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 Γ_{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.

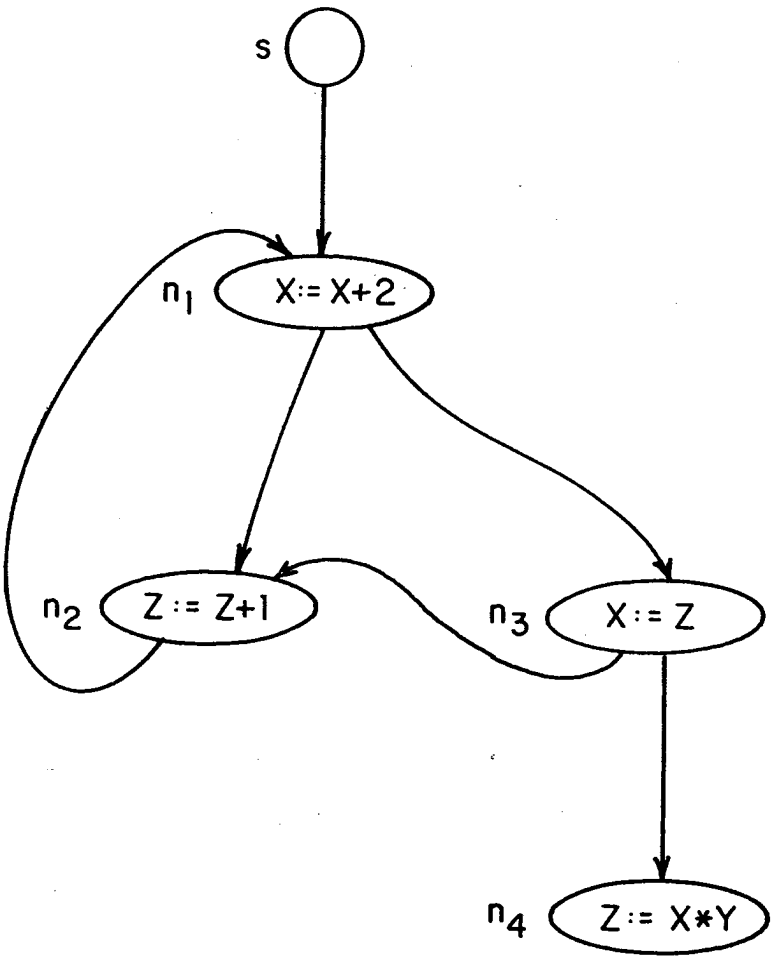

Figure 1.1 A control flow graph.

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.

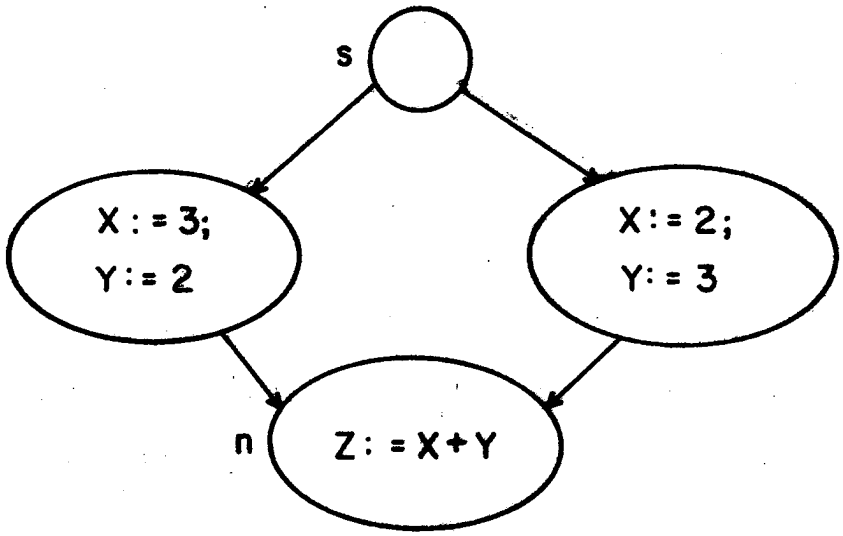

Figure 1.2. (From [KU2]) $Z^{n\rightarrow} = X^{\rightarrow n} + Y^{\rightarrow n}$ is a text expression which is covered by a constant sign but is not discovered by Kildall's algorithm.

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.

