Optimal Code Generation for Expression Trees

A. V. Aho
S. C. Johnson

Bell Laboratories,
Murray Hill, New Jersey 07974

0. Summary

We discuss the problem of generating code for a wide class of machines, restricting ourselves to the computation of expression trees. After defining a broad class of machines and discussing the properties of optimal programs on these machines, we derive a necessary and sufficient condition which can be used to prove the optimality of any code generation algorithm for expression trees on this class. We then present a dynamic programming algorithm which produces optimal code for any machine in the class; this algorithm runs in time which is linearly proportional to the number of vertices in an expression tree.

1. Introduction

Code generation is an area of compiler design that has received relatively little theoretical attention. Bruno and Sethi [BS] show that generating optimal code is difficult, even if the target machine has only one register; specifically, they show that the problem of generating optimal code for straight-line sequences of assignment statements is NP-complete [C, Ka].

On the other hand, if we restrict the class of inputs to straight line programs with no common subexpressions, optimal code generation becomes considerably easier. Sethi and Ullman [SU], extending the work of Nakata [N] and Redziejowski [R], present an efficient code generation algorithm for a class of machines having n "general purpose" registers, symmetric register-to-register operations, but no complex addressing features such as indirection. They prove the optimality of their algorithm on expression trees using a variety of cost criteria, including output code length.

In generating code, the major issues are deciding what instructions to use, in what order to execute them, and which intermediate results to store in temporary memory locations. In general, there are an infinite number of programs which compute a given expression; even restricting ourselves to "reasonable" programs, the number of possible programs may grow exponentially with the size of the expression. Wasilew [W] has suggested an enumerative code generation algorithm which produces good code for expression trees not requiring temporary stores for their evaluation but which can take exponential time. We show that, by using dynamic programming, optimal code can always be generated for expression trees in linear time over a wide class of machine models, including the machines studied by Sethi and Ullman. The code produced has the interesting property that, unlike the Sethi-Ullman algorithm, it is not produced by a simple tree walk of the expression tree.

Additionally, we prove a theorem which characterizes any optimal code generation algorithm for a machine in our class. We also derive a number of results which characterize optimal programs, and prove that optimal programs can be written in a "normal form." This normal form theorem forms the basis of our dynamic programming algorithm.

2. Basic Definitions

In this section, we define expression trees, the machine model, and programs for the model machines.

2.1 Dags and Expression Trees

The sequence of assignment statements

$$\begin{aligned} X &\leftarrow A - C \\ X &\leftarrow X * X \\ Y &\leftarrow B - C \\ Y &\leftarrow Y * Y \\ Z &\leftarrow X + Y \end{aligned}$$

may be represented by the following labeled directed acyclic graph ("dag" for short).

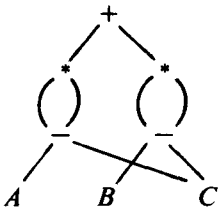


Fig. 1. A dag.

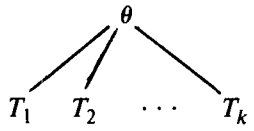
See [AU] for a formal correspondence between straight line programs and dags.

We assume the sequence of assignment statements represents a basic block of a source program. Our objective is to find algorithms that generate good code for dags. Unfortunately, the optimal code generation problem for dags is NP-complete, even for very simple machine models [BS].

For this reason, we shall consider only the special case where the dag is a tree; this occurs when there are no identified common subexpressions or operands. We shall see that, unlike for dags, optimal code generation is relatively easy for this class of inputs over a broad class of machine models.

In particular, we assume a countable set of operands Σ and a finite set of operators Θ. We define an *expression tree* as follows:

1. A single vertex labeled by a name from Σ is an expression tree.
2. If $T_1, T_2, \ldots, T_k$ are expression trees whose leaves all have distinct labels, and θ is a k-ary operator in Θ, then



is an expression tree. For example,

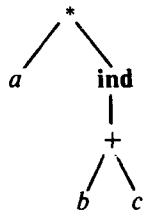


Fig. 2. An expression tree.

is an expression tree, assuming that + and * are binary operators, and **ind** is a unary operator.

If T is an expression tree and S is a subtree of T, then we define T/S as an expression tree which is obtained from T by replacing S by a single leaf labeled by a distinct name from Σ.

2.2 The Machine Model

We assume that we are generating code for a machine with n general purpose registers, and a countable sequence of memory locations. All memory locations are assumed interchangeable, as are all registers.

