COMBINATORIAL ASPECTS OF SYMBOLIC PROGRAM ANALYSIS

A thesis presented

by

John Henry Reif

to

The Division of Engineering and Applied Physics

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Applied Mathematics

Harvard University

Cambridge, Massachusetts

(July, 1977)

PREFACE

This research was supported by Naval Electronics System Command Contract No. N00039-76-C-0168 and Rome Air Development Center Contract No. F30602-76-C-0032.

Many people have contributed to the successful completion of this dissertation.

I am deeply indebted to my advisor Professor Harry Lewis for inspiration and guidance in directing the ideas of this thesis to the printed page. I feel honored to be the first doctoral student of this philosoher, mathematician, and computer scientist; I am sure that many future students will benefit also from his wisdom and counsel.

I wish to express my gratitude to my other advisor, Professor Thomas Cheatham, for teaching me much of what I know about programming languages, for challenging me with problems in this field (some of which are solved herein and some which remain to be solved), and for a reading of this thesis.

I wish to thank Professor Christos Papadimitriou for introducing me to fields unrelated to this thesis but exciting nevertheless, for advice which was always excellent (but not always taken), and for a reading of this thesis.

I would like to thank Mark Davis for many spirited discussions on program optimization, and numerous critical but always helpful suggestions.

I would like to thank Glenn Bresnahan for serving as a cheerful office-mate and friend, and for a very thorough reading of this thesis.

Also, I wish to thank Phil Pura for his meticulous corrections to the grammar and spelling in preliminary versions of the manuscript.

I wish to dedicate this thesis to Jane Anderson for her patience and understanding throughout this work.

TABLE OF CONTENTS

PREFACE		ii
FIGURES		vi
SYNOPSIS		viii
Chapter		
1.	INTRODUCTION	
1.1	Overview	1-1
1.2	Graph Theoretic Notions	1-10
1.3	The Global Flow Model	1-14
1.4	Unsolvability of Various Code Improvements Within the Arithmetic Domain	1-20
1.5	Adding Procedure calls to the Global Flow Model	1-25
2.	SYMBOLIC EVALUATION AND THE GLOBAL VALUE GRAPH	
2.0	Summary	2-1
2.1	Introduction	2-2
2.2	Dags and Global Value Graphs	2-8
2.3	Propagation of Constants	2-20
2.4	A Partial Characterization of ψ, the Minimal Element of $^{\Gamma}$GVG	2-27
2.5	Rank Decomposition of a Reduced GVG	2-31
2.6	P-graph Completion and Decomposition	2-38
2.7	The Algorithm for Symbolic Evaluation	2-44
2.8	Improving the Efficiency of our Algorithm for Symbolic Evaluation	2-49
2.9	Further Applications of Global Value Graphs Live-Dead Analysis	2-53
3.	SYMBOLIC ANALYSIS OF PROGRAMS WITH STRUCTURED DATA	
3.0	Summary	3-1
3.1	Introduction	3-3
3.2	Propagation of Selections	3-11
3.3	Constant Propagation and Covers of Programs with Structured Data	3-18
3.4	Computing ψ^*, the Minimal Fixed Point of Ψ'	3-34
3.5	Type Declarations and Type Covers	3-45
4.	SYMBOLIC PROGRAM ANALYSIS IN ALMOST LINEAR TIME	
4.0	Summary	4-1
4.1	Introduction	4-2
4.2	The Computation of IDEF	4-6
4.3	The Weak Environment	4-20
4.4	Conclusion Computing Approximate Birthpoints and the Simple Cover	4-25

5. CODE MOTION

5.0	Summary	5-1
5.1	Introduction	5-4
5.2	Graph Theoretic Notions	5-9
5.3	Approximate Safe Points of Code Motion	5-13
5.4	Reduction of Code Motion to Cycle Problems	5-18
5.5	The Computation of C1	5-22
5.6	The Computation of C2	5-29
5.7	Computing DDP on Reducible Flow Graphs	5-36
5.8	Niche Flow Graphs	5-43

REFERENCES R-1

FIGURES

1.1 A control flow graph. 1-2

1.2 (From Kam and Ullman[KU2]) A text expression that is covered by a constant sign but not discovered by Kildall's algorithm. 1-7

1.3 A flow graph and its dominator tree. 1-12

1.4 The control flow graph F_Q. 1-21

2.1 An example of a fixed point of Ψ. 2-7

2.2 A dag representation of an expression. 2-9

2.3 The global value graph GVG^*. 2-17

2.4 A simple example of constant propagation through the global value graph. 2-25

2.5 Case (b) of Theorem 2.4. 2-28

2.6 Rank decomposition of a global value graph. 2-32

3.1 Reversal of a list in LISP. 3-10

3.2 An example of selection pairs. 3-12

3.3 The control flow graph F_R. 3-14

3.4 An example of a covering expression. 3-32

3.5 The global value graph GVG^* for the program of Figure 3.4. 3-43

3.6 The selection t is replaced by the selection variable SV_t. 3-44

3.7 A program P in LISP. 3-49

3.8 The type program P_τ derived from the program P of Figure 3.7. 3-50

3.9 A program containing an input variable which has no tight type definition. 3-54

4.1 An example of a simple cover. 4-5

4.2 Cases (1) and (2) of the definition of H'(m,w,S). 4-12

4.3 The dags of the program in Figure 4.1. 4-27

4.4 Dag representation of the simple cover. 4-28

5.1 A simple example of code motion. 5-3

5.2 Transformation of a flow graph F into a niche flow graph F'. 5-46

5.3 The dominator tree of the control flow graph F'. 5-47

5.4 The dominator tree of the reverse of the control flow graph F'. 5-48

SYNOPSIS

Much current research in computer science is devoted to the automatic _analysis and improvement of programs_. The central theme explored in this dissertation is _symbolic evaluation_: the determination of general, symbolic representations for values of text within programs, holding over all executions. These representations are called _covers_ and are terms (i.e. expressions containing no predicates) in a first order logical language.

We are interested in _efficient_ techniques for symbolic evaluation since applications (such as the optimization of source code before compilation) require results swiftly and inexpensively.

Our approach is _combinatorial in nature_; this reflects our view that the discovery of the combinatorial structure of symbolic evaluation is crucial to the development of efficient methods for carrying it out.

We assume a _global flow model_ of a program P wherein the flow of control through P is represented by a directed graph with _nodes_ corresponding the blocks of linear blocks of code and the _edges_ indicate possible flow of control between the blocks.

In Chapter 1, we define the relevant graph terminology, review the global flow model, and formally define the notion

of a cover. Further, we give a construction demonstrating that the problem of computing a minimal (best possible) cover in the domain of integers is recursively unsolvable. This implies that various global flow problems are also unsolvable in the arithmetic domain, including constant propagation, discovery of redundant computations, and loop invariants. Previous results by Kam and Ullman[KU2] showed certain global flow problems in abstract (nonarithmetic) domains to be unsolvable.

Kildall's iterative method[Ki] for symbolic evaluation may be used to compute a class of good, approximately minimal covers. In Chapter 2, we show that the minimal cover of this class is unique. Also, we present a direct (noniterative) algorithm for computing this cover with considerably less time and space complexity than the method of Kildall. This direct method is based on the use of a special class of graphs, called global value graphs, similar to those used by Schwartz[Sc2] to represent the flow of values (rather than control) through the program. Certain key lemmas and theorems in this chapter characterize the cover which we wish to construct in terms of a global value graph and reduce the symbolic evaluation to computing dominator trees (trees used to represent the path structure of digraphs) for which there is a very efficient algorithm due to Tarjan[T4].

Chapter 3 extends our techniques for symbolic analysis to a class of programs (such as those written in LISP 1.0) which have operations for the construction of structured objects (such as _cons_), and selection of subcomponents (such as _car_ and _cdr_), but no "destructive" operations (such as _replaca_ or _replacd_ in LISP 1.5). A key problem here is the "propagation of selections" which is the determination of all objects which a selection operation may reference. The propagation of selections was previously used by Schwartz[Sc2] for a different purpose, but he gave no explicit algorithm for carrying it out. We show that this problem is at least as hard as transitive closure, but give a relatively efficient bit vector algorithm for its solution. We define a class of covers similar to those of Chapter 2, but which take into account reductions due to selections of subcomponents. The computation of covers of this sort is reduced to the techniques of Chapter 2. We also introduce the concept of a type cover: an expression for the type (rather than value) of a text expression and holding over all executions of the program. We show that a type cover of a program P is equivalent to a (value) cover of program derived from P by substituting types for atoms and with an appropriate interpretation containing a universe of types, rather than structured values.

Chapter 4 presents an algorithm for symbolic evaluation which is very fast (requiring an almost linear number of bit vector operations for all flow graphs), but gives in general less powerful results than the method of Chapter 2. The results of this chapter may be used to speed up the method of Chapter 2.

Finally, in Chapter 5 we discuss in detail a particular code optimization, called code motion, which requires the covers we have computed in the preceding chapters. Code motion is the process of moving computations as far as possible out of cycles, to locations in the program where they are executed less frequently. Covers help us determine how far we may move computations before they are no longer defined. We present two formulations of code motion and also give algorithms for carrying them out in almost linear time (our algorithm for the first version is restricted to reducible flow graphs, but the other runs efficiently on all flow graphs).

CHAPTER 1

INTRODUCTION

1.1 Overview.

We rely on a global flow model of a computer program. The only statements in the programming language retained in the model are assignment statements whose left-hand sides are variables and whose right-hand sides are expressions built up from fixed sets of variables, function signs, and constant signs. All intraprogram control flow is reduced to a directed graph called a control flow graph indicating which blocks of assignment statements may be reached from which others, but giving no information about the conditions under which such branches might occur. Executions of the program correspond to paths through the control flow graph beginning at a distinguished start block, although not every such path in this graph need correspond to a possible execution of the program. Section 1.3 describes this global flow model in detail and in Section 1.5 we extend the model to allow for subroutining.

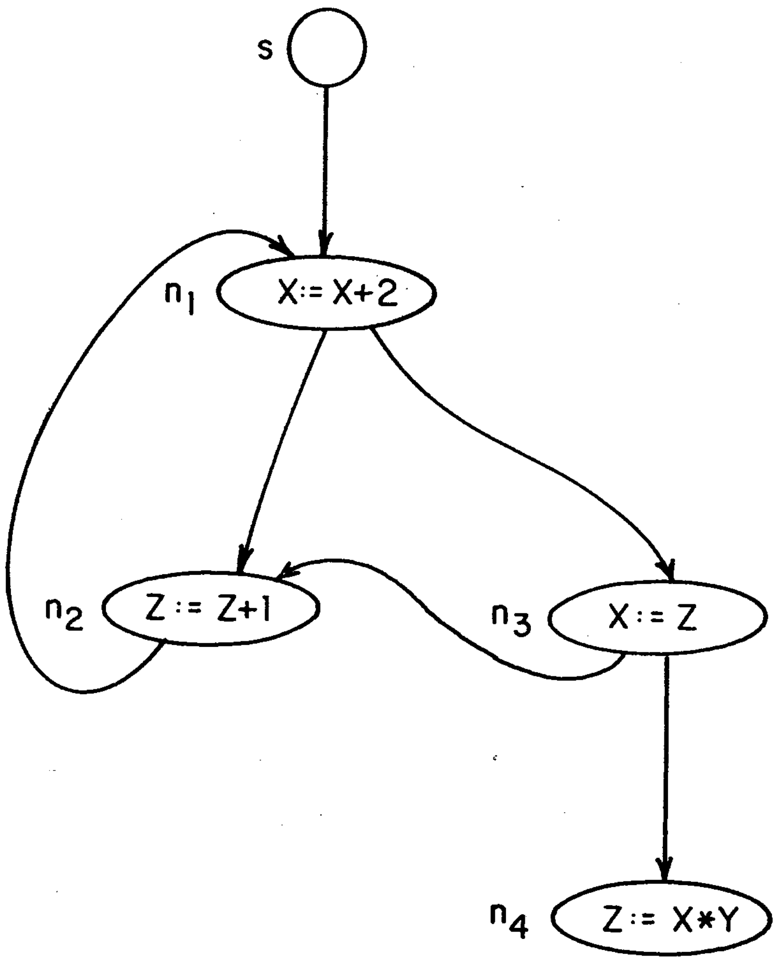


Figure 1.1 A control flow graph.

The utility of the global flow model is that many program analysis and improvement problems may be formulated as combinatorial problems on digraphs. A central program analysis problem of interest is symbolic evaluation: the discovery, for each expression t in the text of the program, of a closed form expression α for the value of t which is valid over all executions of the program. Such an expression α will be said to cover t. We assume that α holds over all paths from the start block to the block where t is located and furthermore, α is a term in a first order language; that is an expression containing no predicates and built from function signs, constant signs, and variables on input to particular blocks of assignment statements.

We now consider Kildall's[Ki] "expression optimizations" for improving the efficiency of object code derived from text expressions, and relate these optimizations to covers.

1) constant propagation (or folding) is the substitution of constant signs for text expressions covered by constants.

2) More generally, a text expression t located at block n is redundant if on all paths from the start block to n another text expression t' yields a computation equivalent to that of t. Thus t may be replaced by a load operation from a temporary address containing the result of some such equivalent previous computation. In a somewhat restricted

version of this optimization, each such t' has the same cover as t.

3) Code motion is the process of moving code as far as possible out of cycles in the control flow graph (i.e. out of program loops). The birth point of text expression t is the earliest block n in the control flow graph (relative to the partial ordering of blocks by domination with the start block first) where the computation of t is defined. Any block n occurring between (relative to this domination ordering) n and the original location of t has a cover for t in terms of covers for the variables at n. The earliest such block m, with the further property that the computation of t can induce no new errors at block m, is called the safe point of t; the computation of t may safely be moved to m. (The text expression appropriate at node n may not be lexically identical to t, but is given by the cover of t in terms of the variables on input to m.) The safety of code movement is also discussed in [CA,G,E,Ke1] and in Chapter 5 we discuss other restrictions to code motion in detail.

4) A cover for a variable on exit from a block in a program loop is a loop invariant. This problem is dicussed in detail in Fong and Ullman[FU] and Wegbreit[W].

Various algorithms[A,C,GW,HU2,KU1,Ke2,Ke3,S, T4,U] have been developed for solving "easy" versions of global flow problems where the transformations through blocks can be computed by bit vector operations. Kildall[Ki] formulates the above expression optimizations in a more general manner so that transformations through blocks are computed by operations on expressions, rather than on bit vectors. Kildall's expression optimizations may give considerably more powerful results than the easier code improvements; however, we shall demonstrate in Section 1.4 that it is not possible in general to compute exact solutions of Kildall's expression optimization problems in the arithmetic domain. (Kam and Ullman[KU2] have recently demonstrated that there exist global flow problems posed in certain non-arithmetic global flow analysis frameworks which are unsolvable.) It follows that we must look for heuristic methods for good, but not optimal, solutions to these problems.

In order to compare our methods with others we must fix the relevant parameters of the program and control flow graph. Let n and a be the cardinality of the node and edge sets, respectively, of the control flow graph; and let σ be the number of variables occurring within more than one block of the program (if we built into the programming language a construct for the declaration of variables local to a block, then the parameter σ is the number of <u>global</u> variables); and let ℓ be the length of the program text. Our careful consideration of the parameter ℓ - avoiding, for example, redundant representations of the same expression - is one of the novelties of our approach; previous authors have analyzed their algorithms primarily from the point of view of the control flow graph parameters n and a.

Kildall[Ki] presents an algorithm, based on an iterative method, for computing approximate solutions to various expression optimization problems. A version of the Kildall algorithm used for the discovery of constant text expressions may require $\Omega(\sigma(\ell+a))$ elementary steps and $\Omega(\sigma a)$ operations on bit vectors of length $O(\sigma\ell)$. ($\Omega(f(x))$ is a function bounded from _below_ by $k \cdot f(x)$ for some k. See Knuth[Kn2].) Kam and Ullman [KU2] show that the Kildall algorithm discovers only a restricted class of text expressions covered by constant signs.

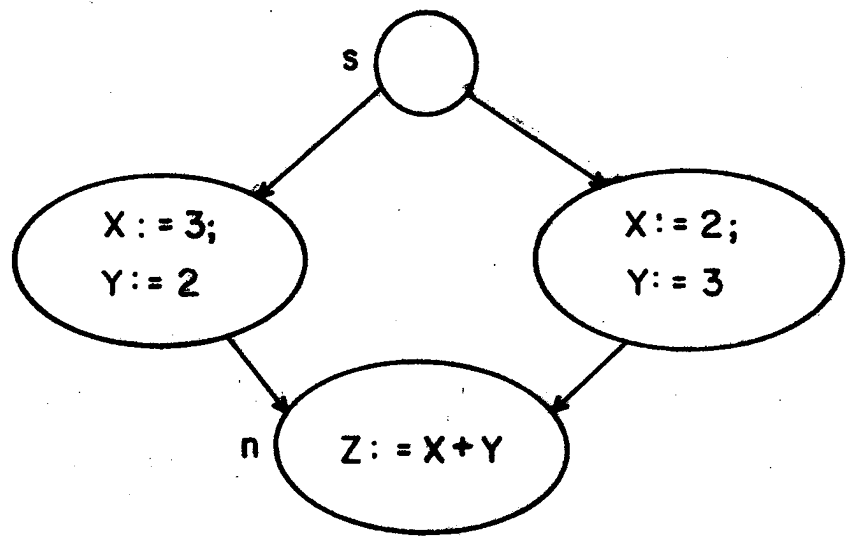


Figure 1.2. (From [KU2]) $Z^{n\rightarrow} = X^{\rightarrow n} + Y^{\rightarrow n}$ is a text expression which is covered by a constant sign but is not discovered by Kildall's algorithm.

Kildall's algorithm may also be used to compute a certain class of covers, which we characterize as fixed points of a functional Ψ mapping approximate covers to improved covers. Fong, Kam, and Ullman[FKU] give another algorithm, based on a direct (noniterative) method which gives weaker results than Kildall's algorithm and is restricted to reducible flow graphs. Kildall's algorithm may require $\Omega(\ell n^2)$ elementary steps and Fong, Kam, and Ullman's algorithm may require $\Omega(\ell a \log(a))$ elementary steps. A main inefficiency of both of these algorithms is in the representation of the covers. Directed acyclic graphs (dags) are used to represent expressions, but separate dags are needed at each node of the flow graph. Since a dag representing a cover may be of size $\Omega(\ell)$ the total space cost may be $\Omega(\ell n)$. Various operations on these dags, which are considered to be "extended" steps by Fong, Kam, and Ullman[FKU], cost $\Omega(\ell)$ elementary steps and cannot be implemented by any fixed number of bit vector operations. In general, any global flow algorithm for symbolic evaluation which attempts to pool information separately at each node of the flow graph will have time cost of $\Omega(\ell a)$, since the pools on every pair of adjacent nodes must be compared. Since $\ell \geq n$, such a time cost may be unacceptable for practical applications.

The _global value graphs_ used in Chapter 2 are related to a structure used by Schwartz[Sc2] to represent the flow of values through the program. The use of a special global value graph GVG* leads to a relatively efficient direct method for symbolic evaluation which works for all flow graphs. The method derives its efficiency by representing the covers with a single dag, rather than a separate dag at each node. GVG* is of size $O(\sigma a+\ell)$, although the results of Chapter 4 may be used to build a global value graph GVG$^+$ which in many cases is of size $O(a+\ell)$ but may grow to the same size as GVG*. In elementary operations, the time cost of our algorithm for the discovery of constants (the constants found by Kildall's algorithm) is linear in the size of GVG$^+$, and our algorithm for finding the cover which is the minimal fixed point of Ψ requires time almost linear in the size of the GVG$^+$. (Our algorithms work for all flowgraphs.) Thus our algorithm for symbolic evaluation takes time almost linear in $\sigma a+\ell$ ($a+\ell$ in many cases), as compared to Kildall's which may require $\Omega(\ell n^2)$ steps.

1.2 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, $1 \leq i < k$. The length of the path p is k-1.

The path p may be built by composing subpaths:

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

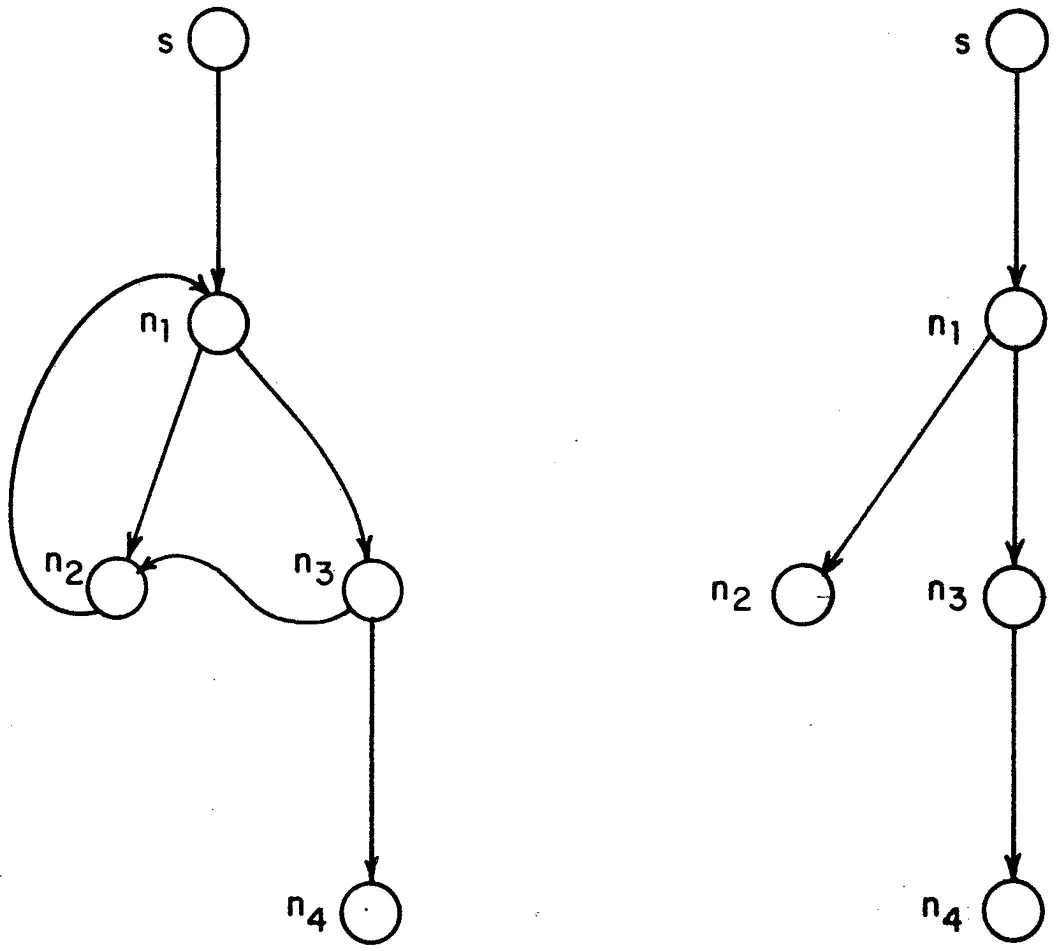
The path p is a cycle if u = w. A strongly connected component of G is a maximal set of nodes contained in a cycle.

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

We shall require various sorts of special digraphs. A rooted digraph (V, E, r) is a triple such that (V, E) is a digraph and r is a distinguished node in V, the root. A flow graph is a rooted digraph such that the root r has no predecessors and every node is reachable from r. A digraph is _labeled_ if it is augmented with a mapping whose domain is the vertex set. A _oriented digraph_ is a digraph augmented with an ordering of the edges departing from each node. We shall allow any given edge of an oriented graph to appear more than once in the edge list.

A digraph G is _acyclic_ if G contains no cycles, _cyclic_ otherwise. Let G be acyclic. If u is reachable from v, u is a _descendant_ of v and v is a _ancestor_ of u (these relations are _proper_ if $u \neq v$). 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 rooted digraph G = (V, E, r) is a tree with node set V, an edge set contained in E, and a root r.



<u>Figure 1.3</u>. A flow graph and its dominator tree.

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. The symbols $\overset{*}{\rightarrow}$, $\overset{+}{\rightarrow}$, $\rightarrow$ denote the dominator, proper dominator, and immediate dominator relations, respectively.

All of the above properties of digraphs may be computed very efficiently. An algorithm has *linear time cost* if the algorithm runs in time O(n) on input of length n and has *almost linear time cost* if the algorithm runs in time $O(n\alpha(n))$ where α is the extremely slow growing function of [T3] (α is related to a functional inverse of Ackermann's function). Using adjacency lists, a digraph G = (V, E) may be represented in space $O(|V|+|E|)$. Knuth[Kn1] gives a linear time algorithm for computing a topological ordering of an acyclic digraph. Tarjan [T1] presents linear time algorithms for computing the strongly connected components of a digraph and a spanning tree and in [T4] gives an almost linear time algorithm for computing the dominator tree of a flow graph.

1.3 The Global Flow Model.

Let P be a program to which we wish to apply various global code improvements. In this section we formulate a global flow model for P, similar to a model described by Aho and Ullman[AU1] and others.

The control flow graph F = (N, A, s) is a flow graph rooted at the start node s $\in$ N. A control path is a path in F. Hereafter $\overset{*}{\rightarrow}$, $\overset{+}{\rightarrow}$, $\rightarrow$ will denote the dominator, proper dominator, and the immediate dominator relations with respect to the fixed rooted digraph F.

As described in Section 1.1, each node n $\in$ N is a block of assignment statements. These blocks do not contain conditional or branch statements; control information is specified by the control flow graph.

Program variables are taken from the set {X, Y, Z,...}. An assignment statement of P is of the form

$$X := \alpha$$

where X is a program variable and α is an expression built from program variables and fixed sets C of constant signs and θ of function signs. A program variable occurring within only a single block n $\in$ N is local to n. Let Σ be the set of program variables occurring within P and not local to any block. For each program variable X $\in$ Σ and block n $\in$ N-{s} we introduce the input variable $X^{\rightarrow n}$ to denote the value of X on __entry__ to block n. We use the symbol $X^{\to s}$, considered to be a constant sign, to denote the value of X on input to the program P at the start block s.

Let EXP be the set of expressions built from input variables, C, Θ. Thus, $\alpha \in$ EXP is a finite expression consisting of either a constant sign $c \in C$, an input variable $X^{\to n}$ representing the value of program variable $X^{\to n}$ on input to block n, or a k-adic function sign $\theta \in \Theta$ prefixed to a k-tuple of expressions in EXP. The text expressions as well as the covering expressions sought are expressions in EXP. For each $X \in \Sigma$ and block $n \in N$ such that X is assigned to at n, let the __output expression__ $Xn^{\to}$ be an expression in EXP for the value of X on exit from block n in terms of constants and input variables at block n. A __text expression__ t is an output expression or a subexpression of an output expression. Note that each text expression t corresponds to a string of text on the right hand side of an assignment statement of P.

For example, let n be the block of code:

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

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

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

An expression in EXP is put in <u>reduced form</u> by repeatedly substituting for each subexpression of the form $(\theta\ c_1 \ldots c_k)$, that constant sign c such that $I(c) = I(\theta)(I(c_1),\ldots,I(c_k))$, until no further substitutions of this kind can be made. We assume the blocks are <u>reduced</u> in the sense of Aho and Ullman[AU1], so each text expression is a reduced expression. We also assume that the output expressions $X^{n\rightarrow}$ are reduced (and thus uniquely determined).

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

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

Let α be an expression in EXP and let p be a control path beginning at the start block s and containing origin(α). Then note that each node in $N(\alpha)$ is contained in p. We give a recursive definition for EXEC(α,p), the expression for the value of α in the context of this control path p. EXEC(α,p) is defined formally as follows:

i) if p = (s) then EXEC(α,p) is the reduced expression derived from α.

ii) otherwise, if $p = p'\cdot(m,n)$ then EXEC(α,p) = EXEC(α',p') where α' is the expression obtained from α by substituting the output expression $Xm^{\rightarrow}$ for each input variable $X^{\rightarrow n}$, and putting the result in reduced form.

An expression $\alpha \in$ EXP <u>covers</u> a text expression t if

EXEC(t,p) = EXEC(α,p)

for every control path p from s to origin(t). Hence, if α covers t then α correctly represents the value of t on every execution of program P. For example in Figure 1.1, $Z^{n4\rightarrow}$ is covered by $Z^{\rightarrow n_1} * Y^{\rightarrow s}$. Note that the origin of any cover α of a text expression t is always well defined since the elements of N(α) will form a chain relative to $\overset{*}{\rightarrow}$.

A <u>cover</u> is a mapping ψ from the text expressions of P to expressions in EXP in reduced form such that for each text expression t, ψ(t) covers t.