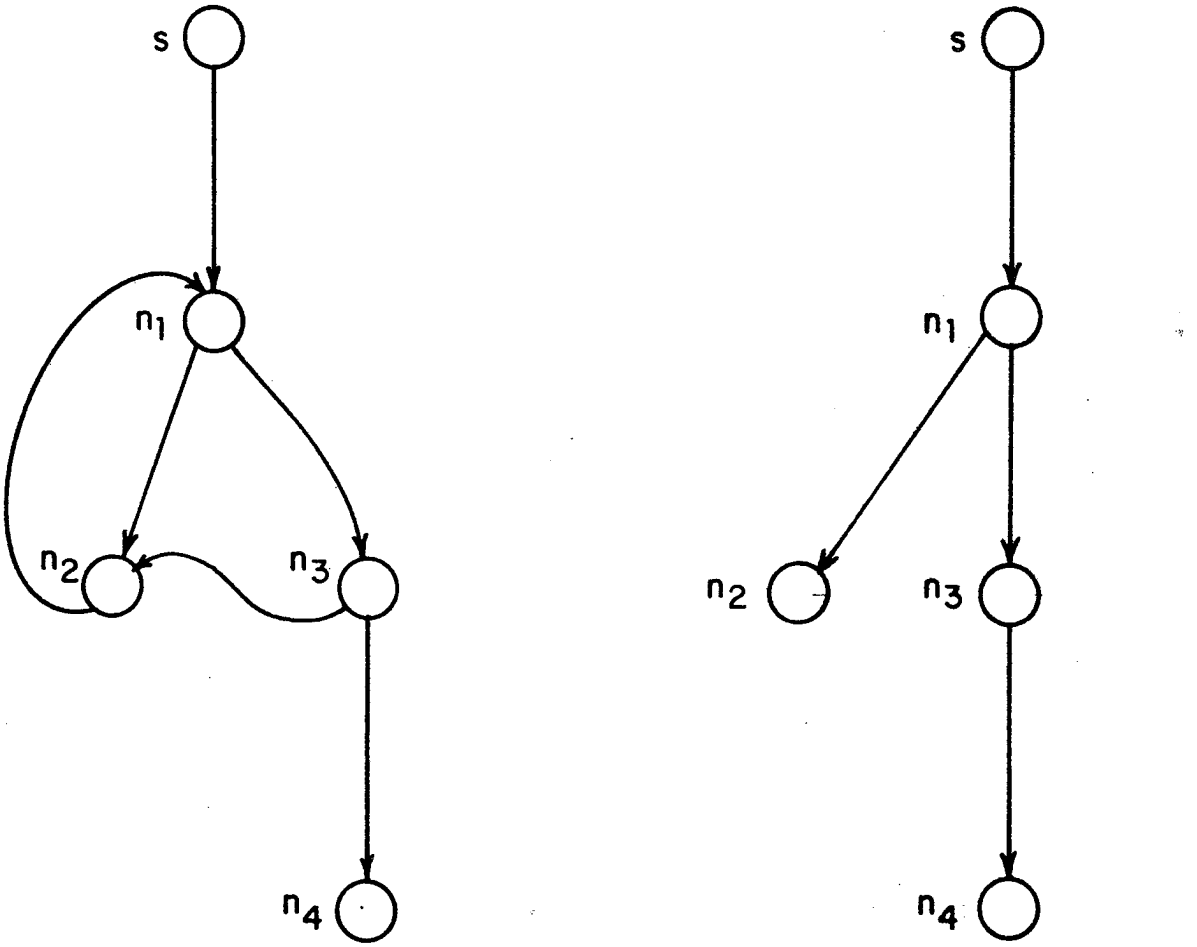

Figure 1.3. A flow graph and its dominator tree.

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 ≠ 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 ϵ 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 ϵ 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 ϵ 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 ϵ Σ and block n ϵ 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 $X^{\rightarrow s}$, 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 $X^{\rightarrow n}$ representing the value of program variable $X^{\rightarrow n}$ 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 $X^{n\rightarrow}$ 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 $Z^{n\rightarrow} = (X^{\rightarrow n}-1)*(Y^{\rightarrow n}+4)$ (or in the more proper prefix notation, (* (- $X^{\rightarrow n}$ 1) (+ $Y^{\rightarrow n}$ 4))) is the text expression associated with the string of text "X * Y" at the last

assignment statement of n.

An interpretation for the program P is an ordered pair (U, I). The universe 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 k-adic operator $I(\theta)$ 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 arithmetic domain 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 reduced form 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 reduced 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 global flow system 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 Π = (F, Σ, 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 $X^{m\rightarrow}$ 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

$$EXEC(t,p) = EXEC(\alpha,p)$$

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 $N(\alpha)$ 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, $\psi(t)$ 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 $EXEC(\alpha,p)$ contains $X^{\rightarrow n}$ but EXEC(t,p) does not, so $EXEC(\alpha,p) \neq EXEC(t,p)$, 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($\psi_1(t)$) $\overset{*}{\rightarrow}$ origin($\psi_2(t)$) 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 $X_0^f/(1+Q(X_1^f, \ldots, X_k^f)^2)$ located at block f. We show t is covered by a constant sign iff Q has no root in the natural numbers.