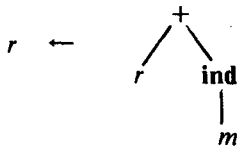
The actual machine operations are of two types:

(a) $r \leftarrow E$

(b) $m \leftarrow r$

The r in (a) and (b) refers to any of the n registers of the machine; the m refers to any memory location. The E is a dag containing operators from Θ, and leaves which are either registers or memory locations. It is assumed that if E contains an operator θ, then all operands of θ are also present. In addition, if E contains any registers, then the register on the left is assumed to be one of the registers in E.

A typical example of this type of instruction is



in which the contents of the memory location pointed to by m is added to register r. To avoid writing trees, we will usually write this in infix notation: $r \leftarrow r +$ (**ind** m). We assume that in each instruction of this type the dag E has a root that dominates all vertices in E. This root will be used to define the value of an instruction in a program.

For convenience, we also assume that every machine has a simple "load" instruction: $r \leftarrow m$. This instruction assigns the value currently in memory location m to the register r. The type (b) instruction is the inverse "store" instruction; it assigns the value currently in register r to memory location m.

We also assume that no instruction has any side effects; in particular, the only instruction to alter the state of memory is the store instruction.

In the type (a) instruction $r \leftarrow E$ we say r is *set* by the instruction. If z is a register or memory location in E, then the instruction is said to *use* z. In a type (b) instruction $m \leftarrow r$, m is set and r is used.

If I is an instruction, we shall use the notation **use** (I) and **set** (I) for the set of things used and the thing set, respectively.

2.3 Programs

A *machine program* consists of a set Σ_I of input memory locations and a finite sequence of instructions $P = I_1 I_2 \cdots I_q$. Each instruction I_t is of the form $r \leftarrow E$ or $m \leftarrow r$.

We define $v_t(z)$, the *value of z at time t*, $0 \leqslant t \leqslant q$, as a rooted dag. z is either a register or memory location. Initially, $v_0(z)$ is a single vertex labeled z if $z \in \Sigma_I$; otherwise $v_0(z)$ is undefined. For $t > 0$ we define $v_t(z)$ as follows:

a. If I_t is $r \leftarrow E$, then $v_t(r)$ is the dag obtained by taking the dag representing E and substituting for each leaf l in E the root of the dag representing $v_{t-1}(l)$. The resulting dag becomes the value of register r at time t. The root of E becomes the root of the new dag. We shall refer to the new dag as the *value of instruction* I_t. If the value of any leaf of E is undefined at time $t-1$, then $v_t(r)$ is undefined.

b. If I_t is $m \leftarrow r$, then $v_t(m)$ is $v_{t-1}(r)$. The dag $v_t(m)$ is said to be the value of I_t.

c. Otherwise, $v_t(z) = v_{t-1}(z)$.

$v(P)$, the *value of program* $P = I_1 I_2 \cdots I_q$, is defined to be the value of instruction I_q.

For example, the program

$$
\begin{aligned}
r_1 &\leftarrow b \\
r_1 &\leftarrow r_1 + c \\
r_2 &\leftarrow a \\
r_2 &\leftarrow r_2 \ ^* (\textbf{ind}\ r_1)
\end{aligned}
$$

(where **ind** is the indirection, or "contents of" operator) has as its value the tree of Fig. 2.

A program P is said to *compute* a dag D if $v(P) = D$. Two programs that compute the same dag are said to be *equivalent.*

Notice that we do not specify a particular register or memory location to hold the final value of a program. Since registers are assumed to be completely interchangeable, we can get a desired value into any register simply by renaming the registers throughout a program. Thus the issue of which register holds the answer is immaterial in our model.

Since the inputs to our code generation algorithms will always be expression trees, we will be interested only in machine programs that compute trees. As a consequence, no such program can use a machine instruction in which the same register or memory location is used more than once. In the next section we shall derive general conditions under which a machine program computes a tree rather than a dag.

3. Properties of Programs which Compute Expression Trees

In this section we give a simple characterization of programs which compute expression trees with no wasted instructions, and state some simple lemmas about the rearrangeability of these programs.

3.1 Useless Instructions

An instruction I_t in a program $P = I_1 I_2 \cdots I_q$ is said to be *useless in P* if the program $I_1 I_2 \cdots I_{t-1} I_{t+1} \cdots I_q$ is equivalent to P. Notice that the uselessness of an instruction depends on its context.

Any program can be reduced to an equivalent shorter program by eliminating some useless instruction. Repeating this finally yields an equivalent program with no useless instructions. Thus, we can freely assume that programs contain no useless instructions.

3.2 Scope of Instructions

Following [AU], we define the *scope* of an instruction I_t in a program $P = I_1 I_2 \cdots I_q$, $1 \leqslant t < q$, as the sequence of statements $I_{t+1} \cdots I_s$, where s is the largest index $t < s \leqslant q$, such that

a. $\textbf{set}(I_t) \in \textbf{use}(I_s)$

b. $\textbf{set}(I_t) \neq \textbf{set}(I_j)$, $t < j < s$.