<u>Lemma 1.3</u> If $\alpha \in$ EXP covers text expression t then origin(α) $\overset{*}{\rightarrow}$ origin(t).

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

We extend $\overset{*}{\rightarrow}$ to a partial ordering of covers. For each pair of covers ψ_1 and ψ_2, $\psi_1 \overset{*}{\rightarrow} \psi_2$ iff origin(ψ_1(t)) $\overset{*}{\rightarrow}$ origin(ψ_2(t)) for all text expressions t.

We wish to compute covers minimal with respect to this partial ordering.

1.4 Unsolvability of Various Code Improvements Within the Arithmetic Domain

The introduction listed a number of code improvements which are related to the problem of determining minimal covers of text expressions. Here we show that even constant propagation, the simplest of these improvements, is recursively unsolvable in the arithmetic domain. Previously, Kam and Ullman [KU2] have shown related global flow problems to be insolvable in an abstract, nonarithmetic domain.

Theorem 1.4. In the arithmetic domain, the problem of discovering all text expressions covered by constant signs is undecidable.

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

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

Consider the flow graph F_Q of Figure 1.4. Let t be the text expression $X_0{\uparrow}f/(1+Q(X_1{\uparrow}f, \ldots, X_k{\uparrow}f)^2)$ located at block f. We show t is covered by a constant sign iff Q has no root in the natural numbers.

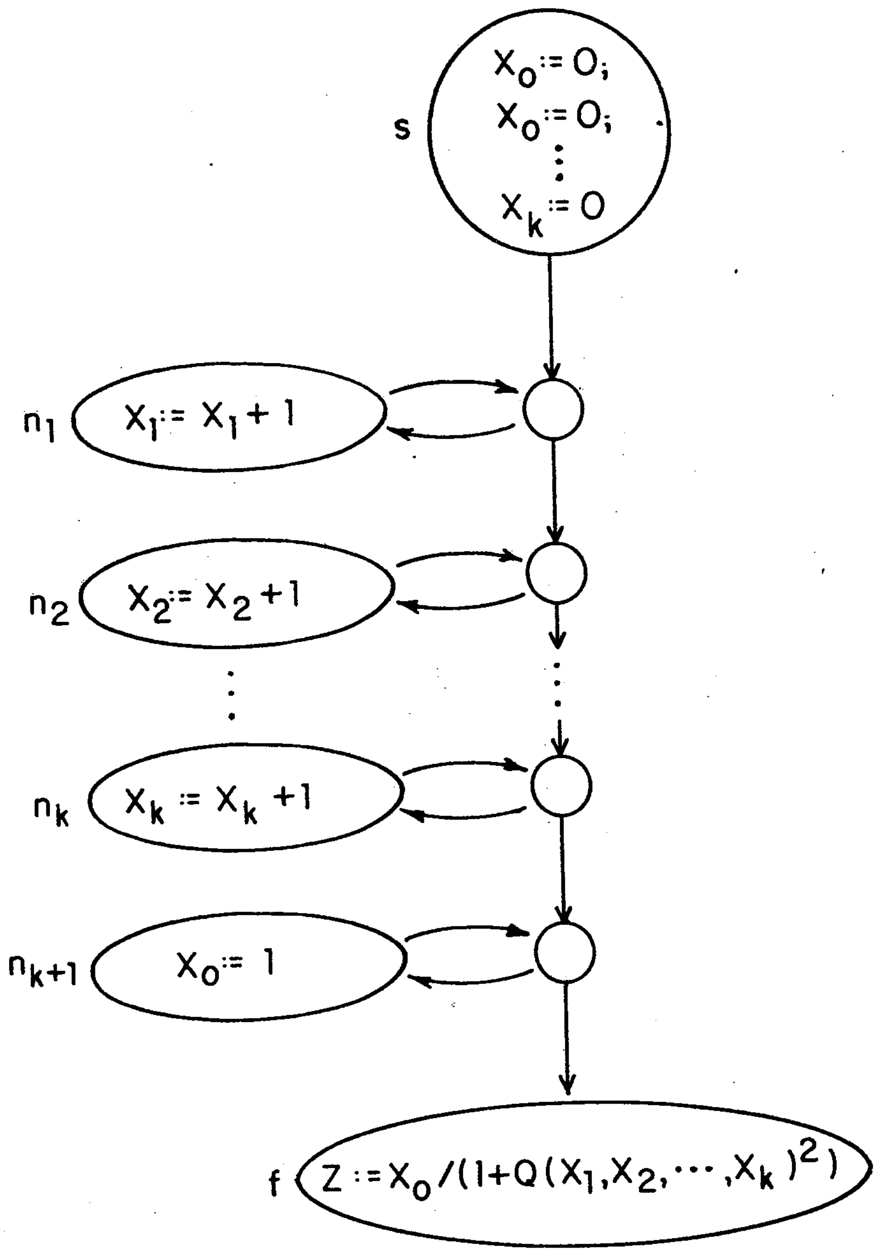


Figure 1.4. The control flow graph F_Q.

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

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

ONLY IF. Suppose Q has a root z in the natural numbers. Then it is possible to find execution paths p and q from s to f such that z = X(p) = X(q) and such that X(p) = 0, X(q) = 1. Hence EXEC(t,p) = 0 and EXEC(t,q) = 1, so t is not covered by a constant sign. □

Corollary 1.4. In the arithmetic domain, the following global flow problems are unsolvable: discovery of minimal covers, birth and safe points of code motion, redundant text expressions, and loop invariants.

Proof. It is easy to show that the problem of discovery of constant text expressions reduces to each of these problems. Add the edge (f,n_1) to the control flow graph F of Figure 1.4, so t is contained is a cycle of F. Then by Theorem 1.4, Q has no root in the natural numbers iff t is covered by 0

iff s is the birth point of t

iff s is the safe point of t

iff t is redundant on entry to f

iff t is a constant loop invariant.

Thus, the problem of discovery of whether text expression t is covered by a constant reduces to each of the above global flow problems. (Note that the problem of safety of code motion is also hard for other reasons; if we add the text expression $t' = 1/Q(\vec{X1}^{f},\ldots,\vec{Xk}^{f})$ to block f then Q has no root in the natural numbers iff t' is safe at f.) □

The above results indicate that we must look for methods for computing approximations to minimal covers. The method of Kildall[Ki] may be applied to compute such a class of covers. In Chapter 2 we define a functional Ψ mapping (as each iteration of Kildall's algorithm might) covers to covers by comparing covers of input variables at given blocks in N to covers of corresponding output expressions of variables at immediately preceding blocks. We show that the minimal fixed point of Ψ exists, is unique, and give an efficient algorithm for computing this cover.

In Chapter 3 we extend our results to programs which manipulate lists and expressions, such as LISP 1.0.

An extremely efficient algorithm is presented in Chapter 4; this almost linear time algorithm yields a cover which is weaker than the minimal fixed point of Ψ computed in Chapter 2, but is probably good enough for most applications.

Finally, in Chapter 5 we investigate in some depth a program improvement called <u>code motion</u> which is the process of moving computations as far as possible out of control cycles into new locations which the computations are executed less frequently. Covers computed by methods of previous chapters are useful here since we must discover the earliest block (relative to the domination ordering) in the control flow graph where the computations are defined.

1.5 Adding Procedure Calls to the Global Flow Model

This section describes how the global flow model of Chapter 1.3 may be extended to take into account non-recursive procedure declarations and calls. That is, we now extend the programming language so as to include in addition to assignment statements, procedure declarations and procedure call statements, and then show how to construct appropriate control flow graphs for programs in the extended language. We assume dynamic binding of procedure variables and call-by-value, though this scheme could be modified to allow static binding or call-by-reference.

We assume a main body M of program text with associated control flow graph (N_M, A_M, s_M).

A procedure declaration is of the form

procedure $R(X_1, \ldots, X_k)$ <procedure body>

where the procedure body contains an arbitrary string in the programming language, followed by a statement of the form:

result t_R;

where t_R is a text expression. The associated control flow graph $F_R = (N_R, A_R, s_R)$ and node $f_R \in N_R$ specify the flow of control within the procedure body of R; all invocations of R start at s_R and finish at f_R. We assume the result statement is the only statement located at f and f has no departing edges in A_R. Each call to R (which may be located within the main program or in the body of any procedure including that of R itself) is an assignment of the form:

$X := R(t_1,\ldots,t_k)$

where $t_1,\ldots,t_k$ are text expressions, and the execution of this call to R invokes an execution of the body of R, with X_j set to the current value of t_j just before entering R, for $j = 1,\ldots,k$. (Hence, we assume call-by-value rather than call-by-reference.) Let r be the value of the result expression t_R. On exit from the body of R, the values of the variables $X_1,\ldots,X_k$ are reset to their original values just before entering R <u>on this invocation</u>. (Hence, we assume dynamic binding rather than static binding.) Finally, the program variable X assigned to $R(t_1,\ldots,t_k)$ is set to value r.

The control flow graph F_P for program P is constructed by

(1) first merging the control flow graph of the main body and the control flow graphs of all procedure bodies,

(2) setting s_M to be the start block,

(3) for each result statement

<u>result</u> t_R

substitute the assignment

$X_R := t_R$

where X_R is a new program variable not assigned anywhere else in the program, and

(4) for each block n of the form:

$stm_1;\ldots;stm_{i-1};stm_i;\ldots stm_k$

where stm_i is a procedure call $X := R(t_1,\ldots,t_k)$,

(a) delete block n and substitute in its place the blocks

n_1 = "$stm_1;\ldots stm_{i-1};X_1:=t_1;\mathfrak{X}_1:=X_1;\ldots\mathfrak{X}_k:=X_k;X_k:=t_k$"

n_2 = "$X_1:=\mathfrak{X}_1;\ldots;X_k:=\mathfrak{X}_k;X:=X_R;stm_{i+1};\ldots;stm_k$"

where $\mathfrak{X}_1,\ldots,\mathfrak{X}_k$ are new program variables.

(b) Add edges (n_1,s_R) and (f_R,n_2) to the edge set and for each edge (m,n) entering n substitute an edge (m,n_1) entering n_1, and for each edge (n,m) departing from n substitute the edge (n_2,m) departing from n_2.

CHAPTER 2

SYMBOLIC EVALUATION AND THE GLOBAL VALUE GRAPH

2.0 Summary.

As in Chapter 1, we assume a global flow model in which the expressions computed are specified, but the flow of control is indicated only by a directed graph whose nodes are blocks of assignment statements. We develop a direct (non-iterative) method for finding general symbolic values for expressions in the text of the program. Our method gives results similar to an iterative method due to Kildall[Ki] and a direct method due to Fong, Kam, and Ullman[FKU]. By means of a structure called a _global value graph_ which compactly represents both symbolic values and the flow of these values through the program, we are able to obtain results that are as strong as either of these algorithms at a lower time and space cost, while retaining applicability to all flow graphs.

2.1 *Introduction*.

Let us review the basic definitions of the global flow model defined in Section 1.2. Let P be the program which we wish to analyze and improve. The flow of control through P is represented by the *control flow graph* $F = (N, A, s)$ where N is a set of blocks of assignment statements, A is a set of edges specifying possible flow of control immediately between blocks, and $s \in N$ is the *start block* from which all flow of control begins. A *control path* is a path in F. Let $\rightarrow$, $\overset{+}{\rightarrow}$, $\overset{*}{\rightarrow}$ denote respectively the immediate dominator relation, proper dominator relation, and the dominator ordering, which is a partial ordering.

Let {X, Y, Z,...} be the set of program variables, and let Σ be the set of program variables occurring within more than one block of N. For each $n \in N-\{s\}$ and program variable $X \in \Sigma$ we introduce the *input variable* $X^{\rightarrow n}$ to denote the value of X on *entry* to block n. Also, $X^{\rightarrow s}$ represents the value of program variable $X \in \Sigma$ on entry to the program P at the start block s; $X^{\rightarrow s}$ is considered to be a constant sign rather than an input variable. Let EXP be a set of expressions built from input variables and fixed sets of constant and k-adic function signs. For each program variable $X \in \Sigma$ and block $n \in N$ such that X is assigned to at n, let $X^{n\rightarrow}$ denote the expression in EXP for the value of X on *exit* from n in terms of constants and input variables at

n. $X^{n\rightarrow}$ is called the _output expression for X at n_. The _text expressions_ of P are the output expressions plus their subexpressions. Note that input variable $X^{\rightarrow n}$ is a text expression only if X occurs in the right hand side of an assignment statement of block n before X is assigned to. In this section and the next, it will be useful to assume that the text expressions include _all_ input variables; for each X ϵ Σ and block n ϵ N-{s} such that $X^{\rightarrow n}$ is not an input variable, add at n the dummy assignment X := X.

An _interpretation_ was defined in Section 1.2 to contain a universe U of values and mappings from contant signs to U, and from k-adic function signs to mappings from U^k to U. An expression in EXP is _reduced_ relative to an interpretation by repeatedly substituting constant signs for constant subexpressions.

For each α ϵ EXP, origin(α) is the earliest block n relative to the domination ordering $\overset{*}{\rightarrow}$ (with the start block s first) such that every block referred to in α is contained on all control paths from s to n. For each control path p from s and containing origin(α), EXEC(α,p) is intuitively the expression in EXP for the value of α relative to p. An expression α ϵ EXP _covers_ text expression t if

$$\text{EXEC}(t,p) = \text{EXEC}(\alpha,p)$$

for all control paths from s to origin(t). A _cover_ ψ is a mapping from text expressions to EXP such that ψ(t) covers t for each text expression t. We use origin to induce the partial ordering $\overset{*}{\rightarrow}$ of covers; for each pair of covers ψ and ψ', $\psi \overset{*}{\rightarrow} \psi'$ iff origin(ψ(t)) $\overset{*}{\rightarrow}$ origin(ψ'(t)) for all text expressions t.

In Chapter 1 we showed that the problem of computing covers minimal with respect to $\overset{*}{\rightarrow}$ over arithmetic domains is unsolvable; hence we consider a simple class of covers that might be computed by an algorithm due to Kildall. To construct this class of covers, Kildall's algorithm would first take a pass through the program and construct a mapping ψ_0 from text expressions to EXP; ψ_0 may not be a cover but has the property that for all text expressions t,

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

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

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

$= X^{\rightarrow n}$, otherwise.

and ψ'(t) is the reduced expression derived from text expression t after substituting $\psi'(X^{\rightarrow n})$ for each input variable $X^{\to n}$ occurring in t. Kildall's algorithm computes $\Psi^k(\psi_0)$ for k = 1,2,... until a fixed point of Ψ is obtained. Note that Ψ maps covers to covers; but Ψ need not be monotonic, i.e. for some cover ψ and text expression t, it may happen that $\Psi(\psi)(t) \not\geq \psi(t)$.

Theorem 2.1. Each ψ which is a fixed point of Ψ is a cover, i.e. $\text{EXEC}(\psi(t),p) = \text{EXEC}(t,p)$ for all text expressions t and control paths p from s to the block where t is located.

Proof by construction. Let p be the shortest control path from s to a block n where there is located a text expression t such that

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

Thus t must contain an input variable $X^{\to n}$ such that

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

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

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

In Section 2.2, we show that Ψ has a unique minimal fixed point ψ^*. We then show (Sections 2.3-2.7) that while the problem of finding minimal covers is hopeless, that of finding ψ^* is not only solvable but can be done efficiently. Thus we provide an efficient algorithm for finding the minimal cover among those of the type computed iteratively by Kildall's algorithm.

In fact the rest of this chapter is presented in a more general setting than is suggested above, so as to lay the foundation for related algorithms in Chapter 3 which deal with programs that operate on structured data. The overall plan is to introduce (in Section 2.2) a special class of graphs called _global value graphs_ which represent the flow of _values_ (rather than _control_) through the program P; and we define, for each global value graph GVG, a set Γ_{GVG} of approximate covers associated with it. Γ_{GVG} is in each case a finite semilattice which thus has a unique minimal element ψ_{GVG}, and which is efficiently calculated by the algorithm presented in Sections 2.3-2.7. As we show in Sections 2.2 and 2.8, for a particular choice of GVG, ψ_{GVG} is actually ψ^*, the minimal fixed point of the functional Ψ, so our general algorithm can be used to find ψ^* efficiently. (Indeed, the whole presentation could be made uniform by replacing the functional Ψ in an appropriate way by a functional Ψ_{GVG} that depends on the particular global value graph; then ψ_{GVG}, the minimal element of Γ_{GVG}, would be in each case the minimal fixed point of Ψ_{GVG}. We have chosen not to do so, since only Ψ as defined here has any historical significance.)

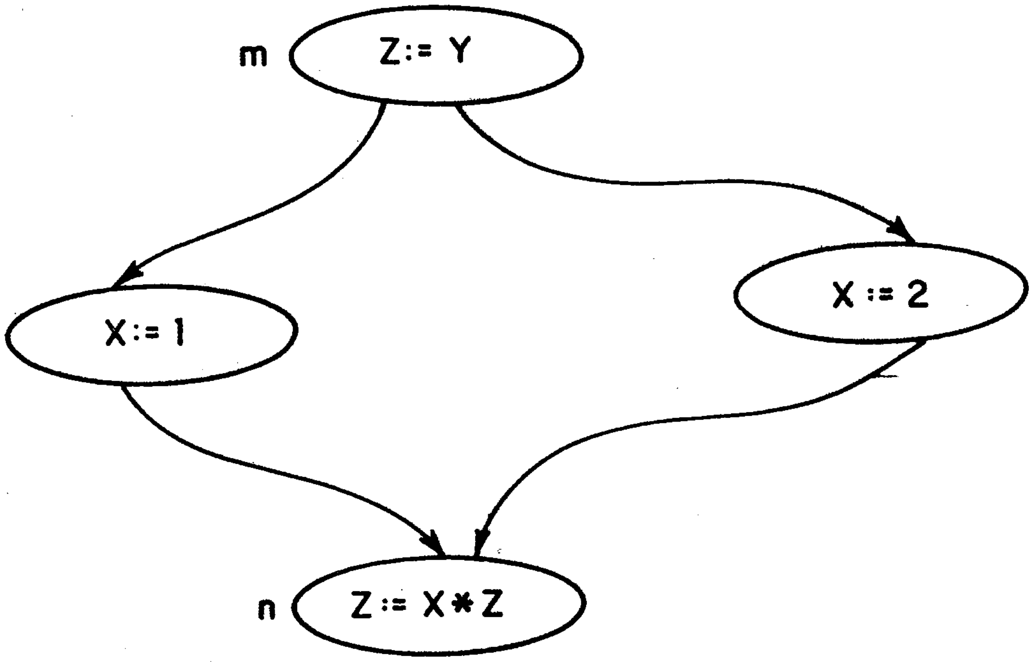


Figure 2.1. A fixed point of Ψ covers Z^{n+} with the expression $X^{\rightarrow n} * Y^{\rightarrow m}$.

2.2 Dags and Global Value Graphs.

A labeled dag D = (V, E, L) is a labeled, acyclic, oriented digraph with a node set V, an edge list E giving the order of edges departing from nodes, and a labeling L of the nodes in V. A rooted labeled dag (D,r) represents an expression α if α is the parenthesized listing of the labels of the subgraph of D rooted at r in topological order from r to the leaves and from left to right. (Where D is fixed, we simply say r represents α if (D,r) so represents α).

The dag D is minimal if each node r ϵ V represents a distinct expression. Any expression or set of expressions may be represented, with no redundancy, by a minimal labeled dag D. In particular, we use the minimal dag D(n) to represent efficiently the set of text expressions located at block n. We have assumed that each block is reduced, so each node in D(n) corresponds to a unique text expression. Aho and Ullman[AU1] describe the use of dags for representing computations within blocks. Kildall[Ki] and Fong, Kam, and Ullman[FKU] have applied dags to various global flow problems.

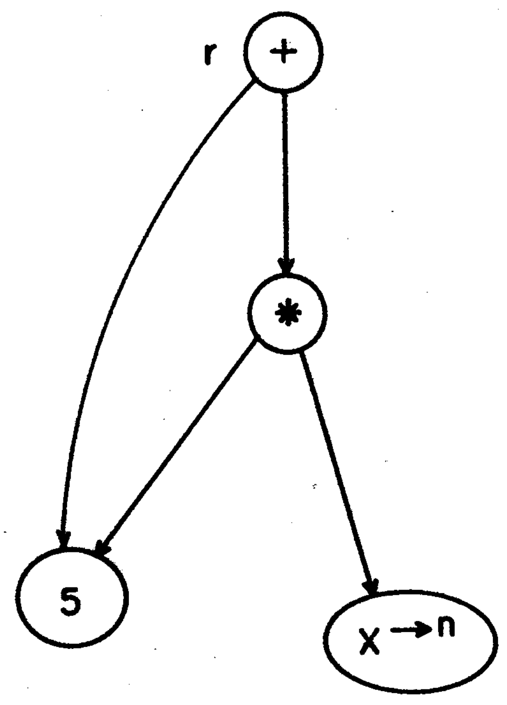


Figure 2.2. (D,r) represents (5+(5*X→n)) (or more properly in prefix notation (+ 5 (* 5 X→n))) where D is the above dag.

We now come to the central definition. To model the flow of values through a program P, we introduce a class of labeled digraphs called <u>global value graphs</u> derived by combining the dags of all the blocks in N and adding a set of edges called <u>value edges</u> which pair nodes labeled with input variables to other nodes. More precisely, a global value graph is a possibly cyclic, labeled, oriented digraph GVG = (V, E, L) such that:

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

(2) E is an edge list containing (a) the edge list of each D(n) and (b) a set of pairs in V^2 (the <u>value edges</u> of GVG) such that (i) the first node of each pair is labeled with an input variable and (ii) for each $v \in V$ labeled with an input variable $X^{\to n}$, and control path p from s to n, there is some value edge departing from v and entering a node located at a block in p and distinct from n.

(3) L is a labeling of V compatible with the labeling of each D(n).

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

A <u>value path</u> is a path in GVG traversing only nodes linked by value edges; a value path is <u>maximal</u> relative to a fixed beginning node if its last node has no departing value edges.

<u>Lemma 2.2.1</u> For any $v \in V$ labeled with an input variable and any control path p from the start block s to loc(v), there is a maximal value path q from v such that all the nodes in q have distinct loc values in p.

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

We assume here, as in Section 2.1, that the set of text expressions of each block $n \in N$ include all input variables at n. Let Γ_{GVG} be the set of mappings ψ from V to EXP such that for all $v \in V$,

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

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

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

Note that for any node v satisfying (2), $\psi(v)$ is determined from the input variables occurring in the text expression which v represents. Hence any $\psi \in \Gamma_{GVG}$ is uniquely specified by the set of input variables satisfying case (3a), so Γ_{GVG} has at most $2^{|N||\Sigma|}$ elements.

<u>Lemma 2.2.2</u>. For any $\psi \in \Gamma_{GVG}$ and $v \in V$, origin(ψ(v)) $\overset{*}{\rightarrow}$ loc(v).

<u>Proof</u> by contradiction. Suppose for some $v \in V$,

origin(ψ(v)) $\overset{*}{\not\rightarrow}$ loc(v).

Hence, there must be an input variable $X^{\rightarrow n}$ occurring in ψ(v) such that n $\overset{*}{\not\rightarrow}$ loc(v), and so there is an n-avoiding path p from the start block s to loc(v). Also, there must exist some $u \in V$ labeled with an input variable and also located at block n, such that $\psi(u) = X^{\rightarrow n}$. By Lemma 2.2.1, we can construct a maximal value path $(u=u_1,\ldots,u_k)$ such that $loc(u_1),\ldots,loc(u_k)$ are distinct blocks in p. Let j be the maximal integer $\leq k$ such that $\psi(u_1) =\ldots= \psi(u_j)$. If $L(u_j)$

is an input variable, then $\psi(u_1) = L(u_j) = X^{\to n}$, so $loc(u_j) = n$ is contained in p, contradicting the assumption that p contains n. Otherwise, if $L(u_j)$ is not an input variable then neither is $\psi(v) = \psi(u_j)$, a contradiction with the assumption that $\psi(u) = X^{\to n}$. $\square$

We shall show that Γ_{GVG} is a finite semilattice with ordering $\overset{*}{\to}$, and hence has a minimal element. Then we shall define a global value graph GVG* such that the minimal fixed point of Ψ, the functional defined in the last section, is the minimal element of Γ_{GVG^*}.

We define a partial mapping <u>min</u>: $EXP^2 \to EXP$ such that for all $\alpha, \alpha' \in EXP$,

α <u>min</u> $\alpha' = \alpha$ if $origin(\alpha) \overset{+}{\to} origin(\alpha')$

$= \alpha'$ if $origin(\alpha') \overset{+}{\to} origin(\alpha)$

or if $origin(\alpha) = origin(\alpha')$ and

(i) if $\alpha = \alpha'$ then α <u>min</u> $\alpha' = \alpha = \alpha'$, or

(ii) if α is a constant sign and α' is a function application, then α <u>min</u> $\alpha' = \alpha'$ <u>min</u> $\alpha = \alpha$, or

(iii) if α, α' are function applications $(\theta\ \alpha_1 \ldots \alpha_k)$, $(\theta\ \alpha'_1 \ldots \alpha'_k)$ respectively, and $\overline{\alpha}_i = \alpha_i$ <u>min</u> α'_i is defined for $i = 1, \ldots, k$ then α <u>min</u> $\alpha' = (\theta\ \overline{\alpha}_1 \ldots \overline{\alpha}_k)$,

and otherwise, α <u>min</u> α' is undefined.

We extend <u>min</u> to the partial mapping from pairs of elements of Γ_{GVG} to Γ_{GVG} defined thus: for $\psi, \psi' \in \Gamma_{GVG}$, if for all $v \in V$ $\psi(v)$ <u>min</u> $\psi'(v) = \overline{\psi}(v)$ is defined then ψ <u>min</u> ψ'

$= \overline{\Psi}$ and otherwise ψ <u>min</u> ψ' is undefined.

<u>Theorem 2.2.1</u>. Γ_{GVG} is a semilattice with ordering $\overset{*}{\rightarrow}$.

<u>Proof</u> It is sufficient to show <u>min</u> is well defined over Γ_{GVG}. We proceed by induction. Suppose for $\psi, \psi' \in \Gamma_{GVG}$ and some α in the domain of Ψ, $\psi(u)$ <u>min</u> $\psi'(u)$ is defined for all $u \in V$ such that $\psi(u)$ is a proper subexpression of α. Consider some text expression v such that $\psi(v) = \alpha$. By Lemma 2.2.2, both origin(ψ(v)) and origin(ψ'(v)) are contained on all control paths from the start block s to loc(v), so we may assume without loss of generality that origin(ψ(v)) $\overset{*}{\rightarrow}$ origin(ψ'(v)). Observe that $\psi(v)$ <u>min</u> $\psi'(v) = \psi(v)$ if origin(ψ(v)) $\overset{+}{\rightarrow}$ origin(ψ'(v)) so we further assume that origin(ψ(v)) = origin(ψ'(v)).

<u>Case 1</u>. If L(v) is a constant sign c then $\psi(v) = \psi'(v) = c$ so $\psi(v)$ <u>min</u> $\psi'(v) = c$.

<u>Case 2</u>. Suppose L(v) is a function sign θ and v has immediate successors $u_1, \ldots, u_k$. By the induction hypothesis $\alpha_i' = \psi(u_i)$ <u>min</u> $\psi'(u_i)$ is defined for $i = 1, \ldots, k$. Hence $\psi(v)$ <u>min</u> $\psi'(v)$ is the reduced expression derived from (θ $\alpha_1' \ldots \alpha_k'$).

<u>Case 3</u>. Otherwise, suppose L(v) is an input variable. Let p be a control path from the start block s to loc(v). By Lemma 2.2.1, we can construct a maximal a value path $(v{=}u_1, \ldots, u_k)$ such that for $i = 1, \ldots, k$ each loc(u_i) is contained in p. Let j be the maximal integer such that $\psi(u_1) = \ldots = \psi(u_j)$.