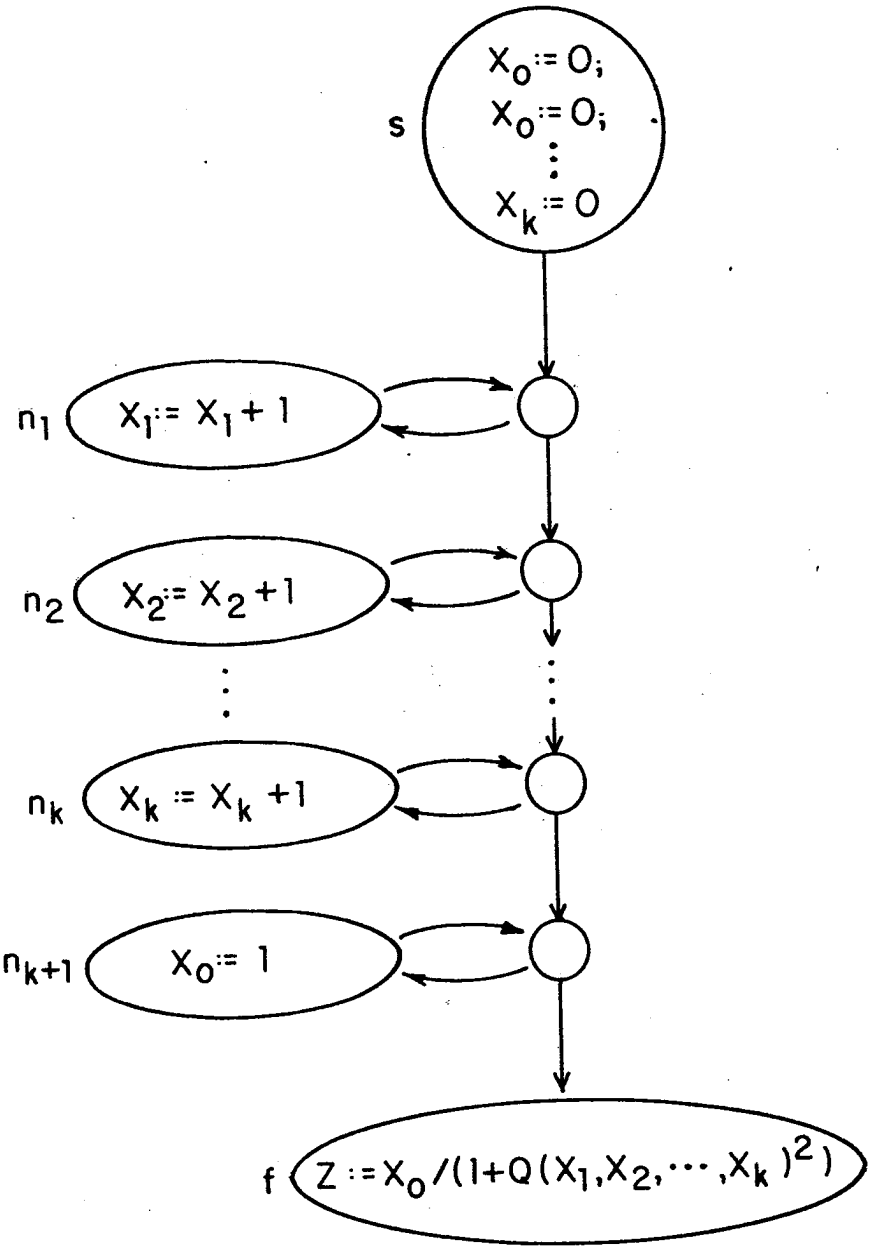

Figure 1.4. The control flow graph F_Q.

For any control path p from the start block s to the final block f and for i = 0,1,...,k let $X_i(p)$ = $I(EXEC(X_i^{\rightarrow f},p))$ = the value of X_i just on entry to f relative to p. Also, let $X(p) = (X_1(p),...,X_k(p))$. 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, $Q(X_1(p),...,X_k(p)) \neq 0$, 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 $t' = 1/Q(X_1^{\rightarrow f},\ldots,X_k^{\rightarrow f})$ 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 code motion 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:

$stm_1;...;stm_{i-1};stm_i;...stm_k$

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

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

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

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

where $\bar{X}_1,...,\bar{X}_k$ 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.