We call s the *use* of t, and write $s = U(t)$. If there is no such s, the scope of I_t is undefined, and we write $U(t) = \infty$. Since U depends on the context P, we sometimes write U_P. If the scope of I_t is undefined, then I_t is useless. If the scope of I_t is defined, then we say that $\textbf{set}(I_t)$ is *active* immediately before every statement in its scope. We shall see that the maximum number of registers that are active at any one time in a program determines the number of registers needed to execute that program.

3.3 Programs which Compute Expression Trees

We may characterize programs which compute expression trees as follows.

If $P = I_1 \cdots I_q$ is a program with no useless instructions, then $v(P)$ is an expression tree if and only if

a. For all $1 \leqslant t < q$, there is exactly one instruction in the scope of I_t which uses $\textbf{set}(I_t)$.

b. For all X in Σ_I, there is at most one instruction which uses X before it is set.

3.4 Rearrangeability of Programs

One of the main problems in code generation is deciding the order in which an expression tree should be evaluated. Thus, it makes sense to ask which rearrangements of the instructions in a program produce an equivalent program.

For example, suppose P is a program in which statement I_t is $r_3 \leftarrow r_3 + X$. We can move I_t in front of statement I_{t-1} if I_{t-1} does not use r_3, or set r_3 or X. Similarly, we can move I_t after statement I_{t+1} if I_{t+1} does not use r_3, or set r_3 or X. If P computes an expression tree, there is a particularly simple characterization of what rearrangements of P are possible without changing the value of P.

Theorem 3.1. (Rearrangement Theorem) Let $P = I_1 I_2 \cdots I_q$ be a program which computes an expression tree and has no useless instructions. Let π be a permutation on $\{1, \ldots, q\}$ with $\pi(q) = q$. Then the program $Q = I_{\pi(1)} \cdots I_{\pi(q)}$ is equivalent to P if and only if

$$\pi(U_P(t)) = U_Q(\pi(t)),$$

for all t, $1 \leqslant t < q$.

Proof. The proof is a straightforward induction on t, based on the definition of value. The details are left to the reader. □

Another important aspect of code generation is determining the number of registers needed to compute an expression tree. We shall see that certain rearrangements of a program may require more registers to be used than other equivalent rearrangements.

Let $P = I_1 \cdots I_q$ be a program with no useless instructions; for each I_t we define the *width* of I_t to be the number of distinct j, $1 \leqslant j \leqslant t$, with $U(j) > t$ and I_j not a store instruction. We define the *width of P* to be the maximum over t, $1 \leqslant t < q$, of the width of I_t.

Intuitively, the width of a program is the maximum number of registers that are active at any one time. It should be clear that eliminating useless instructions from a program can never increase its width.

In our machine model, any one register is indistinguishable from any other. Thus, if at most w registers are active at any time, by renaming registers we can cause any specified set of w registers to be the only ones used.

Lemma 3.2. Let P be a program of width w, and let R be a set of w distinct registers. Then, by renaming the registers used by the instructions of P, we may construct an equivalent program P', with the same number of instructions as P, which uses only registers in R.

Proof. Let $P = I_1 I_2 \cdots I_q$. We shall rewrite P into an equivalent program $Q = J_1 J_2 \cdots J_q$, such that Q uses only registers in R. Each instruction J_t will be equal to I_t with relabeled registers. Provided that the relabeling is consistent, i. e., when a set variable is relabeled its use is also relabeled, P and Q will be equivalent if and only if $U_P(t) = U_Q(t)$, for all t. For a relabeling of P to have this property, it is enough to specify the relabeling on those instructions of the form $r \leftarrow E$ (including load instructions) where E contains no register leaves; in the other cases, the register which is set is forced to be one of the (relabeled) registers of E, by consistency. If I_t is an instruction which uses no registers, then the relabeled register which is set by J_t should not be active in Q right after time $t-1$, since otherwise U_Q will be different from U_P. But, there are at most $w-1$ active registers in P immediately prior to time t (since the width of I_t in P is at most w), and thus this is so in Q; this implies that there is some register in R which is not active immediately prior to time t. We can choose some such register as the register set by J_t. Continuing in this way, we may relabel P using only registers from R, and thus get an equivalent program Q. □

Lemma 3.2 justifies calling a program of width w a program which *uses w registers.*

The following theorem identifies a class of equivalent rearrangements of a program with no useless instructions. At least one of these rearrangements uses the fewest number of registers of any equivalent rearrangement.