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

Case 3b. Otherwise, suppose $\psi'(v) = \psi'(u_1) = \ldots = \psi'(u_j)$ so we have $\psi(v) = \psi(u_j)$ and $\psi'(v) = \psi'(u_j)$. Applying Cases 1 and 2, $\psi(v)$ min $\psi'(v) = \psi(u_j)$ min $\psi'(u_j)$ is defined if $L(u_j)$ is either a constant sign or function application, so we assume $L(u_j)$ is an input variable. Since j is maximal, $\psi(v) = \psi(u_j) = L(u_j)$. If $\psi'(v) = \psi'(u_j) = L(u_j)$ then $\psi(v)$ min $\psi'(v) = L(u_j)$. Otherwise, suppose $\psi'(u_j) \neq L(u_j)$. For each value edge (u_j, v'), by the definition of Γ_{GVG}, $\psi'(u_j) = \psi'(v')$ and by Lemma 2.2.2, $\text{origin}(\psi'(v')) \overset{*}{\rightarrow} \text{loc}(v')$. Hence $\text{origin}(\psi'(v)) = \text{origin}(\psi'(u_j))$ is distinct from $\text{origin}(\psi(v))$, contradicting our assumption that $\text{origin}(\psi'(v)) = \text{origin}(\psi(v))$. □

Theorem 2.2.1 immediately implies that:

Corollary 2.2. Γ_{GVG} has an unique minimal element min Γ_{GVG}.

Now we shall define a special global value graph such that ψ^*, the minimal fixed point of Ψ, the functional defined in Section 2.1, is the minimal element of Γ applied to this graph. Again we assume that the text expressions include all the input variables, and add dummy assignments to satisfy this assumption. Let GVG^* be the global value graph containing the value edges $\{(v,u) \mid v$ represents input variable $X^{\rightarrow n}$ and u represents the output expression $X^{m\rightarrow}$ for each program variable $X \in \Sigma$ and edge $(m,n) \in A$ of the control flow graph $F\}$.

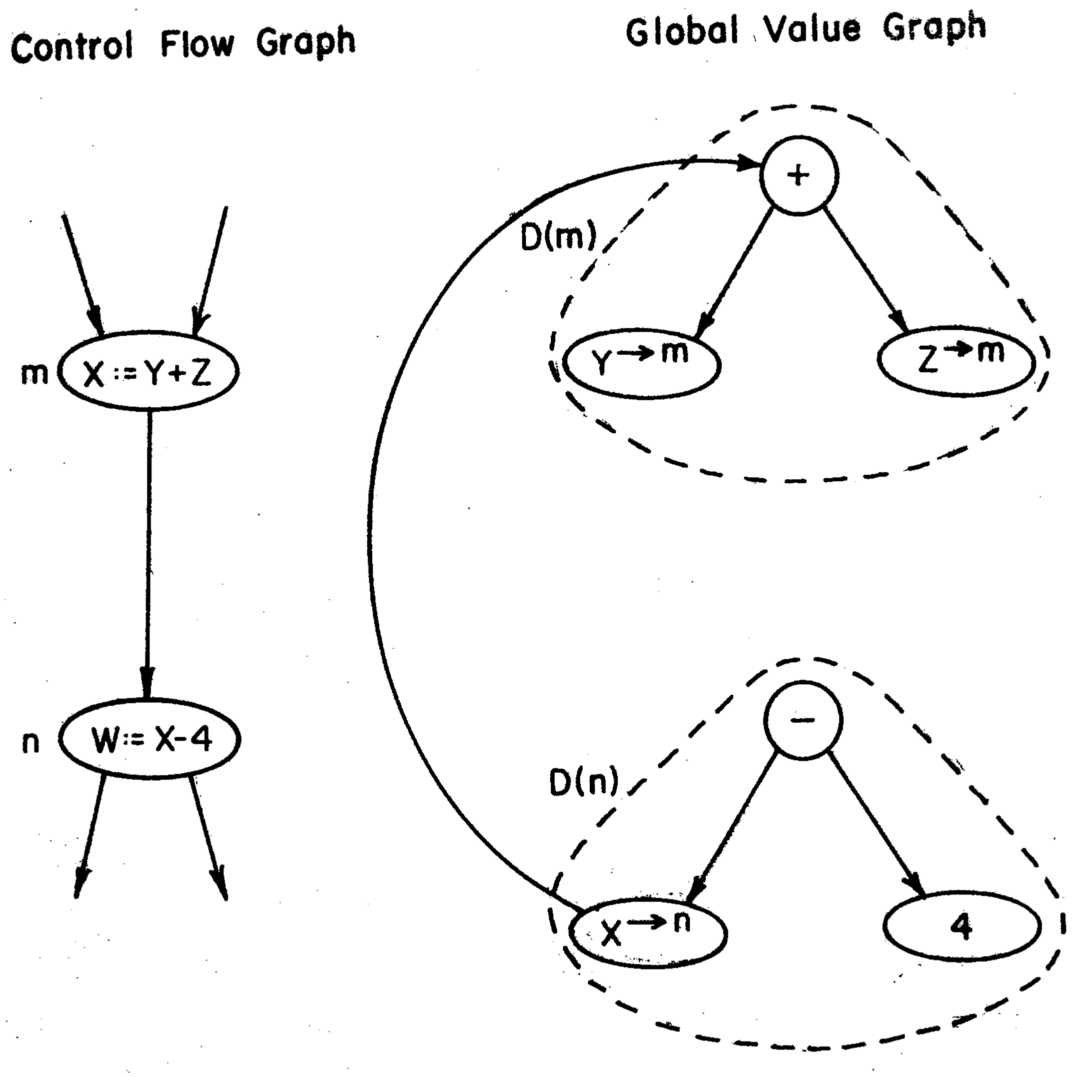


Figure 2.3. The global value graph GVG*.

We have shown that Γ_{GVG^*} is a finite semilattice and hence has a minimal element; we now show that this minimal element is the unique minimal fixed point of Ψ.

<u>Theorem 2.2.2</u> ψ^*, the minimal fixed point of Ψ, is identical to $\hat{\psi}$, the unique minimal element of Γ_{GVG^*}.

<u>Proof</u>. Observe that any fixed point of Ψ is an element of Γ_{GVG^*}. By Corollary 2.2, Γ_{GVG^*} has a unique minimal element $\hat{\psi}$ = <u>min</u> Γ_{GVG^*}. Suppose $\hat{\psi}$ is not a fixed point of Ψ. Observe that since $\hat{\psi} \in \Gamma_{GVG^*}$, for each input variable $X^{\rightarrow n}$, if $\psi(X^{\rightarrow n}) \neq X^{\rightarrow n}$ then $\Psi(\hat{\psi})(X^{\rightarrow n}) = \hat{\psi}(X^{\rightarrow n})$. Hence there is an input variable $X^{\rightarrow n}$ such that $\hat{\psi}(X^{\rightarrow n}) = X^{\rightarrow n}$ but $\Psi(\hat{\psi})(X^{\rightarrow n}) = \alpha$ where $\alpha = \hat{\psi}(X^{m\rightarrow})$ for all blocks m immediately preceding block n in the control flow graph F.

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

<u>Case a</u>. Suppose $\hat{\psi}(Y^{\rightarrow n'}) = Y^{\rightarrow n'}$. If $Y^{\rightarrow n'} \neq X^{\rightarrow n}$ then $\psi(Y^{\rightarrow n'}) = Y^{\rightarrow n'}$. Otherwise, if $Y^{\rightarrow n'} = X^{n\rightarrow}$ then for each block m immediately preceding block n' = n, $\psi(Y^{\rightarrow n'}) = \hat{\psi}(Y^{m\rightarrow}) = \alpha$, and since $X^{\rightarrow n}$ is not contained in α, $\psi(Y^{\rightarrow n'}) = \psi(Y^{m\rightarrow}) = \alpha$.

<u>Case b</u>. If $\hat{\psi}(Y^{\rightarrow n'}) \neq Y^{\rightarrow n}$ then for each block m immediately preceding n' in F, $\hat{\psi}(Y^{\rightarrow n'}) = \hat{\psi}(Y^{m\rightarrow})$ so $\psi(Y^{\rightarrow n'}) = \psi(Y^{m\rightarrow})$.

Thus $\psi \in \Gamma_{GVG}^*$. For each block m immediately preceding n in F, $\alpha = \psi(X^{\rightarrow n}) = \psi(X^{m\rightarrow})$ so

$$\begin{aligned} \text{origin}(\psi(X^{\rightarrow n})) &= \text{origin}(\psi(X^{m\rightarrow})) \\ &\overset{*}{\rightarrow} \text{loc}(X^{m\rightarrow}), \text{ by Lemma 2.2.2} \\ &= m, \end{aligned}$$

and hence $\text{origin}(\psi(X^{\rightarrow n})) \overset{+}{\rightarrow} n = \text{origin}(\hat{\psi}(X^{\rightarrow n}))$. This implies that $\hat{\psi}$ is not the minimal element of Γ_{GVG}^*, a contradiction.

□

2.3 <u>Propagation of Constants</u>

Let ψ be a minimal element of Γ_{GVG} where GVG is an arbitrary global value graph (V, E, L). We wish to compute a new labeling L' of V such that for each $v \in V$, if $\psi(v)$ is a constant sign then L'(v) = c and otherwise L'(v) = L(v). Nodes thus relabeled with constant signs may be discovered by propagating possible constants through GVG, starting from nodes originally labeled with constant signs, and then testing for conflicts. This leads to an algorithm for constant propagation with time cost linear in the size of the GVG.

Recall that a <u>spanning tree</u> of the control flow graph F = (N, A, s) is a tree rooted at s, with node set N, and edge set contained in A. A <u>preordering</u> of a tree orders fathers before sons. Let $<$ be a preordering of some spanning tree of F. We construct an acyclic subgraph of GVG by deleting value edges which are oriented between nodes in V whose loc values are compatable with $<$. More formally, let $E_<$ be the set of all value edges (v,u) such that loc(v) $<$ loc(v). Observe that $(V, E-E_<)$ is acyclic. We shall propagate constants in a topological order of $(V, E-E_<)$, from leaves to roots.

Our algorithm for computing the new labeling L' is given below.

<u>Algorithm</u> <u>2A</u>.

<u>INPUT</u> GVG = (V, E, L), F.

<u>OUTPUT</u> L'.

```
begin
  declare L' := array of length |V|;
  Let < be a preordering of a spanning tree of F;
  Q := E< := the empty set {};
  for all value edges (v,u) ε E such that loc(v) < loc(u)
      do add (v,u) to E<;
  comment propagate constants;
 L0:for each v ε V in topological order of (V, E-E<)
  from leaves to roots do
    if L(v) is a constant sign c then L1: L'(v) := c;
    else if L(v) is a k-adic function sign θ,
       u1,...,uk are the immediate successors of v in
      GVG, and (θ L'(u1)...L'(uk)) reduces to a
      constant sign c then L2: L'(v) := c;
      else if L(v) is an input variable and there
        is a constant sign c such that L'(u) = c
        for all value edges (v,u) departing from v
        then L3: L'(v) := c;
      else begin add v to Q;L'(v):=L(v) end;
    end;
  comment test for conflicts;
 L4:for each v ε V labeled with an input variable do
    if v has a departing value edge (v,u) such that
    L'(v)≠L'(u) then add v to Q;
  till Q = the empty set {} do
    begin
      delete some node v from Q;
      if L'(v) is a constant sign then
    L5: begin
          L'(v) := L(v);
          add all immediate predecessors of v in GVG to Q;
        end;
    end;
 end.
```

Lemma 2.3.1. If $\psi(v)$ is a constant sign then $L'(v)$ is set to $\psi(v)$ at L1, L2, or L3.

Proof, by induction on the topological order of $(V,E-E_<)$.

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

Induction step. Suppose v is in the interior of $(V, E-E_<)$ and $L'(u)$ has been set to $\psi(u)$ for all u occurring before v in the topological order where $\psi(u)$ is a constant sign. Then v represents either a function application or an input variable.

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

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

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

Lemma 2.3.2. $\psi(v)$ is not a constant sign iff some element of $\tilde{Q}$ is reachable in GVG from v.

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

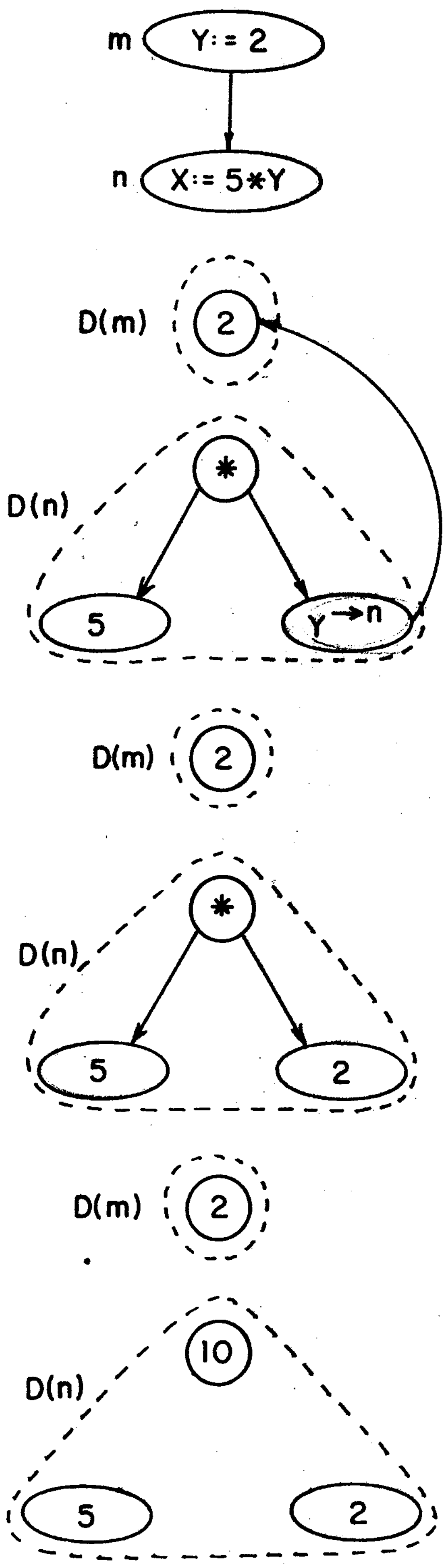
ONLY IF. Suppose some element of $\tilde{Q}$ is reachable from v in GVG. Clearly if $v \in \tilde{Q}$, then $\psi(v)$ is not a constant sign. Assume for some $k > 0$, if there is a path of length less than k in GVG from some $u \in V$ to an element of $\tilde{Q}$, then $\psi(u)$ is not a constant sign.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

Theorem 2.3. Algorithm 2A is correct and has time cost linear in the size of the GVG.

Proof. The correctness of Algorithm 2A follows directly from Lemmas 2.3.1 and 2.3.2.

In addition we must show Algorithm 2A has time cost linear in $|V|+|E|$. The initialization costs time linear in $|V|$. The preordering $<$ may be computed in time linear in $|N|+|A|$ by the depth first search algorithm of [T1]. The time to process each $v \in V$ at steps L0 and L4 is $O(1+\text{outdegree}(v))$. Step L5 can be reached at most $|V|$ times and the time cost to process each node v at step L5 is $O(1+\text{indegree}(v))$. Thus, the total time cost is linear in $|V|+|E|$. □

Figure 2.4. A simple example of constant propagation through the global value graph.



In some cases, we may improve the power of Algorithm 2A for particular interpretations by applying algebraic identities to reduce expressions in EXP more often to constant symbols. For example, in the arithmetic domain we can use the fact that 0 is the identity element under integer multiplication to modify Algorithm 2A so that if node v is labeled by L with the multiplication sign and a successor of v in GVG is covered by 0, then at step L3 we may set L'(v) to the constant sign corresponding to 0.

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

2.4 <u>A Partial Characterization of ψ</u>, <u>the Minimal Element of Γ_{GVG}</u>

Let GVG = (V, E, L) be a reduced global value graph as constructed by Algorithm 2A of the last section and let ψ be the minimal element of Γ_{GVG}. Let $\hat{V}$ be the set of nodes in V representing constant signs and function applications (i.e. nodes labeled with constant and function signs). Observe that Γ_{GVG} characterizes exactly the values of any such ψ over nodes in $\hat{V}$ in terms of the values of ψ over the nodes in $V-\hat{V}$, i.e. in terms of the nodes labeled with input variables. The following Theorem characterizes ψ over $V-\hat{V}$ in terms of ψ over $\hat{V}$.

We require first a few additional definitions. Recall that a <u>value path</u> is a path p in GVG traversing only nodes linked by value edges and is <u>maximal</u> relative to a fixed beginning node if the last node of p has no departing value edges. For any node $v \in V$ labeled with an input variable, let H(v) be the set of nodes in V lying at the end of maximal value paths from v. Note that H(v) is a subset of $\hat{V}$. Call two paths <u>almost disjoint</u> if they have exactly one node in common.

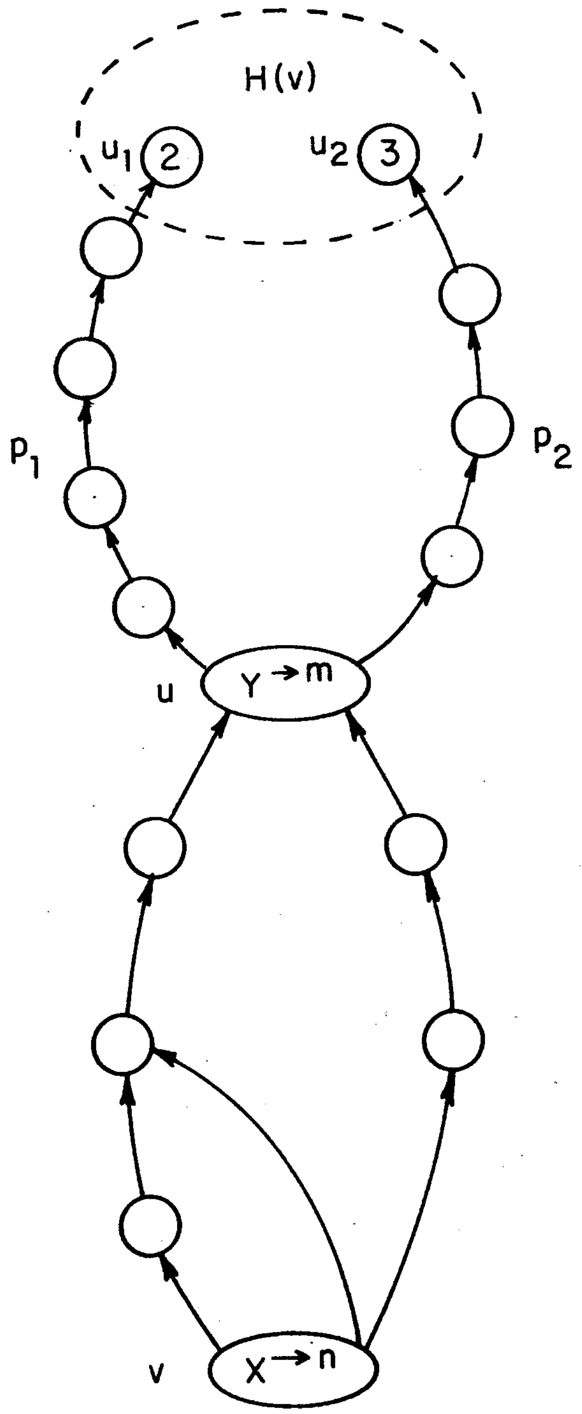


Figure 2.5. Case (b) of Theorem 2.4: all maximal value paths from v contain u and p_1, p_2 are almost disjoint maximal value paths from u to $u_1, u_2 \in H(v)$.

<u>Theorem</u> <u>2.4</u>. If v is labeled with an input variable, then either

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

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

(i) u lies on all maximal value paths from v but

(ii) there are almost disjoint maximal value paths from u to nodes $u_1, u_2 \in H(v)$ such that $\psi(u_1) \neq \psi(u_2)$.

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

Suppose $\psi(v)$ is an input variable, so $\psi(v) = L(u)$ for some $u \in V$. For any maximal value path p from v, let z be the first element of p such that $\psi(z) \neq L(u)$ and let z' be the immediate predecessor of z in p. Then by definition of Γ_{GVG}, $\psi(z') = L(z') = L(u)$ so $z' = u$ is contained on p. Now suppose that there is a node $w \in V$ distinct from u and contained on all maximal value paths from u. Let loc map from nodes in V to the respective blocks in the control from graph where they are located.

Consider any control path q from the start block s to block loc(u). By Lemma 2.2.1, we can construct a maximal value path $(u=w_1,\ldots,w_k)$ such that $loc(w_1),\ldots,loc(w_k)$ are distinct blocks in q. Hence, $loc(w) \overset{+}{\rightarrow} loc(u)$.

Let ψ' be the mapping from V to EXP such that for all $v' \in V$, $\psi'(v')$ is derived from $\psi(v')$ by substituting L(w) for each input variable labeling a node from which all maximal value paths contain w. Then $\psi' \in \Gamma_{GVG}$. But $origin(\psi'(v)) = loc(w) \overset{+}{\rightarrow} loc(u) = origin(\psi(v))$, contradicting our assumption that ψ is minimal over Γ_{GVG}. □

Theorem 2.4 suggests a procedure for calculating ψ, but there is an implicit circularity since the calculation (using Theorem 2.4) of $\psi(v)$ for $v \in V-\hat{V}$ requires the determination (using the definition of Γ_{GVG}) of $\psi(u)$ for $u \in H(v)$; but since $u \in \hat{V}$, the calculation of $\psi(u)$ may require the determination of $\psi(w)$ for some other $w \in V-\hat{V}$. The way out is by the rank decomposition discussed in the next section. There will remain the problem of finding almost disjoint paths, which we consider in Section 2.5. This allows us to apply Theorem 2.4 without circularity.

2.5 Rank Decomposition of a Reduced GVG

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

Fong, Kam, and Ullman[FKU] describe the rank decomposition of a dag; this provides a topological ordering of a dag from leaves to roots over which the dag may be efficiently reduced. Here we generalize the rank decomposition to a possibly cyclic GVG; this provides us a method of partitioning V into sets of text expressions over which ψ may have the same value; it also allows us to apply Theorem 2.4 without circularity, characterizing completely the minimal $\psi \in \Gamma_{GVG}$. In Section 2.7 we apply the rank decomposition to implement our direct method for symbolic evaluation.

The rank of a node $v \in V$ is defined:

rank(v) = 0 if v is labeled with a constant sign
= 1 + MAX{rank(u) | $(v,u) \in E$} for v labeled with a function sign
= MIN{rank(u) | $u \in H(v)$} for v labeled with an input variable.

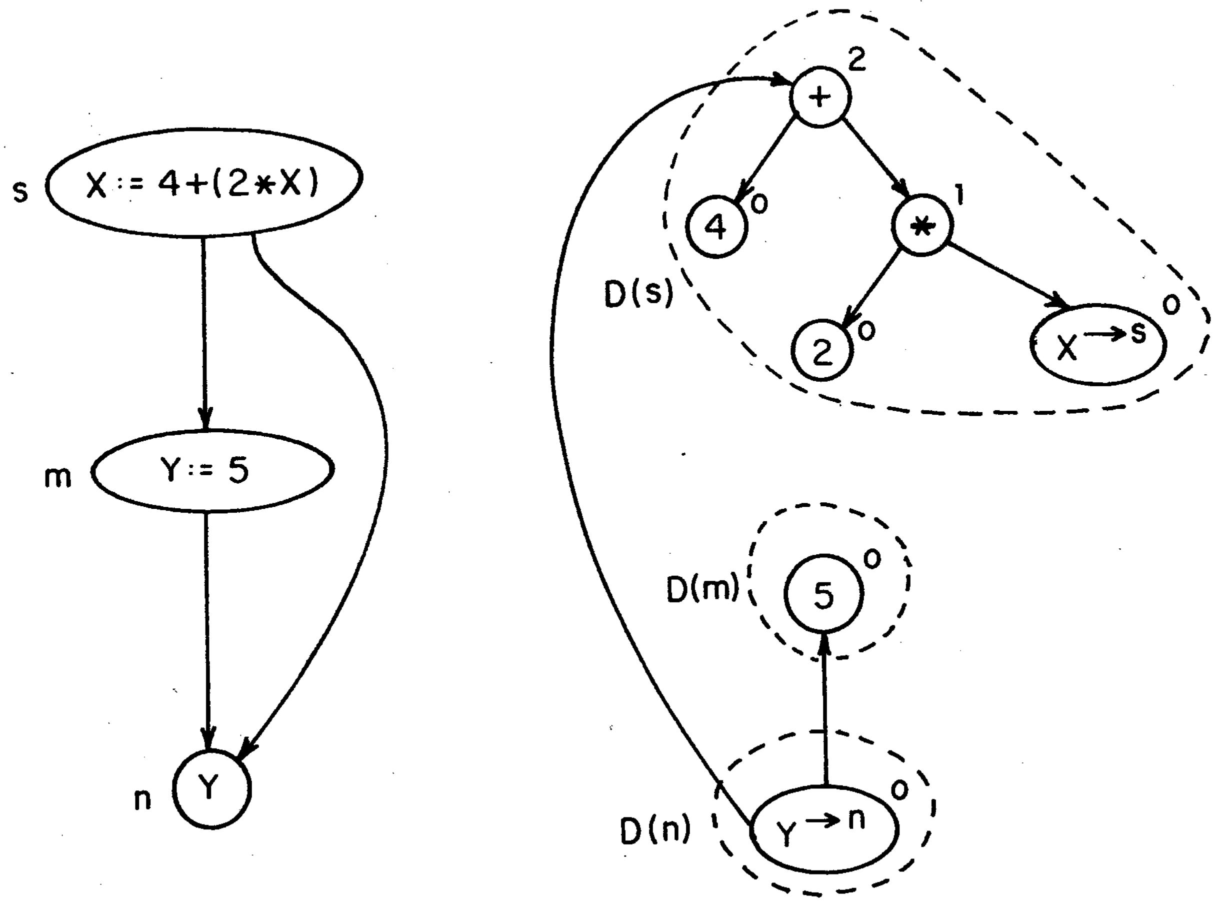


Figure 2.6. Rank decomposition of a global value graph. (The integer on the upper right hand side of each node is its rank.)

Observe that in the very simple case where P contains only a single block of code, (i.e. the start block s) then GVG consists of the dag D(s). Hence the rank of a node v ϵ V is the length of a maximal path from v to a leaf of the dag D(s); inducing a topological ordering of the dag D(s) from leaves to roots.

Lemma 2.5. $\psi(v) = \psi(v')$ implies $\text{rank}(v) = \text{rank}(v')$.

Proof. We proceed by induction on rank of v.

Basis step. Suppose v is of rank 0, so $\psi(v) = \psi(v')$ is a constant sign c. But since GVG is reduced, $L(v') = c$ and v' is also of rank 0.

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

Case a. Suppose $\psi(v) = \psi(v')$ is the function application $(\theta\ \alpha_1 \ldots \alpha_k)$. Then by Theorem 2.4, $\psi(v) = \psi(u)$ for all $u \epsilon H(v)$, and similarily, $\psi(v') = \psi(u')$ for all $u' \epsilon H(v')$. Fix some $u \epsilon H(v)$ and $u' \epsilon H(u')$. By definition of Γ_{GVG}, $L(u) = L(u') = \theta$ and if $w_1,\ldots,w_k$ are the immediate successors of u and $w_1',\ldots,w_k'$ are the immediate successors of u', then $\alpha_i = \psi(w_i) = \psi(w_i')$ for $i = 1,\ldots,k$. By the induction hypothesis, $\text{rank}(w_i) = \text{rank}(w_i')$ for $i = 1,\ldots,k$.

Hence, $\text{rank}(v) = \text{rank}(u)$

$$= 1 + \text{MAX}\{\text{rank}(w_1),\ldots,\text{rank}(w_k)\}$$
$$= 1 + \text{MAX}\{\text{rank}(w_1',\ldots,\text{rank}(w_k')\}$$
$$= \text{rank}(u')$$

= rank (v').

<u>Case b</u>. Suppose $\psi(v) = \psi(v')$ is an input variable. By Theorem 2.4, $\psi(v) = \psi(v') = L(u)$ for some $u \in V$ contained on all value paths from v and v'. Hence, rank(v) = rank(v') = rank(u). □

To compute the rank of all nodes in GVG we use a modified version of the depth first search developed by Tarjan[T1]. Because the search proceeds backwards, we require reverse adjacency lists to store edges in E. Note that the RANK(v) is used in two different ways; first to store the number of successors of node v which have not been visited, and later RANK(v) is set to rank(v). Let V_r, $\hat{V}_r$ be the nodes in V, $\hat{V}$ of rank r. We initially compute $\hat{V}_0$ and on the r'th execution of the main loop we compute $V_r - \hat{V}_r$ and $\hat{V}_{r+1}$.

<u>Algorithm</u> <u>2B</u>.

