] Tarjan, R., Personal communication to M. Karr, (1976).

[Te] Tennenbaum, A., "Compile time determination for very high level languages," Ph.d. Thesis, Courant Computer Science Report No. 3, Courant Institute of Mathematical Sciences, New York University, New York, N.Y., 1974.

[U] Ullman, J.D., "Fast algorithms for elimination of common subexpressions," Acta Informatica, Vol. 2, N. 3, (Jan. 1974), pp 191-213.

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

[R] Reif, J.H., "Combinatorial aspects of symbolic program analysis," Harvard University, Ph.d. Thesis, Dept of Engineering and Appied Physics, (1977).

[RL] Reif, J. H. and Lewis, H. R., "Symbolic evaluation and the global value graph," _Proceedings of the Fourth ACM Symposium on Principles of Programming Languages_, Los Angeles, California, (January 1977).