Theorem 3.2. (Rearrangement Theorem) Let $P = I_1 \cdots I_q$ be a program of width w with no stores and no useless instructions. Suppose I_q uses k registers whose values at time $q-1$ are $A_1, \cdots, A_k$. Then, there exists an equivalent program $Q = J_1 \cdots J_q$ and a permutation π on $\{1, \ldots, k\}$ such that:

1. Q has width at most w,
2. Q can be written as $P_1 P_2 \cdots P_k J_q$, where $v(P_i) = A_{\pi(i)}$ for $1 \leqslant i \leqslant k$, and the width of P_i, by itself, is at most $w - i + 1$.

Proof. Since P has width w, we may assume that P uses at most w registers, say $1, 2, \ldots, w$. Examine the value computed by I_1. Since P has no useless instructions, this value must be a subtree of a unique A_j; thus $\pi(1) = j$. Let P_1 be made

up of the set of instructions whose values are subtrees of A_j, in the order in which they appear in P. It is easy to see that $v(P_1) = A_j$. Since the instructions in P_1 use at most w registers, the width of P_1 is at most w. By relabeling the registers of P_1 we may assume that it computes its value in register w.

Now, consider the remaining instructions in P, not in P_1. These, taken in order, compute the other A values, and then execute I_q. However, the width of this sequence is at most $w-1$, since at each stage in the program there is at least one register devoted to a value which is a subtree of A_j. Thus, we may relabel these instructions to use only the registers $1, 2, \ldots, w-1$. We may then repeat the above construction to obtain P_2, of width at most $w-1$, which computes some $A_{\pi(2)}$, and so on. We are left with the single instruction I_q, and $A_{\pi(1)}, \ldots, A_{\pi(k)}$ computed in registers $w, w-1, \ldots, w-k+1$, respectively. The final instruction J_q is the obvious relabeling of I_q to reflect the relabeling of the registers. The job of proving that this relabeled program is equivalent to P is left to the reader. □

A program is said to be *strongly contiguous* if it satisfies Theorem 3.2 and each P_i is itself strongly contiguous. Using the transformation in the proof of Theorem 3.2 recursively, we can easily prove:

Theorem 3.3. (Strong Rearrangment Theorem) Every program without stores can be transformed into an equivalent strongly contiguous program.

4. Optimal Code Generation Algorithms

This section defines our criterion of program optimality and shows that every optimal program can be put into a normal form. This normal form theorem forms the foundation of our dynamic programming algorithm. A necessary and sufficient condition for an algorithm to generate optimal code follows.

A code generation algorithm A is a mapping from expression trees to programs. We shall usually denote the program produced by algorithm A for expression tree T as $A(T)$. A code generation algorithm is *correct* if $v(A(T)) = T$ for all T.

We shall judge programs by their length. A program P is optimal for T if P is as short as any program that computes T. Clearly, an optimal program for T has no useless instructions.

A code generation algorithm is said to be *optimal* if $A(T)$ is an optimal program for every input T. Our goal is a necessary and sufficient condition for a correct code generation algorithm to be optimal.

4.1 Normal Forms

We begin by showing that every program can be transformed into an equivalent, normal form, program that first computes an expression tree T_1, stores it into a memory location m_1, then computes an expression tree T_2, stores it into a memory location m_2, and so on, and finally combines these expression trees into a resulting expression tree T. The final computation of T proceeds without stores, using the memory locations m_1, m_2, etc., as input variables.

Let $P = I_1 \cdots I_q$ be a machine program with no useless instructions. We say P is in *normal form* if it can be written as

$$P = P_1 J_1 P_2 J_2 \cdots P_{s-1} J_{s-1} P_s$$

such that

1. Each J_i is a store instruction, and no P_i contains a store instruction.
2. No registers are active immediately after each store instruction.

Lemma 4.1. Let P be an optimal program which computes an expression tree. Then there exists a permutation of P which computes the same value and is in normal form.

Proof. We can assume, without loss of generality, that each store instruction in P refers to a distinct memory location. Let I_f be the first store instruction in P. We can determine all statements in $I_1 \cdots I_f$ that are not used in computing any part of $v(I_f)$. We can move these statements *en masse* immediately after statement I_f. We now have a program of the form $P_1 J_1 Q_1$ such that $J_1 = I_f$ and the scope of every statement in P_1 does not extend beyond I_f. Since I_f sets a unique memory location, no statement in Q_1 can reset that memory location. Thus, $v(Q_1) = v(P)$.

We may repeat this same process on Q_1; continuing in this way we obtain the normal form. □

We combine the concepts of strong contiguity and normal form in the following definition and theorem.

Let $P = P_1 J_1 \cdots P_{s-1} J_{s-1} P_s$ be a program in normal form. We say that it is in *strong normal form* if each P_i is strongly contiguous.