<u>INPUT</u> GVG = (V, E, L)

<u>OUTPUT</u> RANK

```
begin
  declare RANK:= an array of integers of length |V|;
  for all v ε V do
    RANK(v) := - outdegree(v);
  r := 0;
  Q' := {v |  L(v) is a constant sign };
  untill  Q' = the empty set {} do
    begin
      Q := Q'; Q' := the empty set {};
      comment Q = $\hat{V}_r$;
    L:  untill Q = the empty set {} do
        begin
          delete v from Q;
          for each immediate predecessor u of v do
            if L(v) is a function sign then
              if RANK(u) = -1 then
                begin
                  comment u ε $\hat{V}_{r+1}$;
                  RANK(u) := r+1;
                  add u to Q'
                end
              else RANK(u) := RANK(u) + 1;
            else if RANK(u) < 0 then
              begin
                comment u ε $V_r - \hat{V}_r$;
                RANK(u) := r;
                add u to Q
              end;
        end;
      r := r + 1;
    end;
 end.
```

<u>Theorem</u> <u>2.5</u>. Algorithm 2B is correct and has time cost linear in $|V|+|E|$.

<u>Proof</u> by induction on r.

<u>Basis</u> <u>step</u>. Initially, RANK(v) is set to - (outdegree of v) for each $v \in V$. So if L(v) is labeled with a constant sign then RANK(v) is set to 0. Also, Q is initially set to $\hat{V}_0$ just before label L.

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

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

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

In the inner loop we add to Q exactly the nodes $V_r - \hat{V}_r = \{v \in V-\hat{V} \mid$ some element of $\hat{V}_r$ is reachable by a value path from $v\}$. For each such $v \in V_r - \hat{V}_r$ added to Q, RANK(v) is set to r. Also, for each $v \in \hat{V}$, if rank(v) > r+1 then RANK(v) is incremented by 1 for each immediate successor of v of rank r; if rank(v) = r+1 then all immediate successors of v are of rank $\leq$ r so RANK(v) is set to r+1 and v is added to Q. Thus, (1) and (2) are satisfied entering the loop on the r+1 time.

Now we show that Algorithm 2B may be implemented in linear time. For each node $v \in V$ we keep a list (the reverse adjacency list), giving all predecessors of v. To process any $v \in Q'$ requires time O(1 + indegree(v)). Since each node is added to Q' exactly once, the total time cost is linear in $|V|+|E|$. □

This suffices for the construction of ψ; $\psi(v)$ for $v \in \hat{V}_0$, $V_0-\hat{V}_0$, $\hat{V}_1$, $V_1-\hat{V}_1$,... may be determined by alternately applying the definition of Γ_{GVG} and Theorem 2.4.

Using this method could be inefficient, since Theorem 2.4 could be expensive to apply and the representations of the values could grow rapidly in size. The first problem is solved by reducing it to the problems of P-graph completion and decomposition as described in the next section. The second problem is solved by constructing a special labeled dag; the construction of this dag and the final algorithm are given in Section 2.7.

2.6 P-graph Completion and Decomposition.

Let GVG = (V, E, L) be a reduced global value graph as above. This section presents an efficient method for applying Theorem 2.4 to nodes in $V_r - \hat{V}_r$ (i.e. nodes of rank r labeled with input variables). Now to compute ψ, the minimal element of Γ_{GVG}, it suffices to find the partitioning of V such that $\psi(v) = \psi(u)$ iff v, u are in the same block of the partition. To represent such a partitioning, we distinguish one node of each block of the partitioning to be the value source of all other nodes of that block. We require that if $v \in V - \hat{V}$ (i.e. v is labeled with an input variable) then $\psi(v) = L(v)$ iff v is a value source. Let V^* be the set of value sources and let VS be a mapping from nodes in V to their value sources. Hence the fixed points of VS are the value sources and $VS^{-1}[V^*]$ is a partitioning of V. Note that, in general, the definition of "value source" is not uniquely determined, so the definition of V^* and VS depends on our particular choice of value sources.

We shall find value sources by reducing this problem to the problems of P-graph completion and decomposition stated below.

Let $G = (V_G, E_G)$ be any directed graph and let $S \subseteq V_G$ be a set of vertices of G such that for each vertex $v \in V_G$ there is some vertex $u \in S$ from which v is reachable.

P-Graph Completion Problem. Find

$S^+ = S \cup \{v \in V_G |$ there are almost disjoint paths from distinct elements of S to v not containing any other element of $S\}$.

This form of the problem is due to Karr[K], who shows that it is equivalent to the original formulation due to Shapiro and Saint[SS]. (Actually, this form is slightly more general than Karr's; Karr satisfies our restriction on S by stipulating that there is a single $r \in S$ from which every $v \in V_G$ is reachable.) Karr proves that for each $v \in V_G$ there is one and only one element of S^+ from which v is reachable (and his proof extends directly to our slightly more general problem).

P-Graph Decomposition Problem. Given G and S^+, find, for each $v \in V_G$, the unique $u \in S^+$ from which v is reachable.

We first show these problems can be solved efficiently. Shapiro and Saint give an $O(|V_G|^2)$ algorithm, while Karr gives a more complex $O(|V_G|\log|V_G|+|E_G|)$ algorithm. Here we reduce these problems to the computation of a certain dominator tree, for which there is an almost linear time algorithm as noted in Section 1.2. (This construction was discovered independently by Tarjan[T6].)

Let h be a new node not in V_G, and let G' be the rooted directed graph

$(V_G \cup \{h\},\ E_G \cup \{(h,v)|v\in S\}-\{(u,v)|u\in V_G,\ v\in S\},\ h)$.

Thus G' is derived from G by adding a new root h, linking h to every node in S, and removing the edges of G which lead to nodes in S. Let T be the dominator tree of G'.

Lemma 2.6.1. The members of S^+ are the sons of h in T.

Proof. IF. Let $v \in S^+$. If $v \in S$ then h is a predecessor of v in G' so h is the father of v in T. If $v \in S^+-S$ then by definition of S^+ there are almost disjoint paths $p1$, $p2$ in G from distinct elements of S to v not containing any other element of S. Clearly p_1 and p_2 are also paths in G' since they contain no edge entering a member of S. Then (h,p_1) and (h,p_2) are paths from h to v in G' which have only their endpoints in common, so v is a son of h in T.

ONLY IF. Suppose v is a son of h in T. If h is a predecessor of v in G' then $v \in S \subseteq S^+$. Otherwise there are in G' paths (h,p_1) and (h,p_2) from h to v which have only their endpoints in common. Moreover these paths contain no element of S except for the first nodes of p_1, p_2, since no edge of G' enters an element of S except from h. Hence p_1, p_2 are almost disjoint paths in G' from distinct members of S to v not containing any other element of S, and hence $v \in S^+$. □

Theorem 2.6.1. For each $v \in V_G$, the unique node in S^+ from which v is reachable in G is the unique node which is a son of h and an ancestor of v in T.

Proof. Let w be that ancestor of v which is a son of h in T. By Lemma 2.6.1, $w \in S^+$, and clearly v is reachable from w in G since it is reachable from w in T. Conversely, if w ϵ S^+ is reachable from v in G then w is a son of h in T by Lemma 2.6.1, and w must be an ancestor of v since otherwise v would be reachable from some other member of S^+. □

Now we establish the relation of these problems to the problem of finding V^* and VS as stated above. Fix some V^* and VS by choosing one node of GVG for each value of ψ on V consistent with our definition of value sources. For each rank r, let $G_r = (V_r, E_r)$, where V_r is the set of all nodes of a reduced GVG of rank r as defined in Section 2.5 and E_r is the edge set derived from E by

(1) deleting all edges except value edges between nodes of rank r,

(2) for those remaining value edges (v,u) entering u ϵ $\hat{V}_r$, substituting instead the edge (v,VS(u)),

(3) finally reversing all edges.

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

<u>Lemma 2.6.2</u>. The members of S_r^+ are the value sources of

rank r.

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

Case 1. By definition, all elements of $\{VS(v) \mid v \in \hat{V}_r\}$ are value sources. Hence we need only consider the case where v is a node of rank r labeled with an input variable which has a departing value edge (v,z) entering a node z of rank greater than r. Since v is of rank r, v must also have a departing value edge (v,u) leading to a node of rank r. By Lemma 2.5, $\psi(z) \neq \psi(u)$, so by the definition of Γ_{GVG}, $\psi(v) = L(v)$ and v is a value source.

Case 2. Suppose there are in G_r almost disjoint paths $(x_1,x_2,\ldots,x_j)$ and $(y_1,y_2,\ldots,y_k)$ in G_r from distinct x_1, $y_1 \in S_r$ to v. By construction of G_r, there exist distinct $\overline{x}_1,\overline{y}_1 \in H(v)$ such that $VS(\overline{x}_1) = x_1$, $VS(\overline{y}_1) = y_1$, and $(x_2,\overline{x}_1)$ and $(y_2,\overline{y}_1)$ are value edges, and so $p_1 = (v{=}x_j,x_{j-1},\ldots,x_2,\overline{x}_1)$ and $p_2 = (v{=}y_k,y_{k-1},\ldots,y_2,\overline{y}_1)$ are almost disjoint value paths. Now suppose v is not a value source. Applying Theorem 2.4, there is a value source u (distinct from v) such that $\psi(v) = \psi(u) = L(u)$. Since p_1 and p_2 are almost disjoint they can not both contain u. Suppose, without loss of generality, that p_1 avoids u. Then all maximal value paths from $\overline{x}_1$ contain u. Also, by definition of S_r, $\overline{x}_1 = x_1$ and there is a value edge (v,z) such that z is not of rank r. Since any maximal value path from z must contain u, rank(z) = rank(u) implying that u is not of rank r. But, by hypothesis, all maximal value paths from v contain u, so rank(v) = rank(u). This implies that v is not of rank r, contradicting our assumptions. □.

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

Theorem 2.6.2. For all nodes $v \in V$ of rank r and labeled with an input variable, VS(v) is the unique value source contained on all paths in G_r from elements of S_r to v.

Thus the problem of computing VS reduces to the problem of decomposing the reduced global value graph by rank and then constructing dominator trees. The former can be done in linear time by Algorithm 2B of Section 2.5, the latter in almost linear time by Tarjan's Algorithm[T4].

2.7 The Algorithm for Symbolic Evaluation.

In this section we pull together the various pieces developed in Sections 2.3-2.6 to give a unified presentation of our algorithm for symbolic evaluation. Instead of using the GVG directly to represent ψ, as suggested in the beginning of Section 2.6, we more economically represent ψ by a dag D^* derived from GVG by collapsing nodes into their value sources; more precisely $D^* = (V^*, E^*, L^*)$ where

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

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

L^* is the restriction of L to V^*.

Recall from Section 2.2 that rooted dags may be used to represent expressions in EXP.

Lemma 2.7. for each node $v \in V$,

$(D^*, VS(v))$ represents $\psi(v)$.

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

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

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

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

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

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

Induction step. Suppose v is in the interior of D^* and (D^*,u) represents $\psi(u)$ for all sons u of v. Thus v must be labeled in L with a function sign θ and have immediate successors $u_1,\ldots,u_2$ in GVG. Then $VS(u_1),\ldots,VS(u_k)$ are the sons of v in D^* and for $i = 1,\ldots,k$ by the induction hypothesis $(D^*,VS(u_i))$ represents $\psi(VS(u_i))$

$= \psi(u_i)$.

Thus (D^*,v) represents $(\theta\ \psi(u_1)\ldots\psi(u_k))$

$= \psi(v)$ by definition of Γ_{GVG}.

□

Our algorithm for symbolic evaluation is given below. As in Section 2.6, we compute ψ and VS in the order of the rank of nodes in V. The array COLOR is used to discover nodes with the same ψ.

Algorithm 2C.

INPUT GVG = (V, E, L)

OUTPUT VS and $D^* = (V^*,E^*,L^*)$.

```
begin
initialize:
  declare VS,COLOR := arrays of length |V|;
  procedure COLLAPSE(S,u):
    for all v ε S do
      begin
        VS(v) := u;
        if u≠ v then
          begin
            for each edge (w,v) entering v do
              substitute (w,u);
            for each edge (v,w) departing from v do
              substitute (u,w);
            delete v from the edge set;
          end;
      end;
  Compute new labeling $L'$ of $V$ by Algorithm 2A
    and reduce GVG as described in Section 2.3;
  Compute rank of nodes in $V$ by Algorithm 2B
    of Section 2.4;
  for r := 0 to MAX{rank(v) | v $\epsilon$ V} do
    begin
      Let $V_r$, $\hat{V}_r$ be the nodes in $V$, $\hat{V}$ of rank r;
      for all v $\epsilon$ $\hat{V}_r$ do
        if r = 0 then COLOR(v) := L'(v)
        else COLOR(v) := <L(v),$u_1$,...,$u_k$> where
        $u_1$,...,$u_k$ are the current immediate
        successors of v;
      radix sort nodes in $\hat{V}_r$ by their COLOR;
      for  each maximal set S $\subseteq$ $\hat{V}_r$ containing nodes
      with the same COLOR do
        begin
          choose some u $\epsilon$ S;
          comment u is made a value source;
          COLLAPSE(S,u);
        end;
      Let h be some node not in $V_r$;
      $E_r$ := $S_r$ := the empty set {};
      for all v $\epsilon$ $\hat{V}_r$ do add VS(v) to $S_r$;
      for all v $\epsilon$ $V_r$-$\hat{V}_r$ do
        for each node u which is currently an
        immediate successor of v do
          if u is of rank r then
            add (u,v) to $E_r$;
          else add u to $S_r$;
      Let $T_r$ be the dominator tree of $G_r$ =
      ($V_r$ $\cup$ {h}, $E_r$ $\cup$ {(h,v)|v $\epsilon$ $S_r$}, h);
      for all sons u of h in $T_r$ do
        begin
          comment By Theorem 2.6.1 and Lemma 2.6.2,
            u is a value source;
          COLLAPSE({the descendants of u in $T_r$},u);
          delete all edges departing from u;
        end;
    end;
  Let $V^*$, $E^*$ be the node set and edge list derived from V, E
    by the above collapses;
  for all v $\epsilon$ $V^*$ do $L^*$(v) := L'(v);
end.
```

Theorem 2.7. Algorithm 2C is correct and can be implemented in almost linear time.

Proof. The correctness of Algorithm 2C follows directly from Theorems 2.6.1, 2.6.2 and Lemmas 2.6.2, 2.7.

In addition, we must show that Algorithm 2C can be implemented in almost linear time. The storage cost of GVG is linear in $|V|+|E|$. The initialization of Algorithm 2C costs time linear in $|N| + |A|$. Algorithms 2A and 2B cost linear time by Theorems 2.3 and 2.5, respectively. The time cost of the r'th execution of the main loop, exclusive of the computation of T_r, is linear in $|V_r| + |E_r|$, plus the sum of the outdegree of all $v \in V_r - \hat{V}_r$. (Here we assume that elements in the range of L' are representable in a fixed number of machine words and that the number of argument-places of function signs is bounded by a fixed constant, so a radix sort can be used to partition $\hat{V}_r$ by COLOR.) The computation of the dominator tree T_r requires by [T4] time cost almost linear in $|V_r| + |E_r|$. Thus, the total time cost is almost linear in $|V| + |E|$. □

This completes the presentation of our algorithm. The next section explains how with the aid of the preprocessing stage of Chapter 4 costing $O((|A|+\ell)\alpha(|A|+\ell))$ bit vector operations, we may construct a global value graph GVG^+ of size $O(d|A|+\ell)$ where d is often of order 1 for block-structured programs but may grow to $|\Sigma|$. (Thus this preprocessing stage offers no theoretical advantage but in practice often leads to a glbal value graph of size linear in the program and flow graph.) GVG^+ has the property that the minimal element of ΓGVG^+ is the minimal fixed point of the functional Ψ defined in Section 2.1. In contrast to Kildall's iterative method, which for a large class of programs has storage cost $\Omega(\ell|N|)$ and time cost $\Omega(\ell|N|^2)$, our direct method has storage cost linear in the size of GVG^+ and time cost almost linear in the size of GVG^+. Although either method may be improved somewhat through the use of domain-specific identities, as shown in Section 1.4, there is in general no algorithm for computing an optimal symbolic evaluation.

In Chapter 3 these methods are extended to programs which operate on structured data in a language such as PASCAL or LISP 1.0.

2.8 Improving the Efficiency of our Algorithm for Symbolic Evaluation

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

The methods of Sections 2.3-2.7 allow us to construct, for any global value graph GVG, the unique minimal element of Γ_{GVG} in space linear in the size of GVG and time almost linear in the size of GVG. Section 2.2 defines a global value graph GVG^* of size $O(|\Sigma||A|+\ell)$ and with the property that ψ^* is the minimal element of Γ_{GVG^*}. Now we shall define a global value graph GVG^+ of size often considerably less but with the property that a restriction of ψ^* is the minimal element of Γ_{GVG^+}.

A path is <u>m-avoiding</u> if the path does not contain node m. Consider blocks m, n in the control flow graph such that m dominates n. A program variable $X \in \Sigma$ is <u>definition-free</u>

between m and n, if (1) $m = n$ or (2) $m \overset{+}{\rightarrow} n$ and X is not assigned to on any m-avoiding control path from an immediate successor of m to an immediate predecessor of n (otherwise X is defined between m and n). Let W be a function from text expressions which are input variables to blocks of the control flow graph; for each input variable $X^{\rightarrow n}$ which is an text expression, $W(X^{\rightarrow n}) = m$, where m is the first block on the dominator chain of the control flow graph from the start block s to n such that X is definition-free between m and n. An algorithm in Chapter 4 computes W in a number of bit vector steps almost linear in $|N|+\ell$.

It will be convenient to assume that for each text expression which is an input variable $X^{\rightarrow n}$ such that $W(X^{\rightarrow n}) = n$, X is assigned to at each block m immediately preceding n. We must add $O(d|N|)$ dummy assignments to accomplish this; d is often constant for block structured programs but may grow to $|\Sigma|$. Let VE be the set of pairs of text expressions (t,t') such that

(1) t is an input variable $X^{\rightarrow n}$

(2) t' is an output expression $X^{m\rightarrow}$

(3) either (a) $W(X^{\rightarrow n}) = n$ and m is an immediate predecessor of n in F, or (b) $W(X^{\rightarrow n}) = m \overset{+}{\rightarrow} n$.

Note that VE contains $O(d|A|+\ell)$ edges. Let GVG^{+} be the global value graph with value edges VE. The nodes in GVG^{+} will be identified with the text expression which they represent. Let $d = |VE|/|A|$ and observe that $d \leq |\Sigma|$. Then GVG+ is of size $O(|VE|+\ell) = O(d|A|+\ell)$.

Let ψ^+ be the minimal element of Γ_{GVG+} and let ψ^* be the minimal fixed point of Ψ.

<u>Theorem 2.8</u>. $\psi^+ = \overline{\Psi}^*$, where $\overline{\Psi}^*$ is the restriction of ψ^* to the domain of ψ^+.

<u>Proof</u> Suppose $\overline{\Psi}^* \notin \Gamma_{GVG+}$, so there must be an input variable $X^{\to n}$ such that $\psi^*(X^{\to n}) \neq X^{\to n}$ and $\psi^*(X^{\to n}) \neq \psi(t)$ for some value edge $(X^{\to n},t) \in VE$. Then $n' = W(X^{\to n}) \overset{+}{\to} n$ and furthermore, $n' \overset{+}{\to}$ birthpoint$(\psi(X^{\to n}))$. Let ψ be the mapping from text expressions to EXP such that for each text expression t, $\psi(t)$ is derived from $\psi(t)$ by substituting $X^{n'\to}$ for each input variable $X^{\to m}$ such that $W(X^{\to m}) = n'$. Thus ψ is an element of Γ_{GVG+}, a contradiction with the assumption that ψ^* is the minimal element of Γ_{GVG+}.

Let $\overline{\Psi}^+$ be the extension of ψ^+ to the domain of ψ^* defined thus: for each input variable $X^{\to n}$ not in the domain of ψ^+, let $\overline{\Psi}^+(X^{\to n}) = \psi^+(X^{m\to})$ where $m = W(X^{\to n})$. We claim $\overline{\Psi}^+ \in \Gamma_{GVG+}$. Suppose $\overline{\Psi}^+ \notin \Gamma_{GVG+}$, so there is an input variable $X^{\to n}$ such that $\overline{\Psi}^+(X^{\to n}) \neq X^{\to n}$ and $\overline{\Psi}^+(X^{\to n}) \neq \overline{\Psi}^+(X^{m\to})$ for some m immediately preceeding n in the control flow graph F. Hence, $\overline{\Psi}^+(X^{\to n}) = \overline{\Psi}^+(X^{n'\to})$ where $n' = W(X^{\to n}) \overset{+}{\to} n$. But X is definition-free between n' and n, hence $(X^{\to m},X^{\to n'})$ is the unique value edge in VE departing from $X^{\to m}$. Let ψ be the mapping from text expressions to EXP such that for each text xpression t, $\psi(t)$ is the reduced expression derived from $\psi(t)$ by substituting $X^{n'\rightarrow}$ for each input variable $X^{\rightarrow\overline{n}}$ such hat $W(X^{\rightarrow\overline{n}}) = n'$. Then $\psi \in \Gamma_{GVG+}$, implying that $\psi+$ is not he minimal element of Γ_{GVG+}, a contradiction. $\Box$

2.9 Further Applications of Global Value Graphs: Live-Dead Analysis

A concept related to the value edges of VE defined in Section 2.8 is due to Schwartz[Sw2]: a pair of text expressions (t,t') is a use-definition link if t is an input variable $X^{\rightarrow n}$, t' is an output variable $X^{m\rightarrow}$, and X is not defined on some (rather than all) m-avoiding control path from an immediate successor of m to an immediate predecessor of n. Unfortunately there may be $\Omega(\ell^2)$ use-definition pairs whereas there are at most $O(|\Sigma||A|+\ell)$ value edges in VE.

A method for live-dead analysis has been described by Schwartz[Sc2]; his method uses use-definition links and would require $\Omega(\ell^2)$ operations even if implemented using efficient depth first search techniques. The global value graph GVG^+ with value edges VE may be also applied to live-dead analysis thus: We distinguish a set $\mathbf{V} \subseteq V$ which are text expressions corresponding to vital computations (for example all text expressions appearing in print statements). Then text expression t is live if t is involved in the computation of EXEC(t',p) for some vital text expression $t' \in \mathbf{V}$ and control path p from s to the block where t is located (otherwise t is dead). Deleting all dead text expressions would thus not interfere with the vital computations of the program. It follows that text expression t is live iff some element of $\mathbf{V}$ is reachable from

t. Live text expressions may thus be discovered in time linear in the size of GVG^+ by a reverse depth first search backwards from elements of $\mathbf{V}$. All text expressions _not_ reached are dead, and may be deleted.

CHAPTER 3

SYMBOLIC ANALYSIS OF PROGRAMS WITH STRUCTURED DATA

3.0 Summary

We discuss the symbolic analysis of a class of programs such as those of LISP 1.0, which have a fixed interpretation for various operations on structured data including: operations for construction of structured objects (such as _cons_ in LISP) and the selection of subcomponents (such as _car_ and _cdr_ in LISP), but no "destructive" operations (such as _replaca_ or _replacd_ in LISP 1.5). We continue to use the global flow model of Chapter 1, in which assignment statements are the only variety of statements and the program flow graph represents the flow of control.

A central problem here is the _propagation of selections_: the determination of the set SP of ordered pairs of selection operations and the objects which they may reference. The elements of SP are called _selection pairs_. We show that this propagation problem is at least as hard as transitive closure of a binary relation and we give an efficient algorithm, using bit vector operations, for computing SP. Schwartz[Sc2] requires the set SP for his method for the automatic construction of recursive type declarations, though he gave no explicit algorithm for propagating selections.

We consider further applications of propagation of selections including: the determination of selection operations that, when executed, always result in an error (i.e., they attempt to access non-existant subcomponents), the propagation of constants, and more generally the determination of covers (symbolic representations of values of text holding for all executions of the program). The methods of Chapter 2 for the determination of covers are improved so as to take into account reductions due to the selection of subcomponents of structured objects.

We apply these improved methods also to the construction of <u>type covers</u> which are representations of types (rather than values) of text expressions and hold for all executions of the program. Type covers are useful for the discovery of construction operations which are redundant in the sense that they have values of the same <u>type</u> as values previously computed but are now dead (no longer referenced).

Finally, we discuss Schwartz's method of recursive type determination.

3.1 Introduction

This Chapter is concerned with the analysis of programs which manipulate structured data; for example:

the lists of LISP

the strings of SNOWBALL

and the arrays in FORTRAN, ALGOL, and PL/1.

Though we allow general operations for construction of structured objects and selection of subcomponents, our analysis is restricted to programs with no <u>destructive operations</u>: they must not modify subcomponents (i.e., install new sublists, insert or delete new characters of strings, or modify elements of arrays). Hence our methods are only applicable to a restricted subclass of the above programming languages with list, string, and array data structures. We believe our methods can be extended (with a certain increase in time and space cost) to programs which allow modification of subcomponents. (At any rate, there exist certain simple programing languages, such as LISP 1.0, which do <u>not</u> allow modification of subcomponents.)

As in the preceding Chapter, we discuss the analysis of a program P relative to a <u>global flow model</u> in which the flow of control is represented by the <u>control flow graph</u> F = (N, A, s) where the nodes of N correspond to contiguous sequences of assignment statements called <u>blocks</u>, the edges in A specify possible flow of control between blocks in N, and all control flow begins at the start block $s \in N$.

Let $\Sigma = \{X,Y,Z,...\}$ be the non-local program variables of P. For each $X \in \Sigma$ and block $n \in N-\{s\}$, we introduce the input variable $X^{\to n}$ denoting the value of X on input to block n. Also, for each $X \in \Sigma$, $X^{\to s}$ is a distinct constant sign denoting the value of X on input to the program P at the start block s. As in Chapter 1, we require a first order language without predicates to represent computations of P and their covers. Let EXP be the set of expressions built from input variables, and fixed sets of constant signs C and k-adic function signs Θ; here Θ is partitioned into the sets:

(1) OP, a set of operator signs used for elementary operations on atomic values,

(2) CONS, a set of constructor signs used to build up structured values,

(3) SEL, a set of 1-adic selector signs used to select subcomponents of structured values.

A function application is an expression of the form

$$\alpha = (\theta\ \alpha_1,...,\alpha_k),$$

where θ is a k-adic function sign in Θ and $\alpha_1,..,\alpha_k \in$ EXP. α is an elementary operation if θ is an operator sign op $\in$ OP, α is a construction operation if θ is a constructor sign cons $\in$ CONS, and α is a selection operation if $k = 1$ and θ is a selector sign sel $\in$ SEL. For each k-adic constructor sign cons ϵ CONS and i, $1 \leq i \leq k$, there exists an unique selector sign sel ϵ SEL called the <u>i^{th} selector of cons</u>.

As described in the examples below, in LISP there is a simple constructor (<u>cons</u>), two selectors (<u>car</u> and <u>cdr</u>), and elementary operations depending on the particular version of the language (none in LISP 1.0, arithmetic and logical operations in LISP 1.5).

SELECT(sel,α) gives the result of selection by a selector sign sel ϵ SEL on expression $\alpha \epsilon$ EXP:

(1) if α is a construction operation

$$(\text{cons } \alpha_1 \ldots \alpha_k)$$

where sel is the ith selector of constructor sign cons, then SELECT(sel,α) = α_i.

(2) If α is a construction operation for which sel is <u>not</u> a selector, or α is a constant sign, or α is an elementary operation then SELECT(sel,α) = <u>error</u>, where <u>error</u> is a distinguished constant sign in C denoting an error condition.

(3) In all other cases (e.g., where α is itself a selection or an input variable) SELECT(sel,α) is left undefined.

For example, in LISP,

$$\text{SELECT(cdr, (cons } \alpha_1\ \alpha_2)) = \alpha_2.$$

We assume an <u>interpretation</u> (U, I) such that

(1) U is an <u>universe</u> of values consisting of

(a) ATOM, a set of atomic values (atoms), and

(b) structured values constructed by prefixing k-adic constructor signs in CONS to k-tuples in the universe U.

(2) I is a homomorphic mapping from EXP to U such that

(a) For each constant sign c ϵ C, I(c) ϵ ATOM. We assume the constant signs in C are in one-to-one correspondence to atoms in ATOM. The distinguished constant sign error is also an atom and is freely interpreted: I(error) = error.

(b) For each k-adic function sign θ ϵ $\boldsymbol{\theta}$, I(θ) is a partial mapping from U^k to U.

(i) Each k-adic operator sign op ϵ OP is interpreted as a mapping I(op) from k-tuples of atoms into individual atoms (note that such mappings may not take structured objects as arguments).

(ii) k-adic constructor signs cons ϵ CONS are freely interpreted:

$$I(cons)(z_1,\ldots,z_k) = (cons\ z_1,\ldots,z_k)$$

for all $z_1,\ldots,z_k$ ϵ U.

(iii) Each selector sign sel ϵ SEL is interpreted to map from expressions in the universe U to their corresponding subexpressions, or where this is not possible, to error. More formally, for each α ϵ EXP such that SELECT(sel,α) is defined:

$$I(sel)(I(\alpha)) = I(SELECT(sel,\alpha)).$$

Example 3A (LISP 1.0)

ATOM = {the empty list nil}

OP = the empty set {}

CONS = {the list constructor cons}

SEL = {car, the first selector of cons}

$\cup$ {cdr, the second selector of cons}

Example 3B

(similar to LISP 1.5 but without replaca and replacd)

ATOM = {the empty list nil}

$\cup$ {the integers}

$\cup$ {the boolean truth values truth and false}

OP = {and, or, plus, minus, mult, and div}

and and or are interpreted as logical conjuncton and disjunction; the other operator signs in OP are interpreted as the usual arithmetic operations. CONS, SEL are as in Example 3A.

Example 3C (Vectors of fixed length)

ATOM = $\{a_1, a_2, \ldots\}$

SEL = {the positive integers}

CONS = $\{\text{vector}^1, \text{vector}^2, \ldots\}$

where vector^k is a k-adic constructor sign and the integer i is the i^{th} selector of vector^k for each $1 \leq i \leq k$. Note that the number of function places of each vector^k is fixed; so it is not possible to construct variable length sequences. However we can easily extend the model to allow function signs with a variable number of arguments.

In Chapter 1 we defined a <u>constant reduction</u> on an expression in EXP to be the repeated substitution of constant signs for constant subexpressions (relative to a fixed interpretation); that is if $\alpha \in$ EXP contains a elementary operation

$$\alpha' = (\text{op } c_1 \ldots \; c_k)$$

where op $\in$ OP and $c_1,\ldots,c_k \in$ C and there exists a constant sign c $\in$ C such that

$$I(c) = I(op)(I(c_1),\ldots,I(c_k)),$$

then we substitute c in the place of α'.

In addition, we define <u>selection reductions</u> to be the result of substituting SELECT(sel,α'), where it is defined, for each selection operation (sel α').

An expression is <u>reduced</u> by repeated constant and selection reductions.

For each program variable X $\in \Sigma$ defined (i.e., assigned to) at block n $\in$ N, the <u>output variable</u> $X^{n\rightarrow}$ is a reduced expression in EXP for the value of X on <u>exit</u> from block n in terms of the constants and input variables at n. For example, $Y^{n\rightarrow}$ = (cons (car $X^{\rightarrow n}$) $Y^{\rightarrow n}$) in the program of Figure 3.1.

The <u>text expressions</u> of P are the output variables and their subexpressions. We assume the text expressions are reduced expressions. For each reduced expression $\alpha \in$ EXP and control path p, EXEC(α,p) is intuitively a reduced expression in EXP for the value of α relative to p. For a more precise definition, see Section 1.3.

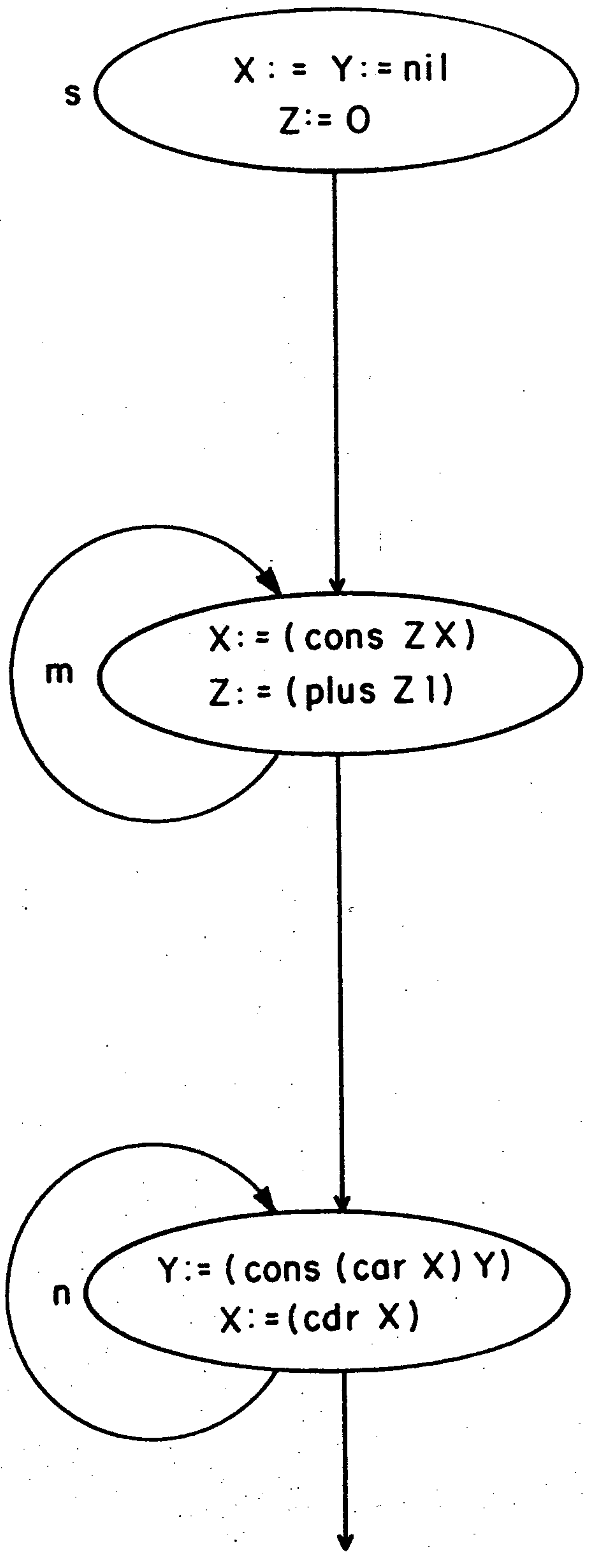


Figure 3.1. Reversal of a list in LISP.

3.2 Propagation of Selections

Our immediate goal is to determine all "selection pairs". Loosely speaking, these are pairs (w,u) where w is a selection operation (sel t) and u is a text expression whose value, relative to some execution of the program, may be obtained from the value of t by the use of the selector sign sel. More precisely, for text expressions t and t', let t' be accessible from t if EXEC(t,p) = t' for some control path p from loc(t') to loc(t). Note that selection sequences are generalizations of the value paths of Chapter 2. A selection pair is an ordered pair of text expressions (w,u) consisting of a selection w = (sel t) and u = SELECT(sel,t'), where SELECT(sel,t') is defined for some text expression t' accessible from t. We assume that the constant sign error is a text expression located at the start block, so t has a departing selection pair (t,error) if EXEC(t,p) = error for some control path p from s to loc(t). We also assume there are no selections at the start block s, so each selection in the text has at least one departing selection pair.

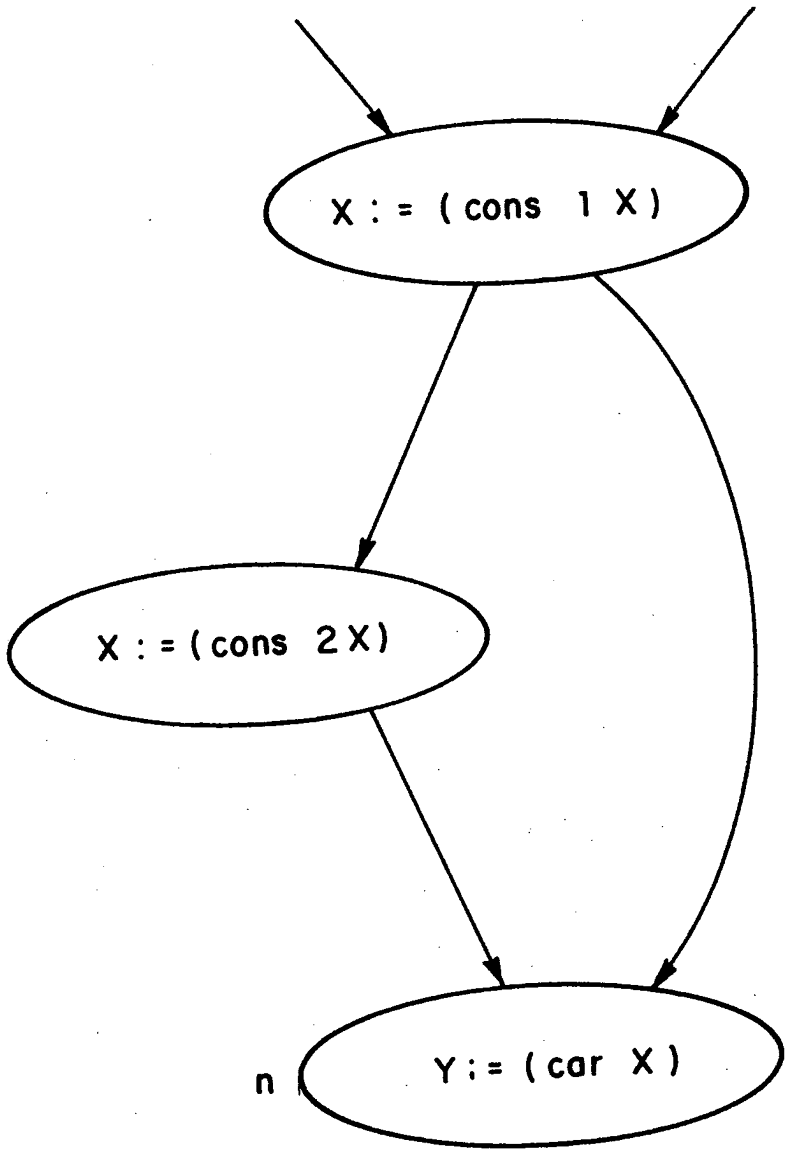


Figure 3.2. (Y_n^+,1) and (Y_n^+,2) are selection pairs.

Selection propagation is the task of discovering all selection pairs.

Theorem 3.2.1 Selection propagation in the interpretation of Example 3A (LISP 1.0) is at least as hard as computing the transitive closure of a binary relation.

Proof Let R be a binary relation on $\{n_1,\ldots,n_r\}$ and let $R^* = \{(n_{i_1},n_{i_k}) \mid (n_{i_1},n_{i_2}),\ldots,(n_{i_{k-1}},n_{i_k}) \in R,\ k \geq 1\}$ be the reflexive transitive closure of R. Consider the control flow graph F_R = (N, A, s) of Figure 3.2 where

$N = \{s=n_0,n_1,\ldots,n_{3r}\}$

and the edge set A consists of

(1) R and

(2) for $i=1,\ldots,r$ edges (n_0,n_{r+i}), (n_{r+i},n_i), and (n_i,n_{2r+i}).

Let the text of n_0 be empty.

For $i = 1,\ldots,r$

(1) the text of n_i is empty

(2) the text of n_{r+i} is X := (cons nil X).

(3) the text of n_{2r+i} is the selection X := (cdr X).

It follows that $(n_i,n_j) \in R^*$

iff there is a value path from $X^{\to n_{2r+j}}$ to $X^{n_{r+i}\to}$

iff $(X^{n_{2r+j}\to},X^{\to n_{r+i}})$ is a selection pair. □

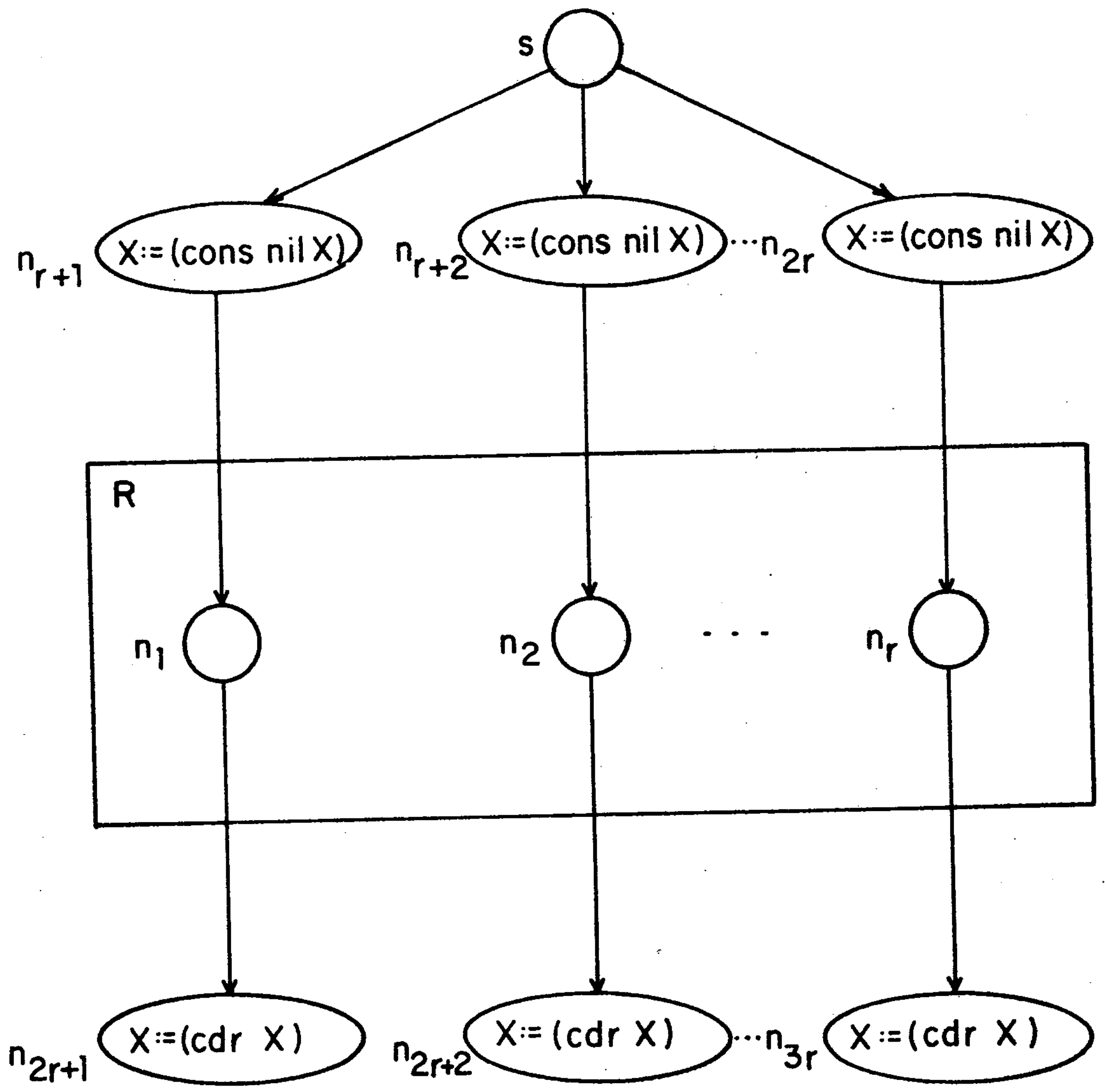


Figure 3.3. The control flow graph FR.

As in Chapter 2, we use a dag D(n) (an acyclic, oriented digraph) to represent computations local to a linear block of code $n \in N$. Each node of D(n) represents an unique text expression located at block n. A global value graph GVG = (V, E, L) is a possibly cyclic, oriented digraph consisting of

(1) the dags of all the blocks in N, and

(2) a set of edges, called value edges of GVG, departing from nodes labeled with input variables. For each node $v \in V$ labeled with an input variable and control path p from the start block s to loc(v), there is a value edge (v,u) such that loc(u) is distinct from loc(v) and is contained in p. (the labeling L is consistent with that of the dags.)

A value path of GVG is a path transversing only nodes linked by value edges.

In Section 2.1 we defined a special global value graph $GVG^* = (V, E, L)$ with value edges defined so as to properly represent the flow of values of program variables between blocks of code; that is (1) the nodes in GVG^* are identified with the text expressions which they represent and (2) (t,t') is a value edge of GVG^* iff t is an input variable $X^{\rightarrow n}$ and t' is the output variable $X^{m\rightarrow}$ for some $(m,n) \in A$. This definition requires that the text expressions include all the input variables; for each input variable $X^{\rightarrow n}$ not originally a text expression at block $n \in N-\{s\}$, add a "dummy" assignment of the form

X := X.

Let V be the set of the resulting new text expressions corresponding to these dummy assignments.

An <u>access sequence</u> is a sequence of text expressions $(t_1,\ldots,t_k)$ such that for $1 \leq i < k$, each (t_i,t_{i+1}) is either a value edge of GVG^* or a selection pair.

<u>Theorem 3.2.2</u> For all $t,t' \in V$, there is an access sequence from t to t' iff t' is accessible from t.

<u>Proof</u> Suppose there exists an access sequence $(t=t_1,\ldots,t_k=t')$. Then for $i = 1,\ldots,k-1$ whether (t_i,t_{i+1}) is a value edge or a selection pair, there is always a control path p_i from $loc(t_{i+1})$ to $loc(t_i)$ such that

$$t_{i+1} = EXEC(t_i,p_i).$$

Hence $t' = EXEC(t,p_{k-1} \cdot p_{k-2} \cdot \ldots \cdot p_1)$ so t' is accessible from t.

On the other hand, suppose there is a control path p of minimal length such that there exists text expressions t,t' such that p begins at loc(t') and ends at loc(t) and

$$t' = EXEC(t,p)$$

but there is no access sequence from t to t'. If t is an input variable, t has a departing value edge (t,τ) such that $loc(\tau)$ is distinct from loc(t) and $loc(\tau)$ is contained in p. If t is a selection, then there is a departing selection pair (t,τ) where $loc(\tau)$ is contained in p. In either case if $p = p_1 \cdot p_2$ where p_2 is a subsequence of p from $loc(\tau)$ to loc(t), then by the induction hypothesis

$$t' = \mathrm{EXEC}(\tau, p_1)$$

so t' is accessible from τ and by the induction hypothesis there is an access sequence q from τ to t'. Hence, $(t,\tau)\cdot q$ is an access sequence from t to t', a contradiction. □

We now present an efficient algorithm for the discovery of all selection pairs.

<u>Algorithm 3A</u>

<u>INPUT</u> $GVG^* = (V, E, L)$ and $\overline{V}$, the set of added text expressions corresponding to dummy assignments.

<u>OUTPUT</u> SP, the set of selection pairs of P.

```
begin
  declare for each $t \in V$
    $VP_t, AS_t, \overline{AS}_t$ := sets of maximum size $|V|$ each
      represented as bit vectors of length $|V|$
      ($3|V|$ sets, initially all empty);
  procedure PROPAGATE(t,t'):
    begin
      add t' to $AS_t$;
      add t to $\overline{AS}_{t'}$;
      add (t,t') to Q;
    end;
  Q := the empty set {};
  Let VE be the edges in E departing from nodes labeled
    with input variables;
  Compute the transitive closure $VE^*$ of VE;
  $VE^*$ is represented by a family of sets $\{VP_t | t \in V\}$
    where for $t,t' \in V$, $t' \in VP_t$ iff there exists
    a value path in $GVG^*$ from t to t';
  for all $t \in V$ do
    for all $t' \in VP_t$ do
      if $t,t' \in V-\overline{V}$ then L0: PROPAGATE(t,t');
  untill Q = the empty set {} do
    begin
  L1: Choose some $(t,t') \in Q$ and delete it from Q;
      for every selection $w \in V$ where w = (sel t) do
        if u = SELECT(sel,t') is defined do
          begin
            add (w,u) to SP;
            L2: PROPAGATE(t,u);
          end;
      for all $u \in AS_{t'}-AS_t$ do L3: PROPAGATE(t,u);
      for all $w \in \overline{AS}_t-\overline{AS}_{t'}$ do L4: PROPAGATE(w,t');
    end;
  return SP;
end;
```

We require two Lemmas to demonstrate the correctness of Algorithm 3A.

Lemma 3.2.1 On every execution of Algorithm 3A we have for all $t,t' \in V-\hat{V}$ at label L0:

(i) $t \in AS_{t'}$ iff $t' \in \overline{AS}_t$

(ii) if $(t,t') \in Q$ then $t \in AS_{t'}$

(iii) if $t \in AS_{t'}$ then there exists an access sequence from t to t'.

Proof by induction on the number of executions of the main loop of Algorithm 3A.

Basis step Initially, Q = the set of all pairs (t,t') such that $t,t' \in V-\bar{V}$ and there exists a value path from t to t', (i),(ii) hold by the calls to PROPAGATE(t,t') at L0, and since any value path is also an access sequence, (iii) also initially holds.

Inductive step Suppose (i),(ii), and (iii) have held over previous executions of the main loop of Algorithm 3A, and consider some (t,t') deleted from Q at L1. By (ii) and (iii), there is an access sequence from t to t'. Observe that if there is an access sequence from text expression y to some text expression z, then after any call to PROPAGATE(y,z), (i),(ii), and (iii) still hold, and for our purposes that call is considered correct.

Case 1 If w is the selection (sel t) and u = SELECT(sel,t') is defined, then (w,u) is a selection pair, which is also an access sequence. Thus, the call to PROPAGATE(w,u) at L2 is correct.

Case 2 If $u \in AS_{t'} - AS_t$, then (ii) implies that there is an access sequence from t' to u, and hence there is an access sequence from t to u. Thus, the call to PROPAGATE(t,u) at L3 is correct.

Case 3 If $w \in \overline{AS}_t - \overline{AS}_{t'}$ then (iii) implies that $t \in AS_w$ and (iii) implies that there is an access sequence from t to w. Hence, there is an access sequence from w to t' and the call to PROPAGATE(w,t') at L4 is correct. □

Lemma 3.2.2 For all $t, t' \in V$, if there is an access sequence p from t to t' then (t,t') is eventually added to Q.

Proof by contradiction. Suppose (t,t') is not eventually added to Q, and let p be of minimal length. Note that if we have a call to PROPAGATE(t,t') then (t,t') is added to Q.

Case 1 If p is a value path from t to t' then there must be a call to PROPAGATE(t,t') at L0.

Case 2 If p is a selection pair then there exist text expressions y,z such that t is of the form t (sel y), t' = SELECT(sel,z), and furthermore there is an access sequence p' from y to z. Since p' is of length less than p, p' does not violate Lemma 3.2.2, so (y,z) is eventually added to Q, and hence there is a call to PROPAGATE(t,t') at L2.

Case 3 Otherwise $p = p_1 \cdot p_2$ where p_1 is an access sequence from t to y and p_2 is an access sequence from y to t'. Since p_1 and p_2 are of length less than p, Lemma 3.2.2 holds over p_1 and p_2, so both (t,y) and (y,t') are eventually added to (and later deleted from) Q.

<u>Case 3a</u> If (t,y) is deleted from Q after (y,t') then

$$t' \in ASy-ASt$$

and so there is a call to PROPAGATE(t,t') at L3.

<u>Case 3b</u> If (y,t') is deleted from Q after (t,y) then

$$t \in \overline{A}Sy-\overline{A}St'$$

and thus there is a call to PROPAGATE(t,t') at L4. □

<u>Theorem 3.2.3</u> Algorithm 3A correctly computes SP in $O(\ell^2+|\Sigma||A|)$ bit vector operations, where $\ell = |V-\overline{V}|$ is the the number of original text expressions before the "dummy" assignments are added.

<u>Proof</u> Suppose (t,t') is a selection pair. By Lemma 3.2.2, (t,t') is added to Q at the call to PROPAGATE(t,t') at L2, and hence (t,t') is also added to SP at L2.

Now suppose that (t,t') is added to SP at L2. Then (t,t') is added to Q in the call to PROPAGATE(t,t') at L2, and by the proof of Lemma 3.2.1, (t,t') is a selection pair.

New we consider the lower time bounds of Algorithm 3A. The computation of VP by [T1] costs $O(|V|+|E|) = O(\ell+|\Sigma||A|)$ bit vector steps. Also, the processing associated with each (t,t') added and then deleted from Q is a constant number of bit vector operations. There may be $O(\ell^2)$ such pairs and no such pair is added to Q more than once. Hence, the total cost of Algorithm 3A is $O(\ell^2+|\Sigma||A|)$ bit vector operations. □

3.3 Constant Propagation and Covers of Programs with Structured Data.

Let P be a program with a fixed interpretation for the constructor and selector signs as in the Introduction of this Chapter. Here we wish to determine text expressions which are constant over all executions of P, and more generally we wish to determine covers: symbolic expressions in EXP for the value of text expressions which hold over all executions of the program. The main difference between the covers of this section and those of Chapter 2 is that here we define a reduced expression to be derived from repeated selection reductions of the sort described in Section 3.1, as well as the usual constant reductions. A reduced expression $\alpha \in$ EXP covers text expression t if

$$\text{EXEC}(\alpha,p) = \text{EXEC}(t,p)$$

for all control paths p from the start block s to loc(t), the block in N where t is located. A cover of P is a mapping ψ from the text expressions to EXP such that for each text expression t, $\psi(t)$ covers t.

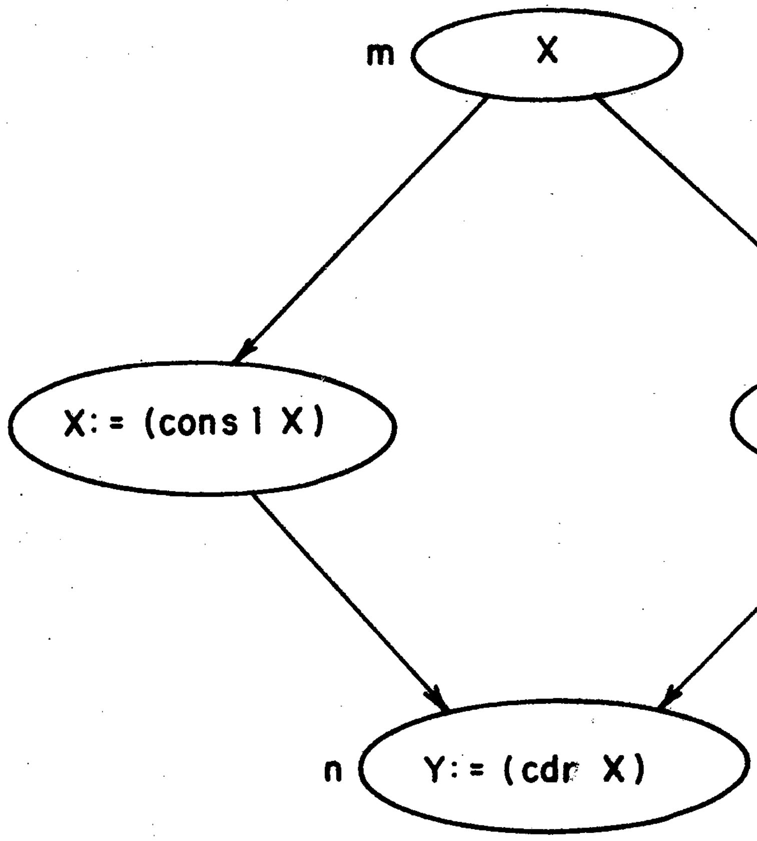


Figure 3.4. $Yn^{\rightarrow}$ = (cdr $X^{\rightarrow n}$) is covered by $X^{\rightarrow m}$.

Recall that the <u>origin</u> of an expression $\alpha \in$ EXP is intuitively the earliest point at which α is defined; formally origin(α) = s if α contains no input variables and otherwise origin(α) is the earliest block n $\in$ N (relative to the dominator ordering of the control flow graph F with the start block s first) such that an input variable $X^{\to n}$ appears in α (provided that this block is uniquely determined). Also, recall that $\overset{*}{\to}$ is the partial ordering of nodes in N by dominator relative of the control flow graph F = (N, A, s). We extend $\overset{*}{\to}$ to a partial ordering of covers. For covers ψ, ψ', $\psi \overset{*}{\to} \psi'$ iff origin(ψ(t)) $\overset{*}{\to}$ origin(ψ'(t)) for each text expression t. It follows from the results of Section 1.3 that if the program P is interpreted in the integer domain (i.e., ATOM is the set of natural numbers and the elementary operator signs in OP are interpreted as the usual arithmetic operations: addition, subtraction, multiplication, and division) then constant propagation is recursively unsolvable, and hence the determination of the covers minimal with respect to $\overset{*}{\to}$ is also impossible within the arithmetic domain.

Good, but not minimal, covers may be computed by an algorithm due to Kildall[Ki] (his algorithm is actually much more general; here we consider a specific application). After computing an approximate cover ψ_0, Kildall's algorithm iteratively compares the approximate covers of input variables to the approximate covers of the output expressions of the corresponding variables at preceding blocks, and propagates the changes to succeeding blocks. In Chapter 2, we define the covers computed by his algorithm as fixed points of a functional Ψ. Here we define a similar functional Ψ'. For any mapping ψ from text expression to EXP, let $\Psi'(\psi)$ be the mapping from text expressions to EXP such that for each text expression t, $\Psi'(\psi)$ is derived from t by repeatedly

(1) substituting expression α for every input variable $X^{\rightarrow n}$ such that $\alpha = \psi(X^{m\rightarrow})$ for all $(m,n) \in A$.