Theorem 4.1. (Strong Normal Form Theorem) Let P be an optimal program of width w. We can transform P into an equivalent program Q such that

1. P and Q have the same length,
2. Q has width at most w, and
3. Q is in strong normal form.

Proof. We can apply Lemma 4.1 to P to transform it into normal form. We can then apply Theorem 3.2 recursively to each P_i in the resulting program to make each P_i strongly contiguous. This satisfies condition (3). Conditions (1) and (2) follow since these two transformations do not change the length of a program and never increase the width. □

4.2 The Optimality Theorem

We can now derive a theorem which gives a necessary and sufficient condition that a code generation algorithm be optimal.

Let A be a correct code generation algorithm. Let $c(T)$ be the cost (number of instructions) of $A(T)$ for any expression tree T. Let $\hat{c}(T)$ be the cost of an optimal program that computes T. Then, clearly, A is optimal if and only if $c(T) = \hat{c}(T)$ for all T.

Theorem 4.2. A is an optimal code generation algorithm if and only if

1. $c(T) = \hat{c}(T)$ for all trees T for which there exists an optimal program with no stores, and
2. $c(T) \leqslant c(S) + c(T/S) + 1$, for each T and every subtree S of T.

Proof. Let T be a tree which has an optimal program with k stores. We shall show by induction on k that if A is a code generation algorithm that satisfies conditions (1) and (2), then $A(T)$ is an optimal program.

Basis. $k = 0$. By condition (1), $c(T) = \hat{c}(T)$.

Inductive Step. Suppose the inductive statement is true for all trees for which there exists an optimal program computing the tree with $k-1$ or fewer stores. Let T be a tree for which an optimal program P exists with k stores. We may assume that P is in strong normal form.

Let $P = P_1 I P_2$, where $I = m_i \leftarrow r_j$ is the first store instruction in P. We may consider P_1 as computing a subtree S of T into register r_j, I as storing this value, and P_2 as then computing a value equivalent to T/S.

Since P is optimal, we can conclude that P_1 is an optimal program for S and P_2 is an optimal program for T/S. Therefore, $\hat{c}(T) = \hat{c}(S) + 1 + \hat{c}(T/S)$, the 1 taking into account the cost of the store instruction I.

Let us now consider $A(T)$, the program produced by A. Since S can be evaluated optimally with no stores (e.g., by P_1), condition (1) implies that A evaluates S optimally. Since T/S can be evaluated optimally with $k-1$ stores (e.g., by P_2), the inductive hypothesis implies that A evaluates T/S optimally. From condition (2) we have

$$\begin{aligned} c(T) &\leqslant c(S) + c(T/S) + 1 \\ &= \hat{c}(S) + \hat{c}(T/S) + 1 \\ &= \hat{c}(T) \end{aligned}$$

Therefore A evaluates T optimally.

Conversely, suppose A is optimal; then (1) is immediate. To prove (2), we note that, given any program P_1 for S and any program P_2 for T/S, we may construct a program $P_1 I P_2$ which computes T, where I stores the result of P_1 into the added memory cell in T/S. Thus, we obtain from $A(S)$ and $A(T/S)$ a program of cost $c(S) + c(T/S) + 1$ which computes T; since A is optimal, this is no better than $c(T)$, *i.e.*, $c(T) \leqslant c(S) + c(T/S) + 1$. □

5. The Dynamic Programming Algorithm

This section presents a dynamic programming algorithm which produces an optimal program to compute a given expression tree; the algorithm can be applied to produce programs for any machine in the class described in Section 2. The running time of the algorithm is linear in the number of vertices in the expression tree being compiled. Before stating the algorithm, we will need some preliminary results.

5.1 Covering

Suppose $P = I_1 \cdots I_q$ is a program with no useless instructions that computes an expression tree T, and suppose the last instruction is of the form $r \leftarrow E$. Then the roots of T and E must correspond. In fact, all of the non-leaf vertices of E must match corresponding vertices in T. Moreover, for each leaf of E which is a register, there is a subtree of T computed into that register. For each leaf of E which is a memory location, there is either a leaf of T or another subtree of T which has been computed and stored in that memory location.

For example, consider the following program P:

$$
\begin{aligned}
r_1 &\leftarrow d \\
r_1 &\leftarrow r_1 + e \\
m_1 &\leftarrow r_1 \\
r_1 &\leftarrow b \\
r_1 &\leftarrow r_1 + c \\
r_1 &\leftarrow r_1 * (\textbf{ind}\ m_1) \\
m_2 &\leftarrow r_1 \\
r_1 &\leftarrow a \\
r_1 &\leftarrow r_1 - m_2
\end{aligned}
$$

P computes the expression tree in Fig. 4.

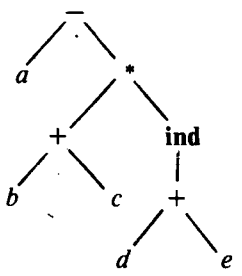


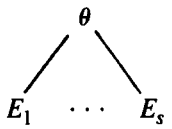
Fig. 4. Expression tree.

The last instruction uses register r_1 and memory location m_2. At that time, register r_1 contains as value a subtree consisting of the single operand a; memory location m_2 contains the subtree dominated by the vertex labeled *. We say that the instruction $r \leftarrow r - m$ *covers* the expression tree in Fig. 4. Given an instruction covering a tree S, we can identify subtrees of S which must be computed into registers, or into memory, in order that the instruction produce S as its value.

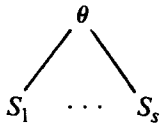
More formally, we shall define an algorithm *cover*(E, S) which will return **true** or **false** depending as E covers S. S is taken to be an expression tree, and E a tree on the right side of an instruction (see Section 2.2). If E covers S, *cover* will place into two (initially empty) sets, *regset* and *memset*, the subtrees which need to be computed in registers and memory, respectively.

algorithm *cover*(E, S):

1. If E is a single register vertex, add S to *regset* and return **true**.
2. If E is a single memory vertex, add S to *memset* and return **true**.
3. If E has the form



then, if the root of S is not θ, return **false**. Otherwise, write S as



Then, for all i from 1 to s, recursively invoke *cover*(E_i, S_i). If any such invocation returns **false**, then return **false**. Otherwise, return **true**. □

If E covers S, we shall denote the resulting sets *regset* and *memset* by $\{S_1, \ldots, S_k\}$ and $\{T_1, \ldots, T_l\}$, respectively.

We say that an instruction $r \leftarrow E$ *covers* an expression tree S if *cover*(E, S) is **true**. Note, as a boundary condition, that the load instruction covers all trees, with $k=0$, $l=1$, and $T_1 = S$. By comparison, the store instruction covers all trees with $k=1$, $l=0$, and $S_1 = S$.

Lemma 5.1. If P is a program that computes an expression tree T, the last instruction of P covers T.

Proof. Straightforward, from the definition of value. □

5.2 The Algorithm

We shall associate with the root of each subtree S of T an array of costs $c_j(S)$ for $0 \leqslant j \leqslant n$. Here, as before, n is the number of registers in the machine. For all such j and S, $c_j(S)$ is an integer or ∞; $c_j(S)$ gives the minimal number of instructions needed to compute S by a program in strong normal form

$$Q = Q_1 J_1 \cdots Q_{s-1} J_{s-1} Q_s$$

in which the width of Q_s is at most j. If $j = 0$, we shall interpret this to mean the minimal cost of computing S into a memory location. Thus, if S is a leaf of T, we set $c_0(S) = 0$. Clearly, $c_j(S)$ is a nonincreasing function of j for $1 \leqslant j \leqslant n$.

The dynamic programming algorithm consists of three phases. In the first phase, we compute the $c_j(S)$ values for all subtrees S and all values of j, $0 \leqslant j \leqslant n$. In the second phase, we use the values of the c_j arrays to traverse T and mark a sequence of vertices. These vertices represent subtrees of T that must be computed into memory locations either because there are not enough registers available or because certain machine instructions require some operands to be in memory. Thus, after the second phase the required number of stores is known, as well as which subcomputations of T will be stored in temporary variables. In the final phase, we generate code for the each of the subcomputations. The result of the third phase is a program of op-

timal length computing T.

We shall now describe the algorithm for a particular machine with n registers and a fixed set of instructions. As input, we are given an expression tree T for which we are to construct an optimal program.

Phase 1. Computing $c_j(S)$ for each subtree S of T.

Initially, we set $c_j(S) = \infty$ for all subtrees S of T, and all j with $0 \leqslant j \leqslant n$. We then visit each vertex of T in postorder.* At each vertex, we compute the $c_j(S)$ array for the subtree dominated by that vertex as follows:

a. If S consists of a leaf, set $c_0(S) = 0$.

b. Loop over all instructions $r \leftarrow E$ which cover S. For each such instruction obtain from $cover(E, S)$ the subtrees $S_1, \ldots, S_k$ which must be computed into registers, and the subtrees $T_1, \ldots, T_l$ which must be computed into memory. Then, for each permutation π of $\{1, 2, \ldots, k\}$ and for all j, $k \leqslant j \leqslant n$, compute

$$c_j(S) = \min\left(c_j(S), \sum_{i=1}^{k} c_{j-i+1}(S_{\pi(i)}) + \sum_{i=1}^{l} c_0(T_i) + 1\right)$$