(2) substituting the expression α for any selection u in t such that $\alpha = \psi(u')$ for all selection pairs (u,u').

(3) reducing (by both selection and constant reductions) the resulting expression.

We shall show, as we did for a similar functional Ψ in Chapter 2, that the fixed points of Ψ' are covers and that there exists a unique, minimal fixed point of Ψ'.

Theorem 3.3.1 Each fixed point of Ψ' is a cover.

Proof by contradiction. Suppose ψ is a fixed point of Ψ' and ψ is not a cover. Let p be the shortest control path from the start block s to a block $n \in N$ containing a text expression t such that

$$\text{EXEC}(\psi(t),p) \not\equiv \text{EXEC}(t,p).$$

Furthermore assume for each proper subexpression t' of t,

$$\text{EXEC}(\psi(t'),p) = \text{EXEC}(t',p).$$

The case where t is an input variable was shown (in the proof of Theorem 2.1) to be an impossible case. Otherwise, $\psi(t) = \psi(t')$ for all selection pairs (t,t'). But there is a selection pair (t,t') such that

$$EXEC(t,p2) = t'$$

for $p = p1 \cdot p2$ where p1 ends and p2 begins at loc(t').

Hence, $EXEC(t,p) = EXEC(t',p1)$

$= EXEC(\psi(t'),p1)$ by the induction hypothesis

$= EXEC(\psi(t),p1)$ since $\psi(t) = \psi(t')$

$= EXEC(\psi(t),p)$. □

Let GVG = (V,E,L) be an arbitrary global value graph as defined in Section 3.2. In Section 2.2 we also defined the set Γ_{GVG} of mappings from the nodes of GVG to EXP such that for each $\psi \in \Gamma_{GVG}$ and node $v \in V$ in GVG,

(1) If v is labeled with a constant sign c then $\psi(v) = c$.

(2) If L(v) is a k-adic function sign θ and $u_1,\ldots,u_k$ are the immediate successors of v in GVG then $\psi(v)$ is the expression derived by <u>constant</u> reductions from $(\theta\ \psi(u_1)\ldots\psi(u_k))$.

(3) If v is labeled with an input variable, then either $\psi(v) = X^{\to n}$ or $\psi(v) = \alpha$ where $\alpha = \psi(u)$ for all value edges $(v,u) \in E$ departing from v.

Let Γ'_{GVG} be a set of mappings ψ from V to EXP such that for all $v \in V$, $\psi(v)$ satisfies cases (1), (2), (3), or the additional case

(3') L(v) is a selector sign and $\psi(v) = \alpha$ where $\alpha = \psi(u)$ for all selection pairs (v,u) departing from v.

Note that the set of nodes satisfying cases (3) and (3') are sufficient to characterize an element of Γ'_{GVG}; and hence Γ'_{GVG} is finite.

Let $GVG^* = (V, E, L)$ be the special global value graph defined in Section 2.2 where each node $v \in V$ is identified with the text expression which it represents (hence, the node set V is considered to be the set of text expressions) and the value edges of GVG^* represent the flow of values through the program. Recall that (t,t') is a value edge of GVG^* iff t is an input variable $X^{\to n}$ and t' is the output expression X^m for some $(m,n) \in A$. For any text expression t $\in$ V that is a selection, and $\psi \in \Gamma'_{GVG^*}$, if (3') holds for t then t is _simplifed_ by ψ. If in addition, $\psi(t) \neq$ _error_, then t is _properly_ _simplified_ by ψ. Selection t is (_properly_) _simplifiable_ if t is (properly) simplified by some element of Γ'_{GVG^*}.

Our proof that selection simplifications actually improve elements of Γ'_{GVG^*} (Theorem 3.3.2) will allow us to show that Γ'_{GVG^*} is a semilattice with respect to the partial ordering $\overset{*}{\to}$ (Theorem 3.3.3). The unique minimal element of Γ'_{GVG^*} will then be shown in Theorem 3.3.4 to be the minimal fixed point of Ψ'. We require first some technical Lemmas.

Lemma _3.3.1_ For each $t \in V$ which is a selection or input variable, and every control path from the start block s to loc(t), there is a maximal access sequence $(t=u_1,\ldots,u_k)$ such that $loc(u_1),\ldots,loc(u_k)$ are distinct blocks in p.

Proof by induction. We consider (t) to be a trivial access sequence. Suppose we have an access sequence $(t=u_1,\ldots,u_i)$ such that $loc(u_1),\ldots,loc(u_i)$ are distinct blocks in p. We further assume that $loc(u_i)$ occurs in p before $loc(u_1),\ldots,loc(u_{i-1})$. If u_i is neither a selection or input variable then $(t=u_1,\ldots,u_i)$ is a maximal access sequence. Otherwise, let p_i be the subsequence of p from s to the first occurrence of block $loc(u_i)$. Then there is a text expression u_{i+1} such that (1) $loc(u_{i+1})$ is contained in p_i and distinct from $loc(u_i)$ and (2) (u_i,u_{i+1}) is either a value edge (in the case u_i is an input variable) or a selection pair (if u_i is a selector sign). Hence $(t=u_1,\ldots,u_i,u_{i+1})$ is an access sequence and $loc(u_{i+1})$ is distinct from $loc(u_1),\ldots,loc(u_i)$. Since p is finite, we have our result. □

Lemma 3.3.1 will be used to construct maximal access sequences relative to fixed control paths. The next Lemma is analogous to Lemma 2.2.2 of Chapter 2.

Lemma 3.3.2 For each $\psi \in \Gamma'_{GVG^*}$ and $t \in V$, $origin(\psi(t)) \overset{*}{\to} loc(t)$.

Proof by contradiction. Suppose for some $t \in V$,

$$origin(\psi(t)) \overset{*}{\not\to} loc(t).$$

Then there must exist an input variable $X^{\to n}$ in $\psi(t)$ such that $n \overset{*}{\not\to} loc(v)$, and hence there is an n-avoiding control path p from the start block s to loc(t). Also, there must exist an $u \in V$ also located at n such that $\psi(u) = X^{\to n}$. By Lemma 3.3.1, there is a maximal access sequence $(t=u_1,\ldots,u_k)$ such that $loc(u_1),\ldots,loc(u_k)$ are distinct blocks in p. Let j be the maximal integer $\leq$ k such that $\psi(u_1) = \ldots = \psi(u_j)$. If $L(u_j)$ is an input variable the $\psi(u_1) = \psi(u_j) = X^{\to n}$, so $loc(u_k) = n$ is contained on p, contradicting the assumption that p avoids n. Otherwise, if $L(u_j)$ is a function sign or constant sign, then $\psi(u) = \psi(u_k) \overset{+}{\to} X^{\to n}$, a contradiction with $\psi(u) = X^{\to n}$. □

The following Lemma shows that certain covers of simplifiable selection operations have a very special form.

<u>Lemma 3.3.3</u> For each properly simplifiable selection $t \in V$, and $\psi \in \Gamma'_{GVG^*}$, if t is <u>not</u> simplified by ψ, then $\psi(t)$ is of the form $(sel_1 \ \ldots \ sel_k \ X^{\to n})$ where $sel_1,\ldots,sel_k$ are selector signs and $X^{\to n}$ is an input variable.

<u>Proof</u> by induction on subexpressions of $\psi(t)$.

<u>Basis Step</u>. By assumption t = (sel u) is not simplified by ψ, so $\psi(t)$ is of the form (sel $\psi(u)$). Also, note that since t is simplified by some element of Γ'_{GVG^*}, t has no departing selection pairs entering <u>error</u>.

<u>Induction step</u>. Suppose for some i, $1 \leq i \leq k$, $\psi(t)$ is of the form $(sel_1 \ \ldots \ sel_k \ \alpha)$. Consider any selector operation $t' = (sel_i \ u')$ such that $\psi(u') = \alpha$. We also assume in our induction hypothesis that t' has no departing

selection pairs entering _error_.

Suppose $\psi(u') = \alpha$ is not a selection operation or input variable.

Case 1. Suppose u' is an input variable. Let p be a control path from the start block s to loc(u'). By Lemma 3.3.1 we can construct a maximal access sequence from u' to some $\bar{u} \in V$. From this we can show that t' has a departing selection pair entering _error_, a contradiction with the induction hypothesis.

Case 2. Suppose u' is not an input variable. If u' is a construction operation for which sel_i is a selection, then t' is _not_ a reduced expression, which is impossible. Otherwise, if u' is a constant sign or some other sort of function application other than a selection, then t' has a departing selection pair entering _error_, a contradiction with the induction hypothesis.

Hence $\psi(u')$ is either a selection or input variable. To complete our induction proof, for any selection $\bar{t}$ such that $\psi(\bar{t}) = \alpha$, if $\bar{t}$ has a departing selection pair entering _error_, then so does t', a contradiction. □

Now we show that simplification of selection operations always improves an element of Γ'_{GVG^*}.

Theorem 3.3.2 For $\psi, \psi' \in \Gamma'_{GVG^*}$ and selection operation $t \in V$, if t is _not_ simplified by ψ and t is properly simplified by ψ' then $\text{origin}(\psi'(t)) \overset{+}{\rightarrow} \text{origin}(\psi(t))$.

<u>Proof</u>. For any $N' \subseteq N$, let LCA(N') be the latest (furtherest from the start block s) common ancestor of the nodes in N' relative to the dominator tree of the control flow graph F. By Lemma 3.3.2, $\psi(t)$ is of the form (sel1 $\cdots$ sel_k $X \rightarrow n$). We proceed by induction on subexpressions of $\psi(t)$.

Suppose for some i, $1 \leq i \leq k$, if $i < k$, for every selection $\overline{t} \in V$ such that $\psi(\overline{t}) = (\text{sel}_{i+1} \ldots \text{sel}_k\ X \overset{+}{\rightarrow} n)$ then LCA$\{\overline{w} \mid (\overline{t},\overline{w})$ is a selection pair departing from $\overline{t}\} \overset{+}{\rightarrow} n$. Consider any selection $t' \in V$ such that $\psi(t') = (\text{sel}_i \ldots \text{sel}_k$ $X \rightarrow n)$. Let u' be the immediate subexpression of t', so origin($\psi(t')$) = origin($\psi(u')$). Then there exists a (possibly trivial) maximal access sequence from u' to some $\overline{t}$ such that $\psi(u') = \psi(\overline{t})$. By the induction hypothesis, LCA$\{\overline{w}$ $\mid (\overline{t},\overline{w})$ is a selection pair departing from $\overline{t}\} \overset{+}{\rightarrow} n$. We can then show that origin($\psi'(t)$) $\overset{*}{\rightarrow}$ LCA$\{w' \mid (t',w')$ is a selection pair departing from $t'\} \overset{+}{\rightarrow} n$.

Since t is simplified by ψ', $\psi'(t) = \alpha$ where $\psi'(t') = \alpha$ for all selector pairs (t,t'). Hence,

origin($\psi'(t)$) = origin(α)

$\overset{*}{\rightarrow}$ LCA$\{w \mid (t,w)$ is a selection pair$\} \overset{+}{\rightarrow} n$.

$\overset{+}{\rightarrow} n$ = origin($\psi(t)$). □

In Section 2.2 we defined a partial function <u>min</u> from EXP^2 to EXP; we extend <u>min</u> to a partial mapping from $(\Gamma'\text{GVG}^*)^2$ to $\Gamma'\text{GVG}^*$ so that for each $\psi,\psi' \in \Gamma'\text{GVG}^*$, if $\overline{\psi}(t)$

= $\psi(t)$ min $\psi'(t)$ is defined for each text expression t, then ψ min $\psi' = \bar{\Psi}$, and otherwise ψ min ψ' is undefined.

Theorem 3.3.3 $\Gamma'GVG^*$ forms a finite semilattice with respect to $\overset{*}{\rightarrow}$.

Proof. It is sufficient to show that min is well defined over $\Gamma'GVG^*$. Suppose for ψ', $\psi \in \Gamma'GVG^*$, ψ min ψ' is defined, so there is a text expression t such that $\psi(t)$ min $\psi'(t)$ is undefined but $\psi(u)$ min $\psi'(u)$ is defined for all u which are proper subexpressions of t' such that $\psi(t) = \psi(t')$. Thus t is either a selection operation or an input variable. Consider any control path p from the start block s to loc(t). By Lemma 3.3.1, we can construct a maximal access sequence $(t=u_1,\ldots,u_k)$ such that $loc(u_1),\ldots,loc(u_k)$ are unique blocks of p. Let j be the maximal integer $\leq$ k such that $\psi(u_1) = \ldots = \psi(u_j)$. By the proof of Theorem 2.2.1 of Section 2.2, we need only consider the case where t_j is a selection operation (sel u). Since j is maximal, t_j is not simplified by ψ and $\psi(t_j)$ = (sel $\psi(u)$). If t_j is also not simplified by ψ' then $\psi'(t_j)$ = (sel $\psi'(u)$) and by the induction hypothesis $\alpha = \psi(u)$ min $\psi'(u)$ is defined, so $\psi(t_j)$ min $\psi'(t_j)$ = (sel α). Otherwise, suppose t_j is simplified by ψ'. If $\psi'(t_j)$ = error then $\psi(t_j)$ min $\psi'(t_j)$ = error. If t is properly simplified by ψ' then by Theorem 3.3.2, origin($\psi'(t_j)$) $\overset{+}{\rightarrow}$ origin($\psi(t_j)$), so $\psi(t_j)$ min $\psi'(t_j) = \psi'(t_j)$. □

Theorem 3.3.4 Ψ' has a unique, minimal fixed point ψ^* which is the minimal element of Γ'_{GVG^*}.

Proof Clearly, any fixed point of Ψ' is an element of Γ'_{GVG^*}. By Theorem 3.3.3, Γ'_{GVG^*} has a unique minimal element $\psi^* = \underline{\min}\ \Gamma'_{GVG^*}$. Let $\overline{\Psi}^* = \Psi'(\psi^*)$. In proof of Theorem 2.2.1, we showed that $\overline{\Psi}^*(X^{\rightarrow n}) = X^{\rightarrow n}$ for each input variable $X^{\rightarrow n}$ such that $\psi^*(X^{\rightarrow n}) = X^{\rightarrow n}$. Now suppose there is a selection $t \in V$ such that $\overline{\Psi}^*(t) = \alpha$ where $\alpha = \psi^*(t')$ for all selection pairs (t,t'), but $\psi^*(t) \neq \alpha$. Let ψ be the mapping from text expressions to EXP such that for each text expression u, $\psi(u)$ is derived from $\overline{\Psi}^*(u)$ by substituting α for each occurrence of $\overline{\Psi}(t)$ in $\overline{\Psi}^*(u)$, and reducing the resulting expression. Hence $\psi \in \Gamma'_{GVG^*}$ but by Theorem 3.3.2 $\text{origin}(\psi(t)) \overset{+}{\rightarrow} \text{origin}(\psi^*(t))$, a contradiction with the assumption that ψ^* is the minimal element of Γ'_{GVG^*}. □

In the next section we describe a method for actually constructing ψ^*, the minimal element of Γ'_{GVG^*}.

3.4 <u>The Computation of ψ^*, the minimal fixed point of Ψ'</u>

Now we describe a method for actually constructing ψ^*, the minimal fixed point of Ψ' which was shown in Theorem 3.3.4 to be the minimal element of $\Gamma' GVG^*$. There are two main steps. We first reduce constant propagation with selection and constant reductions to constant propagation with only constant reductions; the latter problem is solved efficiently by the methods of Chapter 2. We then find ψ^* by constructing, by the methods of Chapter 2, the minimal element of ΓGVG_0, ΓGVG_1, ... ΓGVG_R where GVG_0, GVG_1, ..., GVG_R is a sequence of global value graphs derived from GVG^*.

Associate with each text expression t which is a selection (sel u), a new , distinct program variable SV_t called the <u>selection variable of t</u>. The corresponding input variable $SV_t^{\rightarrow loc(t)}$ will be unambiguously represented by dropping its superscript. The selection variable SV_t is <u>installed in place of t</u> in GVG^* by relabeling t with the selection variable SV_t, deleting the edge (t,u) originally departing from t and adding the selection pairs departing from t to the edge set. Conversely, the selection variable SV_t is <u>replaced by t</u> by reversing this process.

Let GVG be a labeled digraph derived from GVG^* by replacing <u>any</u> number of selections with their corresponding selection variables. Note that by definition of selection pairs, for any selection t relabeled in GVG with selection variable SV_t, if p is a control path from the start block s to loc(t) then t has a departing selection pair (t,u), which is also a value edge of GVG, such that loc(u) is distinct from loc(t) and contained in p. Hence, GVG is a global value graph. Also, note that since the node set of GVG is V, the node set of GVG*, we continue to identify the nodes in V with text expressions. However selections in V may now be labeled in GVG with selection variables rather than selector signs. By Theorem 2.2.1, Γ_{GVG} has a unique, minimal element ψ. Chapter 2 gives an efficient method for the construction of ψ. Let us review these results.

GVG is <u>reduced</u> if $\psi(t)$ is the label of t in GVG for all $t \in V$ such that $\psi(t)$ is a constant sign. A reduced global value graph may be derived from GVG by the simple constant propagation algorithm presented in Section 2.3. We now assume GVG is reduced.

Recall that our method proceeds by induction on rank of text expressions. The <u>rank</u> of $t \in V$ labeled in GVG with a constant sign in GVG is 0. If t is labeled in GVG with a function sign θ, and $u_1,\dots,u_k$ are the immediate successors of t in GVG, then the rank of t in GVG is

$$1+\text{MAX}\{\text{rank}(u_1),\dots,\text{rank}(u_k)\},$$

and by definition of Γ_{GVG},

$$\psi(t) = (\theta\ \psi(u_1)\ \dots\ \psi(u_k)).$$

Note that the rank induces a topological ordering (from leaves to roots) of the dags of blocks from which GVG is built.

The case in which t is labeled with an input variable $X\to n$ is more difficult. Recall that a <u>value path</u> in GVG is a path p traversing only nodes linked by value edges and p is <u>maximal</u> relative to a fixed beginning node if p ends at a node with no departing value edges. The rank of t is

MIN{rank(w) | w lies at the end of a maximal value path in GVG from t}.

This $t \in V$ is a <u>value source</u> relative to ψ if $\psi(t) = X\to n$.

We have from Chapter 2

<u>Theorem 2.4</u>. t is a value source of ψ iff there exist two maximal, almost disjoint (containing only one element in common) value paths in GVG from t to $u_1, u_2 \in V$ such that $\psi(u_1) \neq \psi(u_2)$. Furthermore, for each $t \in V$ labeled with an input variable $X\to n$, either

(1) $\psi(t) = \psi(u)$ for all u contained at the end of maximal value paths in GVG from t, or

(2) $\psi(t) = \psi(u)$ where u is the unique value source contained on all maximal value paths in GVG from t.

The problem of discovering the value sources of ψ is reduced in Section 2.6 to the computation of dominator trees, for which there is an efficient algorithm due to Tarjan[T4].

The next Theorem reduces constant propagation with selection and constant reductions, to constant propagation with only constant reductions.

For this we will require two special mappings

M1(GVG): Γ_{GVG} to Γ'_{GVG^*}

M2(GVG): Γ'_{GVG^*} to Γ_{GVG}

For any $\psi \in \Gamma_{GVG}$, let M1(GVG)(ψ) be the mapping ψ_1 from V to EXP such that for all $t \in V$, $\psi_1(t)$ is derived from t by repeatedly

(1) substituting (sel ψ(u)) for each selection w = (sel u) such that $\psi(w) = SV_t$ is the selection variable of t.

(2) substituting $\psi(u)$ for each $u \in V$ labeled in GVG with an input variable.

Observe that $\psi_1 \in \Gamma'_{GVG^*}$.

Let ψ_2 = M2(GVG)(ψ) be the mapping from V to EXP such that for each $t \in V$,

(1) t' is derived by substituting selection variable SV_u for each nonsimplifiable selection $u \in V$ such that u is labeled in GVG with the selection variable SV_u.

(2) $\psi_2(t)$ is derived from t' by substituting $\psi^*(u)$ for each u labeled with an input variable in GVG and such that $\psi^*(u') = \psi(u')$ for each $u' \in V$ such that $\psi^*(u')$ is a proper subexpression of $\psi^*(u)$.

Observe that $\psi_2 \in \Gamma_{GVG}$.

<u>Theorem 3.4.1</u> If $\overline{GVG}$ is derived from GVG* by substituting selection variables for <u>all</u> selections, and $\overline{\Psi}$ is the minimal element of $\Gamma_{\overline{GVG}}$, then for each $t \in V$, $\psi^*(t)$ is a constant

sign c iff $\overline{\psi}(t) = c$.

<u>Proof</u> <u>IF</u>. Suppose $\overline{\psi}(t)$ is a constant sign c, but $\psi^*(t) \neq c$ Let $\psi_1 = M1(\overline{GVG})(\overline{\psi})$. Hence $\psi_1(t) \overset{*}{\rightarrow} \psi^*(t)$ and $\psi_1 \in \Gamma'_{GVG^*}$, contradiction with the assumption that ψ^* is the minima element of Γ'_{GVG^*}.

<u>ONLY IF</u>. Suppose $\psi^*(t)$ is a constant sign c, but $\psi'(t) \neq c$ Let $\psi_2 = M2(\overline{GVG})(\overline{\psi})$. Then $\psi_2(t) \overset{*}{\rightarrow} \overline{\psi}(t)$ and $\psi_2 \in \Gamma_{\overline{GVG}}$, contradiction with the assumption that $\overline{\psi}$ is the minima element of $\Gamma_{\overline{GVG}}$. □

We now define a sequence of global value graphs

$GVG_0, GVG_1, \ldots$

derived from $\overline{GVG}$. GVG_0 is the reduced graph derived fro $\overline{GVG}$. For $r = 0,1,\ldots$ let NSS(r) be the set of selection of rank r which are not simplifiable and let GVG_{r+1} b derived from GVG_r by restoring each $t \in NSS(r)$; i.e., if t (sel u) then the label of t is set to sel, all selectio pairs departing from t are deleted, replaced by the origina edge (t,u). Let ψ_r be the minimal element of Γ_{GVG_r}. Let = MAX{r | GVG_r contains a node of rank r}.

<u>Theorem</u> <u>3.4.2</u> $\psi_R = \psi^*$.

<u>Proof</u> Observe that each selection $t \in V$ is labeled with selection variable SV_t iff t is not simplifiable. Alsow have $\psi^* \in \Gamma_{GVG_R}$ which implies that $\psi^* \overset{*}{\rightarrow} \psi_R$, and we have ψ_R Γ'_{GVG^*} which implies that $\psi_R \overset{*}{\rightarrow} \psi^*$. Hence $\psi^* = \psi_R$. □

The remaining problem is the determination o simplifiable selections in V.

Theorem 3.4.3 For all selections $t \in V$ of rank r and labeled in GVG_r with a selection variable, $t \in NSS(r)$ iff t is a value source relative to ψ_r.

Proof IF. Suppose $t \in NSS(r)$ but t is not a value source of ψ_r, so $\psi_r(t) = \alpha$ where $\alpha = \psi_r(t')$ for all selection pairs (t,t'). Let $\psi_r' = M1(GVG_r)(\psi_r)$. Then $\psi_r' \in \Gamma_{GVG^*}$ and t is simplified by ψ_r', so by Theorem 3.3.2, $origin(\psi_r'(t)) \overset{+}{\rightarrow} origin(\psi^*(t))$, a contradiction with the assumption that ψ^* is the minimal element of Γ_{GVG^*}.

ONLY IF. Suppose t is simplified by ψ^*, so there exists an expression α such that $\psi^*(t) = \psi^*(t') = \alpha$ for all selection pairs (t,t'). Let $\hat{\psi}_r = M2(GVG_r)(\psi_r)$. Then $\hat{\psi}_r(t) = \hat{\psi}_r(t') = \alpha$ for all selection pairs (t,t') and $\hat{\psi}_r \in \Gamma_{GVG_R}$. If t is a value source of ψ_r, then by Theorem 3.3.2, $origin(\hat{\psi}_r) \overset{+}{\rightarrow} origin(\psi_r)$, a contradiction with the assumption that ψ_r is the minimal element of Γ_{GVG_r}. Thus t is not a value source of ψ_r. □

Let a trivial value path be of the form (t) where $t \in V$ is a node labeled with either a constant or function sign.

Corollary 3.4.1 For each $t \in V$, t is of rank r in GVG_R iff there is a (possibly trivial) maximal value path in GVG_r from t to a node of rank r in GVG_r and such that p avoids all elements of NSS(r).

Proof Observe that for any $t \in V$ labeled in GVG_r with a constant or function sign, t is of rank r in GVG_r iff t is of rank r in GVG_R. Otherwise, suppose $t \epsilon V$ is labeled with an input variable in GVG_r.

Suppose t is of rank r in GVG_R. Then there is a maximal value path p from t to some $t' \epsilon V$ such that all nodes of p are of rank r in GVG_R. Hence, p avoids all elements of NSS(r), and p is a maximal value path in GVG_r. Since t' is labeled with a constant or function sign, t' is also of rank r in GVG_r.

Suppose, on the other hand, that there is in GVG_r a maximal value path p from t to some t' or rank r in GVG_r such that p avoids all elements of NSS(r). It is always possible to find such a p containing only nodes of rank r in GVG_r. Hence, p is a maximal value path of GVG_r and t is of rank r in GVG_R. □

Our algorithm for computing ψ^*, the minimal fixed point of Ψ', is summarized below.

Algorithm 3B.

INPUT GVG*.

OUTPUT ψ^*, the minimal fixed point of Ψ'.

```
begin
  Discover all selection pairs by Algorithm 3A;
  Let $\overline{GVG}$ be derived from GVG$^*$ by installing a seletion
  variables in the place of each selection;
  Apply Algorithm 2A to construct GVG$_0$, the reduction of
  $\overline{GVG}$;
  for r := 0 by 1 to oo do
    begin
    Apply Algorithm 2B to construct V$_r$, the set of all text
    expressions of rank r in GVG$_r$;
    if V$_r$ is empty then return $\psi^*$;
    Compute by Algorithm 2C, $\psi_r$, the minimal element of
    $^r$GVG$_r$;
    Let NSS(r) be the set of selection of V$_r$ which are value
    sources relative to $\psi_r$;
    comment By Theorem 3.4.3, NSS(r) is the set of
    nonsimplifiable selections of rank r in GVG$_r$;
    for all t $\epsilon$ V$_r$ contained on a (possibly trivial) maximal
    value path in GVG$_r$ avoiding all elements of NSS(r) do
      begin
        comment By Corollary 3.4.1, t is of rank r in GVG$_R$;
        $\psi^*$(t) := $\psi_r$(t);
      end;
    Let GVG$_{r+1}$ be derived from GVG$_r$ by replacing each
    selection variable SV$_t$ in VS$_r$ with the original
    selection t;
    end;
end;
```

Let ℓ be the length of the text of program P and recall that GVG* is of size $O(|V|+|E|) = O(\ell+|\Sigma||A|)$.

Theorem 3.4.4 Algorithm 3B is correct and costs $O(\ell^2+|\Sigma||A|)$ bit vector and $O(\ell(\ell+|\Sigma||A|))$ elementary operations.

Proof The correctness of Algorithm 3b follows directly from Theorems 3.4.1-3.4.3.