In the last expression the first term represents the cost of computing the S_i subtrees into registers in the order $S_{\pi(1)} \cdots S_{\pi(k)}$, using j registers for the first, $j-1$ for the second, etc. The second term is the cost of computing the subtrees of S which must be in memory. The third term, 1, represents the cost of executing the instruction $r \leftarrow E$.

c. Having done step b for all instructions which cover S, set

$$c_0(S) = \min(c_0(S), c_n(S) + 1)$$

and, for $1 \leqslant j \leqslant n$

$$c_j(S) = \min(c_j(S), c_0(S) + 1)$$

The first equation represents a computation of S into memory by initially computing S into a register r and then storing it with a store instruction. The second equation represents the possibility that we can compute S when j registers are available by computing it at some earlier time into memory, and then simply loading it. □

By using a postorder traversal the c_j arrays are computed for all subtrees of S before c_j is computed for S itself. After finishing Phase 1, we have c_j defined over the entire tree T. As is usual in dynamic programming, we may either save at each vertex a choice of an instruction and a permutation π which attain the minimum cost for each j, or make another pass over the tree and find an instruction and permutation π which attain the minimum cost when we need it. In any case, we shall assume that an optimal instruction and permutation π are known for each $c_j(S)$. In particular, the optimal instructions associated with the two equations in step (c), above, are the store and load instructions, respectively. Note that the optimal instruction associated with $c_j(S)$ covers S, for all j.

Phase 2. Determining the Subtrees to be Stored.

Phase 2 walks over the tree T, making use of the $c_j(S)$ arrays computed in Phase 1 to create a sequence of vertices $x_1, \ldots, x_s$ of T. These vertices are the roots of those subtrees of T which must be computed into memory; x_s is the root of T. These subtrees are so ordered that for all i, x_i never dominates any x_j for $j > i$. Thus if one subtree S requires some of its subtrees to be computed into memory, these subtrees will be evaluated before S is evaluated.

Phase 2 consists of calling the algorithm $mark(T, n)$, defined below, to mark $x_1, x_2, \cdots, x_{s-1}$. The variable s, representing the number of vertices marked, is initially set to 0. Upon returning from mark, we increment s and set x_s to the root of T, completing Phase 2.

algorithm $mark(S, j)$:

a. Let $z \leftarrow E$ be the optimal instruction associated with $c_j(S)$, and π the optimal permutation. Invoke $cover(E, S)$ to obtain the subtrees $S_1, \ldots, S_k$ and $T_1, \ldots, T_l$ of S.

b. For all i from 1 to k, recursively invoke $mark(S_{\pi(i)}, j-i+1)$.

c. For all i from 1 to l, recursively invoke $mark(T_i, 0)$.

d. If j is 0, and the instruction $z \leftarrow E$ is a store, increment s and set x_s equal to the root of S.

e. Return.

Notice that the marking is done after all the descendent subtrees have been visited; this

---

* A postorder traversal of a tree T with root r having subtrees $T_1, T_2, \cdots, T_k$ is defined recursively as follows:
1. Traverse each of $T_1, T_2, \cdots T_k$ in postorder.
2. Traverse r.
See [Kn] or [AHU] for more details.

will ensure that when we compute a subtree starting at one of the x_i vertices, all subcomputations which must be stored into memory will have been completed.

Phase 3. Code Generation.

This phase makes a series of walks over subtrees of T, generating code. These walks start at the vertices $x_1, \ldots, x_s$ computed in Phase 2. After each walk except the last, we generate a store into a distinct temporary memory location m_i, and rewrite vertex x_i to make it behave like an input variable m_i in later walks that might encounter it.

The walking and code generation is is done by a routine *code*(S, j), which generates code for a subtree S using the integer parameter j to control the walk. The unspecified routine *alloc* allocates registers from the set of registers which are available (initially, all n of them), and the routine *free* puts a register back into the allocatable set. *code*(S, j) never requires more than j free registers and returns a register α whose value upon return is $v(S)$.

Thus, the outermost flow of control in Phase 3 is:

a. Set $i=1$ and invoke *code*(x_i, n). Let α be the register returned by this invocation. Then, generate the instruction $m_i \leftarrow \alpha$, invoke *free*(α), and rewrite the vertex x_i to make it represent memory location m_i. Repeat this step for $i = 2, \ldots, s-1$.

b. Finally, invoke *code*(x_s, n); the register returned contains the value of T. (Recall that x_s is the root of T.) □

It remains only to describe the algorithm *code*:

algorithm *code*(S, j):

a. Let $z \leftarrow E$ be the optimal instruction for $c_j(S)$, and π the optimal permutation. Invoke *code*(E, S) to obtain the subtrees $S_1, \ldots, S_k$ which must be computed into registers.