By Theorem 3.2.2, the computation of all selector edges by Algorithm 3A costs $O(\ell^2+|\Sigma||A|)$ bit vector operations. For each $r = 1,2,...$ the computation of Ψ_r may cost $O(\ell+|\Sigma||A|)$ elementary operations by the results of Chapter 2. Since the maximum r such that V_r is not empty is $\leq \ell$, the total time cost of Algorithm 3b is $O(\ell^2+|\Sigma||A|)$ bit vector and $O(\ell(\ell+|\Sigma||A|))$ elementary operations. □

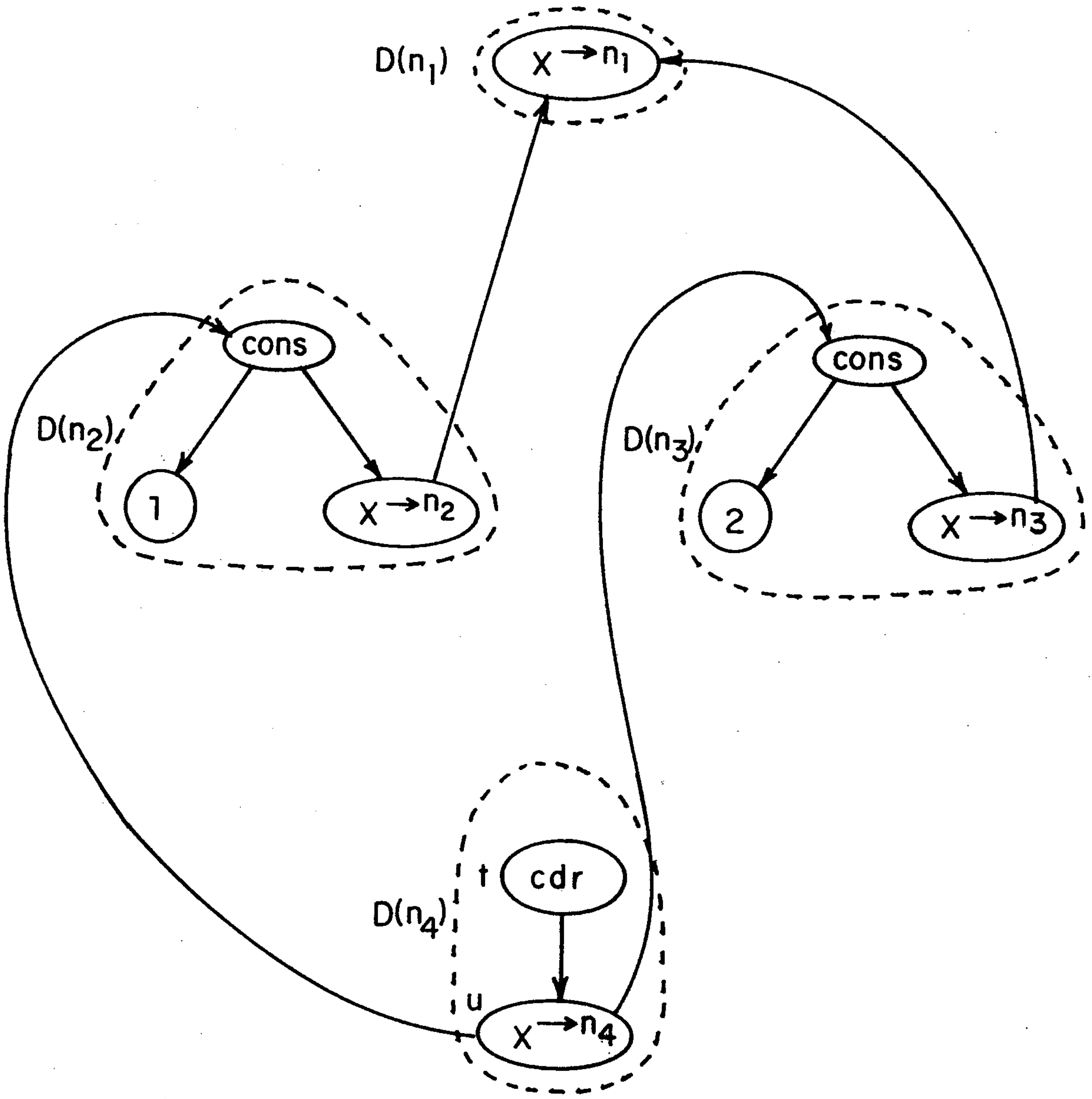


Figure 3.5. The global value graph GVG* for the program of Figure 3.4.

3.5 Type Covers and Type Declarations.

Types are expressions used to specify the shape of structured objects. A *type cover* of text expression t is a closed form expression for the *type* of t which holds on all executions of the program P. We show that the methods of the last section may be applied to the construction of type covers. Type covers have applications analogous to the usual sort of covers used to represent *values* of text expressions. For example, if the type cover of text expression t is a constant type, then the value of t has a fixed type over all executions of P. Text expressions which have the same type cover have values of the same shape on each execution of P (whereas, text expressions given the same type declaration may have different values of different shape over particular executions of P; see the latter part of this Section). A text expression t which is a construction operation is *redundant* if (1) every execution of P from the start block s to loc(t) passes thru a block containing a text expression t' with type cover common to t and furthermore, (2) the structured object computed by t' is dead (not referenced) on every execution path following loc(t) (in other words, the storage allocated for t' could be used to store t).

A *type declaration of program P* is used to specify, for each text expression t, the set of all types of values that t may evaluate to, over all executions of P. A _recursive type declaration_ uses recursion to specify infinite sets of types. In the latter part of this Section we discuss methods due to Tennenbaum[Te] and Schwartz[Sc2] for the automatic construction of type declarations for "type-free" programs (programs written without explicit type declarations). The method due to Schwartz is direct (noniterative) and more powerful than Tennenbaum's iterative method since it results in recursive type declarations for text which may have an infinite set of types (whereas, the method of Tennenbaum results in weaker, non-recursive type declarations).

We shall observe that the set of all possible types of a given text expression, over all executions of the program P, need not be a context-free language although the type declaration facilities of most programming languages are essentially context-free grammars. Hence, it is not possible to construct "tight" (exact) type declarations within most programming languages.

Fix (U,I) as an interpretation of program P as described in Section 3.1. Recall that the _universe of structured values_ U is built from a set of atoms in the fixed set ATOM and k-adic constructor signs in CONS. Also, recall that EXP is the set of expressions built from input variables (representing the value of program variables on input to blocks in N), constant signs in C, and k-adic function signs in θ (including operator signs in OP, constructor signs in CONS, and selector signs in SEL).

Let τ be a mapping initially of domain ATOM $\cup$ Σ into EXP such that

(1) For each $a \in$ ATOM, $\tau(a)$ is a symbol denoting the type of a.

(2) For each program variable $X \in \Sigma$, there exists an unique variable $\tau(X) = TX$.

Extend τ to a homomorphic mapping from EXP to EXP thusly:

(a) for each constant sign $c \in C$, if c is of the form $X^{\to s}$ (representing the value of program variable X on input to the start block s) let $\tau(X^{\to s}) = \tau(X)^{\to s} = TX^{\to s}$. Otherwise, let $\tau(c) = \tau(I(c))$.

(b) for each input variable $X^{\to n}$, $\tau(X^{\to n}) = \tau(X)^{\to n} = TX^{\to n}$.

(c) τ distributes over function applications thus:

$$\tau(\theta\ \alpha_1 \ldots \alpha_k) = (\theta\ \tau(\alpha_1) \ldots \tau(\alpha_k)).$$

Also, extend τ to subsets S of EXP and U:

$$\tau(S) = \{\tau(\alpha) \mid \alpha \in S\}.$$

A <u>type cover</u> of program P is a mapping ψ from the text expressions of P to τ(EXP) such that for each text expression t of P,

$$\text{EXEC}(\psi(t),p) = \tau(\text{EXEC}(t,p))$$

for all control paths p from the start block s to loc(t).

For example, consider the control flow graph of Figure 3.7. Let $\tau(1)$ = int. Note that $Z^{n\rightarrow}$ and $X^{m\rightarrow}$ do not have the same covers but do have the same type cover (cons int $TY^{\rightarrow m}$).

Let P_τ be the program derived from P by substituting $\tau(t)$ for each text expression t. Fix $(\tau(U), I_\tau)$ be the interpretation of P_τ where I_τ is the identity mapping over τ(ATOM), and for each k-adic elementary operation sign op ϵ OP (recall that I(op) is a mapping from ATOM^k to ATOM) and $a_1,\ldots,a_k$ ϵ ATOM,

$$I_\tau(op)(\tau(a_1),\ldots,\tau(a_k)) = \tau(I(op)(a_1,\ldots,a_k)).$$

Theorem 3.5.1 ψ is a cover of P_τ iff ψ is a type cover of P.

Proof Consider any text expression t and control path from the start block s to loc(t). By the fact that τ is a homomorphism over EXP,

$$\text{EXEC}(\tau(t),p) = \tau(\text{EXEC}(t,p)).$$

If ψ is a cover of P_τ then

$$\begin{aligned}\text{EXEC}(\psi(t),p) &= \text{EXEC}(\tau(t),p),\\ &= \tau(\text{EXEC}(t,p)).\end{aligned}$$

On the other hand, if ψ is a type cover of P then

$$\begin{aligned}\text{EXEC}(\psi(t),p)) &= \tau(\text{EXEC}(t,p))\\ &= \text{EXEC}(\tau(t),p). \quad \square.\end{aligned}$$

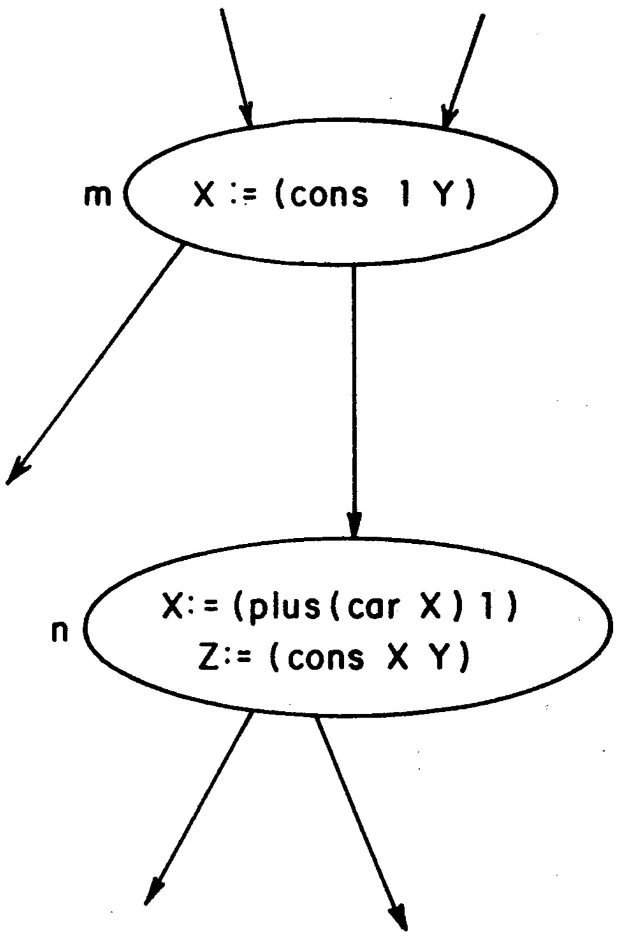
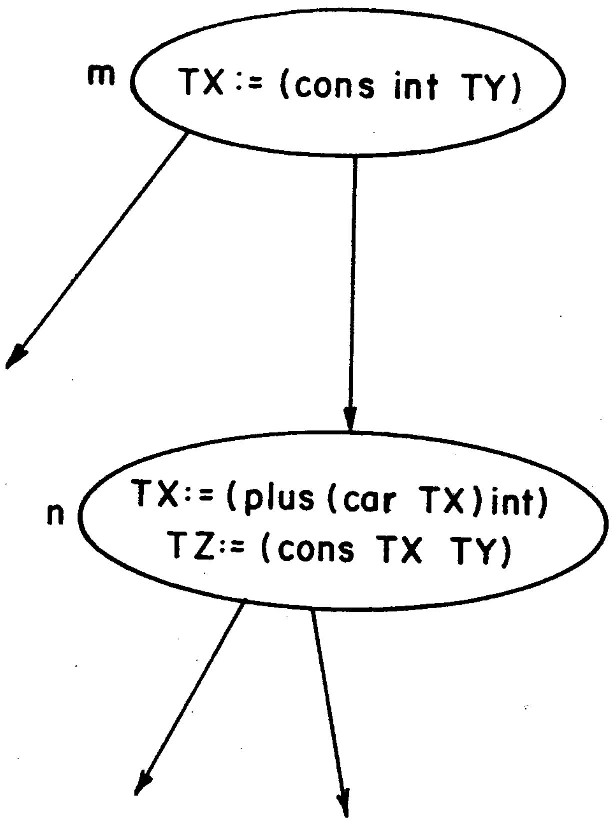


Figure 3.7. The control flow graph of a program P in LISP.



igure 3.8. The type program P_τ derived from the program P f Figure 3.7.

Let TV = $\{T_1,T_2,\ldots\}$ be a set of _type variables_; in the following we assume TV and the special symbol _oneof_ is distinct from the elements of τ(ATOM) and CONS. We distinguish the type variable _any_ ϵ TV which will represent the set of all types. Let TEXP be the set of expressions built from τ(ATOM), TV, and CONS. We shall assume that for a fixed program P, CONS and τ(ATOM) are finite sets.

A _type declaration_ for input variable $X^{\to}n$ consists of a statement of the form

declare $X^{\to}n$ _type_ α

where α ϵ TEXP.

A type declaration is interpreted in the context of a _type variable definition block_ TDEF, consisting of a sequence of statements of the form

T = _oneof_$\{\alpha_1,\ldots,\alpha_k\}$

(or just T = α_1 if k=1) where T ϵ TV-{any} and $\alpha_1,\ldots,\alpha_k$ ϵ TEXP. We assume no T ϵ TV occurs more than once on the left hand side of a statement in TDEF.

We now construct a set of productions (in the sense of formal language theory) by substituting for each statement

T = _oneof_$\{\alpha_1,\ldots,\alpha_k\}$

of TDEF, the context-free productions

$T \to \alpha_1, T \to \alpha_2, \ldots, T \to \alpha_k$.

Also, for the special symbol any we have the productions

any $\to \tau(a)$

for each a ϵ ATOM, and

any $\rightarrow$ (cons any ...k-times... any)

for each k > 0 and k-adic constructor sign cons ϵ CONS. For each T ϵ TV, let TDEF[T] be the context-free language generated by these productions with T considered to be the start symbol, the type variables as nonterminals, and the terminal symbols are taken from CONS $\cup$ τ(ATOM). Note that TDEF[T] is a subset of τ(U) and TDEF(any) = τ(U). Also, for each α ϵ TEXP, let TDEF[α] be the set {α' ϵ τ(U) | α' is derived from α by substituting some element of TDEF[T] for each type variable T occurring in α}.

For each α ϵ τ(U), let EXPAND(α) = {α' ϵ τ(U) | α' is derived from α by substituting some element of τ(U) for each constant sign of the form $X^{\rightarrow s}$}. For each input variable $X^{\rightarrow n}$, let TYPES($X^{\rightarrow n}$) = {α | α ϵ EXPAND(τ(EXEC($X^{\rightarrow n}$,p)) and such that p is some control path from the start block s to n}.

Consider again the type declaration

<u>declare</u> $X^{\rightarrow n}$ <u>type</u> α.

This type declaration is <u>proper</u> in the context of TDEF if

TYPES($X^{\rightarrow n}$) $\subseteq$ TDEF[α]

and is <u>tight</u> if

TYPES($X^{\rightarrow n}$) = TDEF[α]

For example, a proper type declaration for input variable $Xn^{\rightarrow}$ of Figure 3.7 is

<u>declare</u> $X^{\to n}$ <u>type</u> (cons int any).

Although the type definition facilities of many programming languages employ essentially the above scheme, it is interesting to note the scheme is not even powerful enough to give tight type definitions of programs without selection operations. Let f, g, and h be constructor signs of arity 1,1, and 3 respectively. Also, let $\tau(0)$ = int. In Figure 3.9,

$$\begin{aligned} \text{TYPES}(Zm^{\to}) &= \text{TYPES}(Z^{\to}n) \\ &= \{h(f^k(\text{int}),g^k(\text{int}),f^k(\text{int})) \mid k \geq 1\} \end{aligned}$$

which is clearly not a context-free language language and hence is not definable by the above type declaration scheme.
□

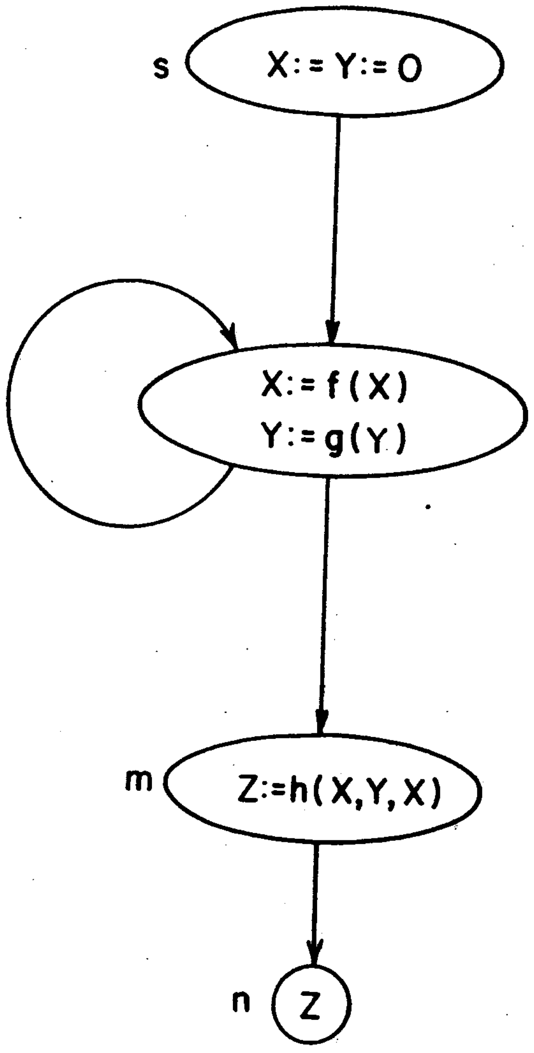


Figure 3.9. There is no tight type declaration for input variable $Z^{\to n}$.

We now describe a simplified version of the method of Schwartz[Sc2] for constructing proper (but not necessarily tight) type declarations. We require the special global value graph GVG^* and selection pairs of Section 3.2. To simplify the method, we assume that for each k-adic elementary operation sign op ϵ OP, there exists a unique α_{op} ϵ $\tau(U)$ such that $\alpha_{op} = \tau(I(op)(a_1,\ldots,a_k))$ for all $a_1,\ldots,a_k$ ϵ ATOM.

For each text expression t which is a selection, let SV_t ϵ TV be the unique <u>selection variable</u>. Let $\hat{\tau}$ be the mapping from text expressions to TEXP such that,

(1) For each constant sign c ϵ C,

  (a) if c is of the form $X^{\rightarrow s}$ (representing the value of program variable X on input to the start block s) then $\hat{\tau}(c)$ = any,

  (b) and otherwise, let $\hat{\tau}(c) = \tau(c)$.

(2) For each input variable $X^{\rightarrow n}$, $\hat{\tau}(X^{\rightarrow n}) = TX^{\rightarrow n}$.

(3) For each function application $t = (\theta\ \alpha_1\ldots\alpha_k)$,

  (a) if θ is a elementary operator op ϵ OP then

$$\tau(t) = \alpha_{op}.$$

  (b) if θ is a constructor sign cons ϵ CONS then $\tau(t)$ = (cons $\hat{\tau}(\alpha_1)\ldots\hat{\tau}(\alpha_k)$).

  (c) if θ is a selector sign in SEL then

$$\hat{\tau}(t) = SV_t,$$

  the unique selection variable associated with t.

We assume that $TX^{\to n} \in TV$, for each input variable $X^{\to n}$. Consider the special type variable definition block $TDEF^*$ such that for each input variable $X^{\to n}$, we have the statement:

$$TX^{\to n} = \underline{\text{oneof}}\{\hat{\tau}(t) \mid (X^{\to n},t) \text{ is a value edge of } GVG^*\},$$

and for each text expression t which is a selection, there is a type declaration statement:

$$SV_t = \underline{\text{oneof}}\{\hat{\tau}(u) \mid (t,u) \text{ is a selection pair}\}.$$

<u>Theorem 3.5.2</u> For each text expression t and each control path from the start block s to loc(t), $EXPAND(\tau(EXEC(t,p)))$ is contained in $TDEF^*[\hat{\tau}(t)]$.

<u>Proof</u> Let p be the shortest control path from the start block s to some block n containing a text expression t such that $EXPAND(\tau(EXEC(t,p)))$ is not contained in $TDEF^*[\hat{\tau}(t)]$. Clearly, $n \neq s$. We proceed by induction on subexpressions of t.

Consider a constant sign c. If c is of the form $X^{\to s}$ then $\hat{\tau}(X^{\to s}) = any$ and so $EXPAND(\tau(EXEC(X^{\to s}))) = \tau(U) = TDEF[any]$. Otherwise $\tau(EXEC(c,p)) = \tau(c) = \hat{\tau}(c)$ and so $EXPAND(\tau(EXEC(c,p))) = \{\tau(c)\} = TDEF^*[\hat{\tau}(c)]$.

For each input variable $X^{\to n}$, $EXEC(X^{\to n},p) = EXEC(X^{m\to},p')$ where $p = p' \cdot (m,n)$. By the induction hypothesis, $EXPAND(\tau(EXEC(X^{m\to},p')))$ is contained in $TDEF^*[\hat{\tau}(X^{m\to})]$. By

definition, $TDEF^*[\tau(X^{\to n})]$ contains $TDEF^*[\hat{\tau}(X^{m\to})]$. Henc
$EXPAND(\tau(EXEC(X^{\to n},p))) = EXPAND(\tau(EXEC(X^{m\to})))$ is containe
in $TDEF^*[\hat{\tau}(X^{\to n})]$.

If u is a selection contained within t, then u has
departing selection pair (u,u') such that $EXEC(u,\bar{p})$ =
$EXEC(u',\bar{p})$ for some control path $\bar{p}$ which is a subsequence o
p starting at s. By definition, $\hat{\tau}(u)$ is the selectio
variable SV_u and $TDEF^*[SV_u]$ contains $TDEF^*[\hat{\tau}(u')]$. Also, b
the induction hypothesis $EXPAND(\tau(EXEC(u',\bar{p})))$ is contained
in $TDEF^*[\hat{\tau}(u')]$. Hence, $EXPAND(\tau(EXEC(u,p)))$ is contained
in $TDEF^*[\hat{\tau}(u)]$.

Now suppose t is a function application $(\theta\ t_1 \ldots t_k)$
such that θ is not a selection and $\tau(EXEC(t_i,p))$ is
contained in $\hat{\tau}(t_i)$ for $i = 1,\ldots,k$. But

$$\tau(EXEC(t,p)) = EXEC((\theta\ \tau(t_1)\ldots\tau(t_k)),p)$$
$$= (\theta\ \tau(EXEC(t_1,p))\ldots\tau(EXEC(t_k,p)))$$

Hence $\tau(EXEC(t,p))$ is contained in $\hat{\tau}(t) = (\theta\ \hat{\tau}(t_1)\ldots\hat{\tau}(t_k))$,
a contradiction. □.

This immediately implies that

Corollary 3.5 For each input variable $X^{\to n}$,

declare $X^{\to n}$ type $TX^{\to n}$

is proper, relative to type variable definition block $TDEF^*$.

The above method for constructing type declarations is
due to Scwartz[Sc2]. We conclude by listing $TDEF^*$ for the
program of Figure 3.1. Let $\tau(nil)$ = null, $\tau(0)$ = int, t =

(car $X^{\rightarrow n}$), and t' = (cdr $X^{\rightarrow n}$).

TDEF* = (T$X^{\rightarrow m}$ = oneof{null, (cons int T$X^{\rightarrow m}$)},

T$Y^{\rightarrow m}$ = null,

T$Z^{\rightarrow m}$ = int,

T$X^{\rightarrow n}$ = $SV_{t'}$,

T$Y^{\rightarrow n}$ = oneof{T$Y^{\rightarrow m}$, (cons SV_t T$Y^{\rightarrow n}$)},

SV_t = oneof{error, int},

$SV_{t'}$ = oneof{error, null, T$X^{\rightarrow m}$})

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 <u>control flow graph</u>. We present an algorithm for symbolic evaluation requiring $O(\ell+a\alpha(a))$ bit vector operations on <u>all</u> 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 <u>simple cover</u> 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 $\{W(X^{\rightarrow n} \mid X^{\rightarrow n}$ occurs in n$\}$, relative to the dominator relation.

(2) An expression $\alpha \in$ EXP covers text expression t if α represents the value of t over all executions of the program. The origin 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 cover is a mapping from text expressions to covering expressions, and is minimal 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 simple cover which is a cover ψ such that for each text expression t, $\psi(t)$ is derived from t by substituting $\psi(X^{m\rightarrow})$ for each input variable $X^{\rightarrow n}$ such that $m = W(X^{\rightarrow n})$ 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 $X^{m\rightarrow} = X^{\rightarrow m}$ is a new text expression. At most $O(\ell)$ dummy assignments must be so added.

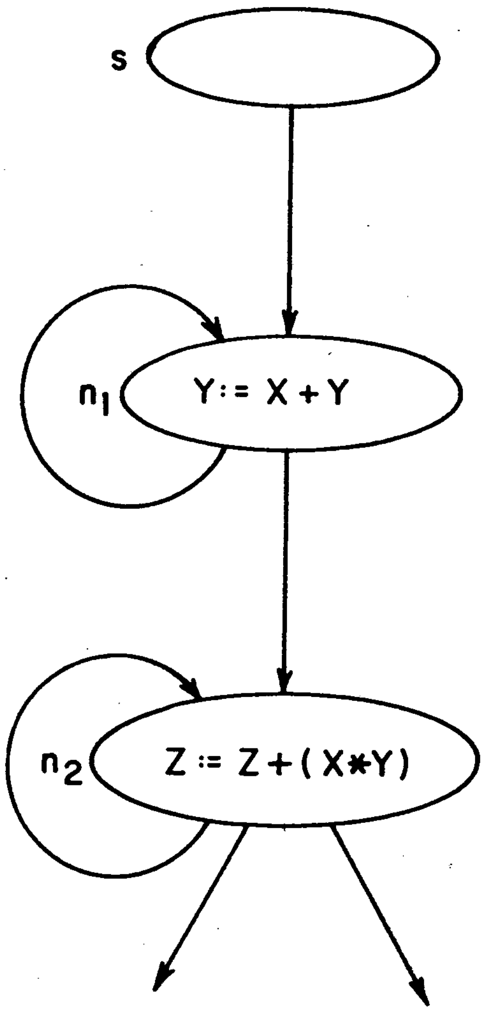


Figure 4.1 $Z^{n\rightarrow} = Z^{\rightarrow n}+(X^{\rightarrow n}*Y^{\rightarrow n})$ has simple cover $Z^{\rightarrow n}+(X^{\rightarrow s}*(X^{\rightarrow s}+Y^{\rightarrow m}))$.

(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 ε 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 ε N-{s} let IDOM(n) be the immediate dominator of n, and let

$$\text{IDEF}(n) = \text{DEF}(\text{IDOM}(n),n)$$

i.e. the set of program variables defined between IDOM(n) and n. The above equation may be inverted as follows:

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

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 $\vec{F}$ 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 $G_W^{\prime}$ 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

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

and otherwise, if S is nontrivial let

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

Also, define

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

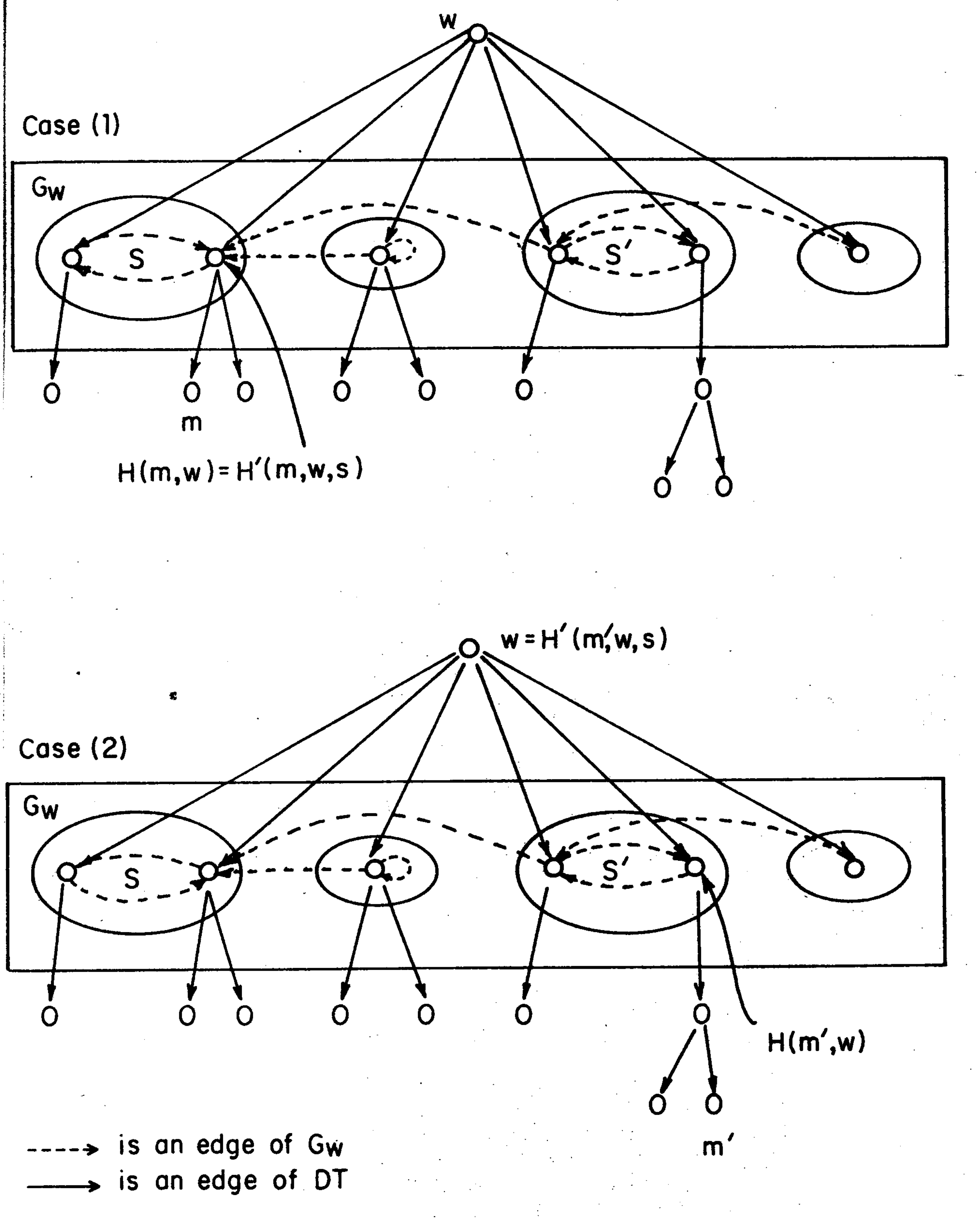


Figure 4.2.

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

<u>Theorem</u> <u>4.2.1</u> 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'.)

<u>Proof</u> 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 $1 < i < k$.

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

<u>Case</u> <u>2</u>. 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 $(u_{j-1},u_j) \in A_w$.

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

<u>Case</u> <u>2.2</u>. 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 algorith involves path compression on balanced trees and i frequently used in the implementation of UNION-FIND disjoin set operations. Also, Tarjan[T4] gives an almost linea time algorithm (again utilizing the method of pat compression) for processing a sequence of FIND, LINK, an EVAL instructions, given that the sequence is know beforehand, except for the values which are to label th edges in the LINK operations.

The following algorithm for computing IDEF uses, lik the algorithm of [T5], a preprocessing stage that execute all FIND and LINK instructions <u>but</u> <u>not</u> EVAL instructions this allows us in the second pass to efficiently process th 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 $(m,n) \in A$ such that IDOM(n) = w
      and $m \neq w$ 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 $n \in S$ do
            $Q_S := Q_S \cup OUT(n)$;
        comment add $Q_S^2$ to $Q_S$;
        for all $n \in S$ do
          for all $(m,n) \in A_w$ do
            $Q_S := Q_S \cup EVAL(m) \cup OUT(m)$;
        for all $n \in S$ do
          begin
        L4: LINK(n,w,$Q_S$);
            comment apply Theorem 4.2.1;
            IDEF(n) := $Q_S$;
          end;
      end;
    end;
end;
```

<u>Theorem</u> <u>4.2.2</u> Algorithm 4A correctly computes IDEF.

<u>Proof</u> (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

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

and then add to Q_S the set

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

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

$= Q_S^2$

Hence by Theorem 4.2.1, for each $n \in N$, IDEF(n) is correctly set to $Q_S = Q_S^1 \cup Q_S^2$.

Let S' be the 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 $H(m,w) \in S$, 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. □

<u>Theorem</u> <u>4.2.3</u>. Algorithm 4A costs an almost linear number of bit vector operations.

<u>Proof</u> 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 $r_w = |\text{IDOM}^{-1}[w]|+|A_w|$. 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]. $\square$

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

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

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

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

for all n' on the domination chain following $W(X^{\to n})$ to n. To assure the 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 4.3.1

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

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

Proof By definition of R, if X ϵ R(n) then there exists some node m ϵ 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 ϵ 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 ϵ 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.

<u>Proof</u> 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 $O(\ell+|A|\alpha(|A|))$. $\square$

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 $n \epsilon N$, 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 $n \epsilon N$, 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 $Xm^{\rightarrow}$, 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.

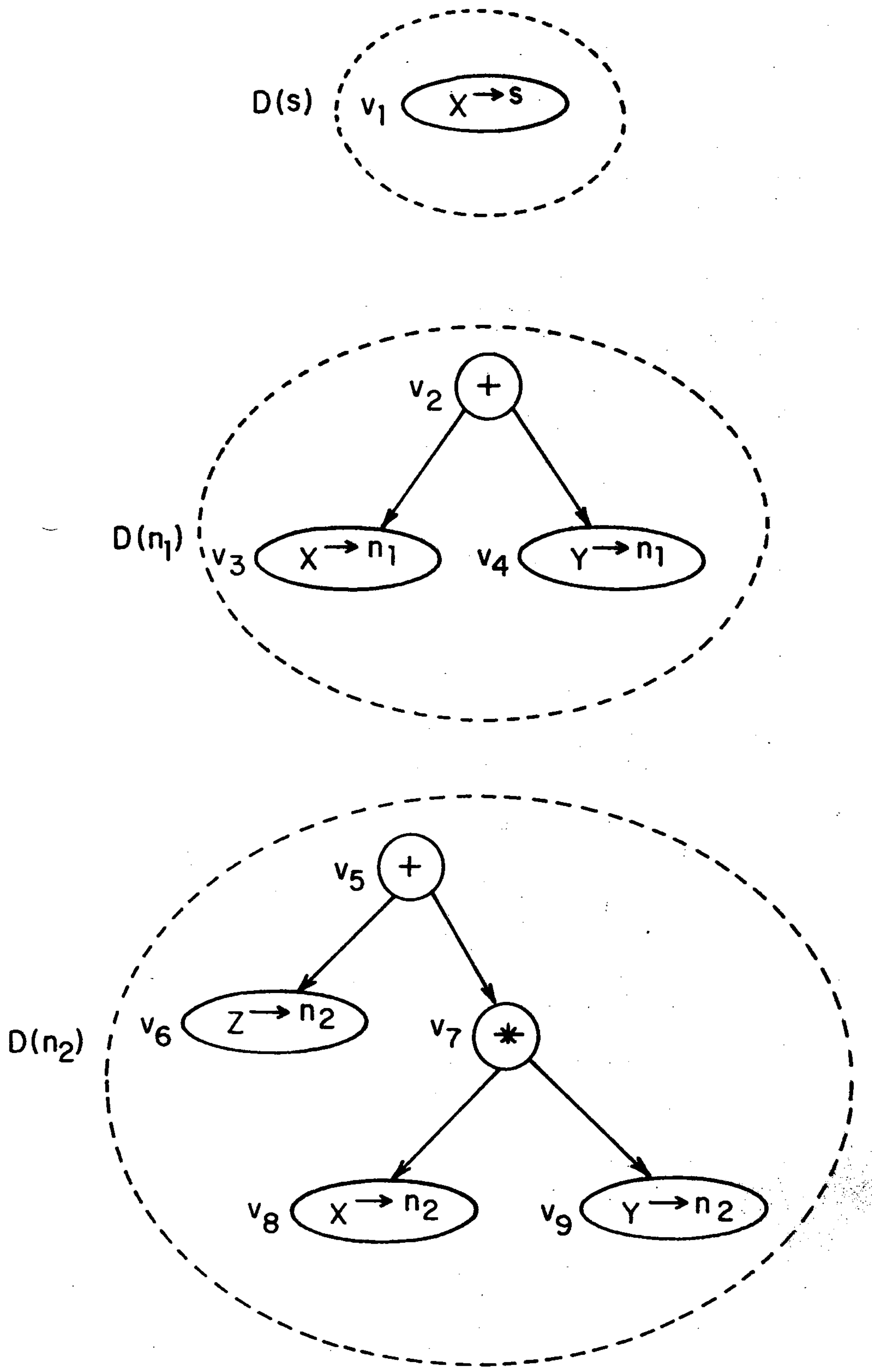


Figure 4.3 The dags of the program in Figure 4.1.

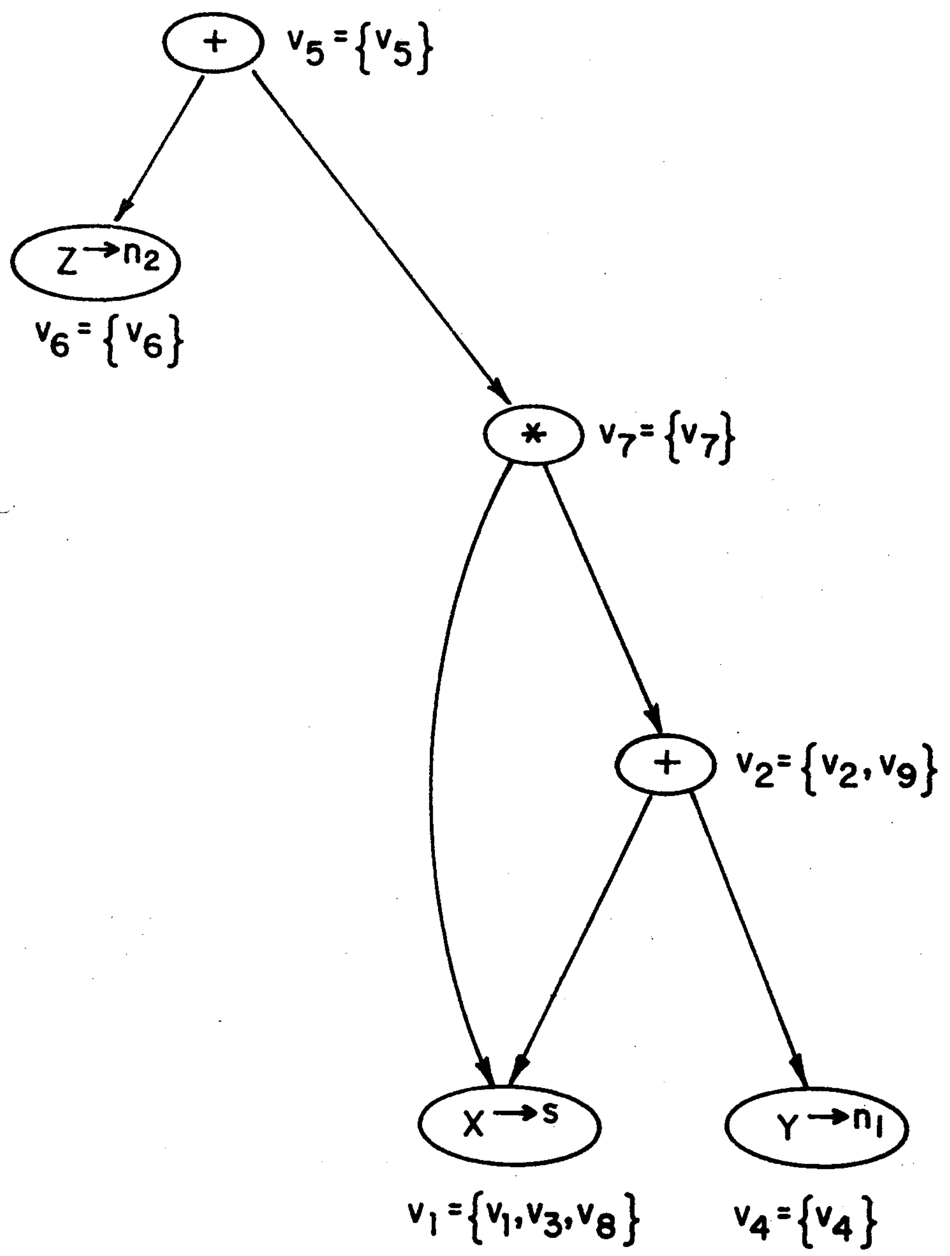


Figure 4.4. Dag representation of the simple cover.

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.

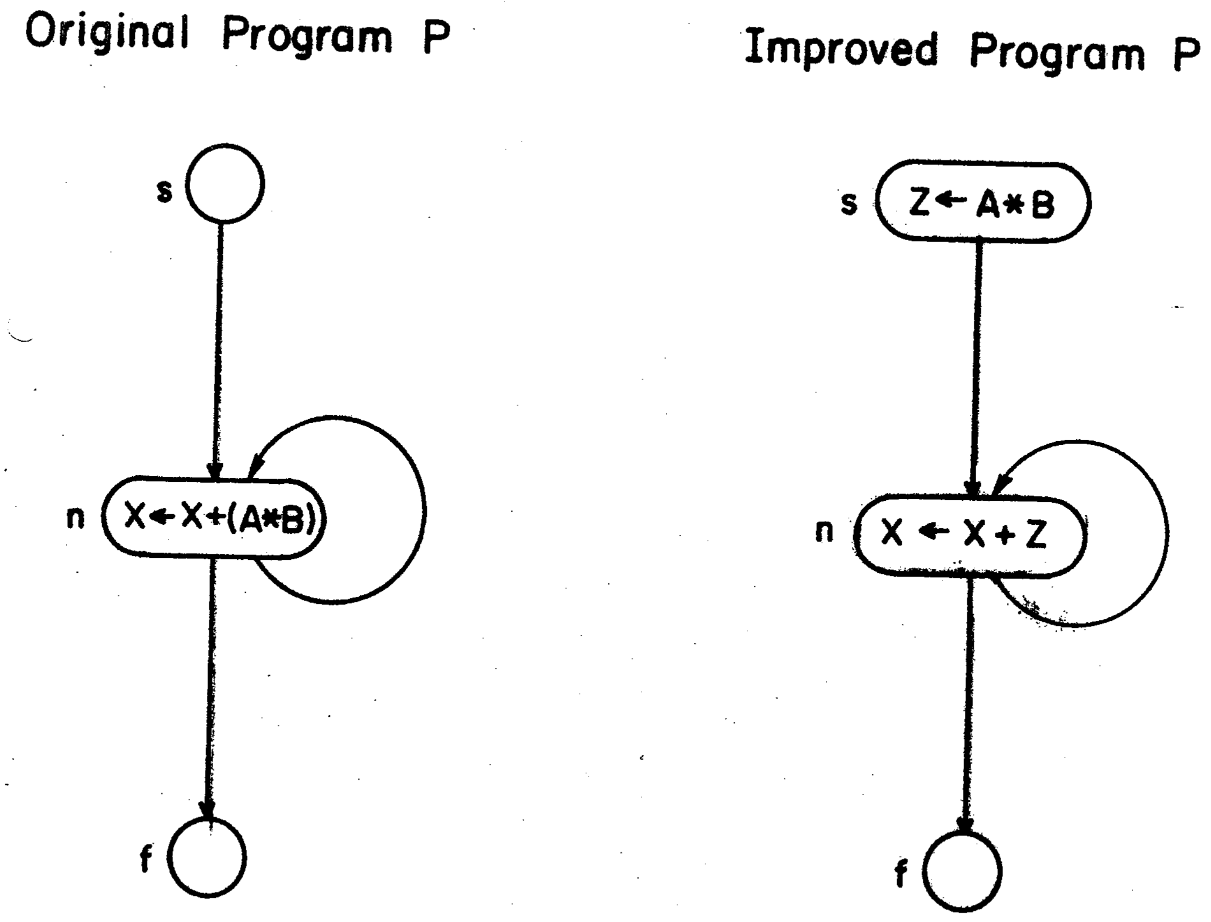


Figure 5.1. A simple example of code motion.

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:

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

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

<u>R3</u> The computation must be <u>safe</u> 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 <u>dominator chain</u>, from s to loc(t). The <u>birth point</u> 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 $\text{movept}_i(t)$ 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, $1 \leq i < k$.

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 $u \neq v$). 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 ϵ 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 $\ell$;
  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 $\in$ TEXT do
  L2: add t to EARLY(BIRTHPT(t)) and LATE(loc(t));
  for w := 1 to n do
  L3: EARLY(w) := EARLY(w)
              $\cup$ (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) $\cup$ LATE(u);
      comment Apply Lemma 5.3.1;
      for all t $\in$ EARLY(w) $\cap$ 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).

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

Proof. 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.

<u>Proof</u> 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. □

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

<u>Proof</u>. 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,

$$\text{movept}_2(t) = C1_F(C1_R(\text{loc}(t),\text{HPT}(t)),\text{loc}(t)).$$

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 $w = C2_F(\text{HPT}(t),\text{loc}(t))$ 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, $w = C2_F(\text{HPT}(t),\text{loc}(t)) = C1_R(\text{loc}(t),\text{HPT}(t))$. Hence, $\text{movept}_2(t) = C1_F(w,\text{loc}(t))$ is the first node in M_2^1 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,...,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 $\in$ (J(w)-I(w))-{w} so y is immediately dominated by some y' $\in$ 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 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($g(w,x)-1$),...,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 $\ell$ do add $x_i$ to BUCKET($w_i$);
  for w := n by -1 to 1 do
    begin
      for all x $\in$ BUCKET(w) do
        begin
          if FLAG(x) then
            $C1_G$(w,x) := the father of FIND(x) in DT;
          else $C1_G$(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 $\in$ I(w) do
                begin
                  z := FIND(y);
                  if NOT FLAG(z) then
                    begin
                  D:  for all x $\in$ IDOM$^{-1}$(z) do
                        if FLAG(x) then
                          UNION(FIND(x),w);
                      FLAG(z) := TRUE;
                    end;
                  if z $\neq$ w do UNION(z,w);
                end;
            end;
        end;
    end;
end;
```

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

Proof (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 ϵ 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 ϵ 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 ϵ 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•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. □

Lemma 5.6.2 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).

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

Corollary 5.6.1 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 $h(w,x)$.

Proof 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 ∈ 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 ∈ 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) $v \neq w$ 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 ∈ 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 $x \in V$, 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.

<u>Lemma</u> <u>5.8.1</u> If G is reducible and y ε V-{r} is contained of an IDOM(y)-avoiding cycle q, then y has an entering A-cycle edge.

<u>Proof</u> 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.

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

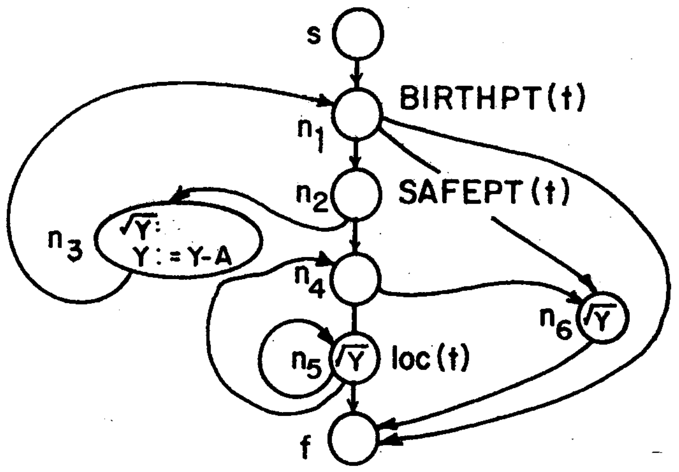
<u>Proof</u> Recall that PV(w-1) is defined to be derived from PV(w) by collapsing into w each set z containing at least one element y ε J(w)-{w}. Suppose there is a set z ∉ I(w) in PV(w) containing some y ε (J(w)-I(w))-{w}. Then, by definition of J(w), y is contained on a w-avoiding cycle q and IDOM(y) ε I(w). But since z ∉ 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.

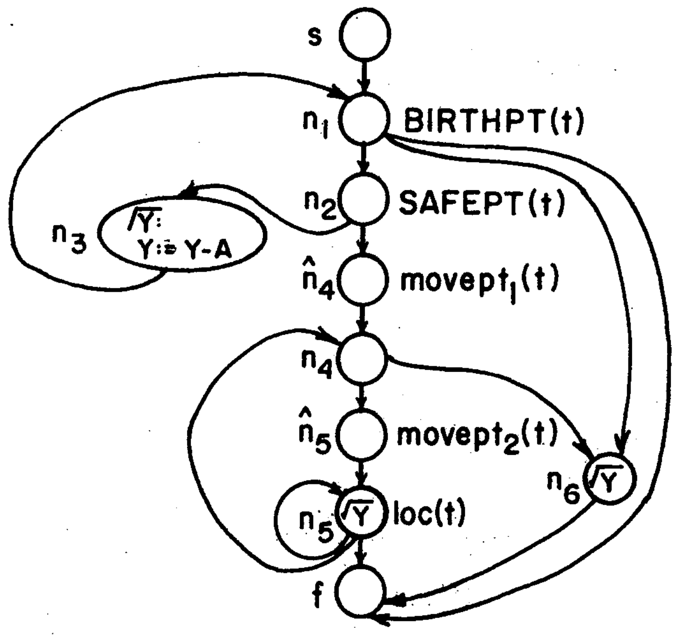
Theorem 5.8.2 If G is a reducible niche node and DDP(x) $\neq$ x, then K(x) = those nodes of the dominator chain from DDP(x) to IDOM(x).

Proof Suppose there exists some x $\in$ 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

<u>Figure 5.2</u>. Transformation of a flow graph F into a niche flow graph F'.

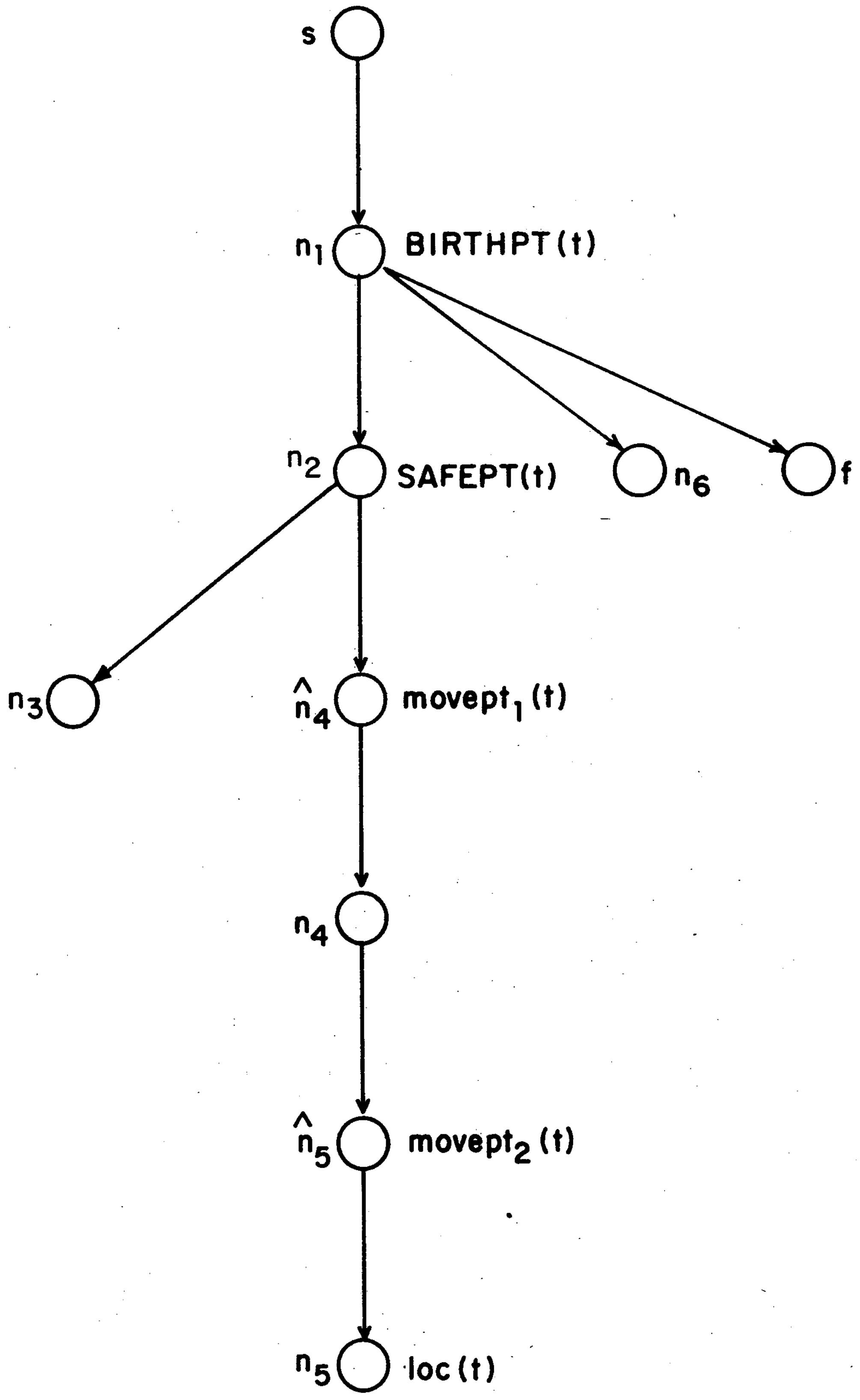


Figure 5.3. The dominator tree of the control flow grapl F'.

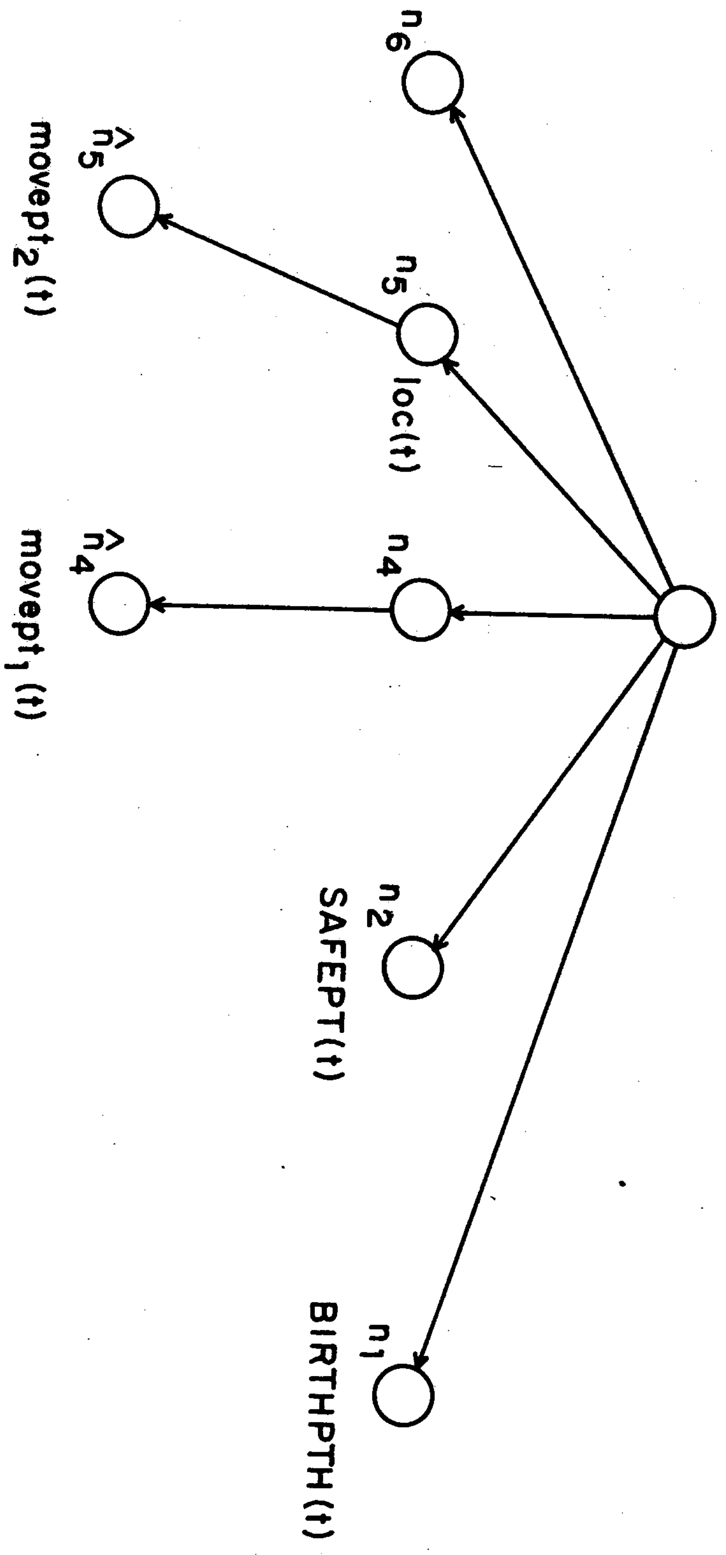


Figure 5.4. The dominator tree of the reverse of the control flow graph F'.

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).