b. For i from 1 to k, recursively invoke *code*$(S_{\pi(i)}, j-i+1)$. Let $\alpha_1, \alpha_2, \cdots, \alpha_k$ be the registers returned.

c. If k is nonzero, the register α returned by *code*(S, j) must be one of the registers α_i. If k is zero, call the procedure *alloc* to obtain an unused register to return.

d. Issue the instruction $\alpha \leftarrow E$ with registers $\alpha_1, \alpha_2, \cdots, \alpha_k$ substituted for the registers used by E. Any memory locations used by E either will be leaves of T, or will have been precomputed by earlier calls to *code*; in either case, they can be immediately substituted into E.

e. Finally, call *free* on $\alpha_1, \alpha_2, , \cdots, \alpha_k$ except α. Return α as the value of *code*(S, j). □

5.3 Properties of the Algorithm

We now address ourselves to the correctness, optimality, and time complexity of the dynamic programming algorithm. Our proofs are necessarily sketchy, but hopefully suggestive enough to enable the mathematically fastidious reader to fill in the details.

It is relatively simple to establish the correctness of the algorithm. To begin, we must show that each phase terminates and produces the appropriate answer. Showing that the value of the program P produced by Phase 3 is T has two major aspects. We must show that all memory locations used by each instruction in P have their values previously computed and that the register allocation routine *alloc* never runs out of registers. This last fact can be established by showing that *code*(S, j) never requires more than j nonactive registers, an easy inductive argument. We can then show by induction on T that P actually computes T as its value and has $c_n(T)$ instructions in it.

Proving that $c_n(T)$ instructions is optimal is more subtle, and resembles the proof of Theorem 4.2. As is frequently the case in mathematics, it is easier to prove something somewhat stronger.

Theorem 5.1. $c_j(T)$ is the minimal cost over all strong normal form programs

$$P_1 J_1 \cdots P_{s-1} J_{s-1} P_s$$

which compute T such that the width of P_s is at most j.

Proof. By Theorem 2.2 the optimality of $c_n(T)$ over programs in strong normal form implies optimality over all programs. The proof will be by induction on the number of vertices of T. The basis is trivial. For the inductive step suppose T is some tree such that the theorem is true for all trees with a smaller number of vertices. Let P, as above, be an optimal strong normal form program computing T, with the width of P_s less than or equal to j.

When $j > 0$, we argue as follows. Since P_s is strongly contiguous, we may write it as

$$P_s = P_{s1} \cdots P_{sk} I$$

where P_{s1} through P_{sk} are strongly contiguous programs computing into registers the values $S_{\pi(1)}, \ldots, S_{\pi(k)}$ needed by I. Moreover, each

P_{si}, by itself, has width at most $j-i+1$.

Each of the earlier code segments $P_t I_t$ is involved in the computation of some value, either for use by some P_{si} or use as one of the values $T_1, \ldots, T_l$ required in memory by instruction I. We associate all of the segments either with the associated $S_{\pi(i)}$ or with the associated T_i. This yields k strong normal form programs which compute the $S_{\pi(i)}$; by induction, the cost of each is at least $c_{j-i+1}(S_{\pi(i)})$. Moreover, there are up to l nontrivial strong normal form programs which compute those T_i values which are not leaves of T. The cost of each of these is, again by induction, at least $c_0(T_i)$.

Thus, we have established that the cost of this optimal program is at least

$$1 + \sum_{i=1}^{k} c_{j-i+1}(S_{\pi(i)}) + \sum_{i=1}^{l} c_0(T_i)$$

which, by Phase 1, is always greater than or equal to $c_j(T)$. Thus, $c_j(T)$ is minimal.

The argument when j is 0 is similar. We remove the final instruction of P, which must be a store instruction; we then argue as above. □

The time complexity of the dynamic programming algorithm is easy to calculate. Phase 1 requires am time, where m is the number of vertices in T. The constant a depends on the size of the instruction set of the machine, the "aryness" of the instructions, and the number of registers in the machine. If a machine has a large number of identical registers, then it may be more economical to store only the distinct entries of the c_j arrays at each vertex of T using linked lists. Phases 2 and 3 each require time proportional to the number of vertices in T; the time taken to process each vertex is bounded by a constant. Thus, the entire algorithm is $O(m)$ in time complexity.

6. Conclusions

We have given an algorithm to compile optimal code for expression trees over a broad class of machines. For simple machine architectures it is possible (and desirable) to specialize the algorithm to the specific machine to reduce the amount of information that needs to be collected by Phase 1 at each vertex of the expression tree being compiled. For example, for the register machines studied by Sethi and Ullman a single integer is sufficient to characterize the difficulty of computing a subtree [SU]. Bruno and Lassagne give a similar characterization for a class of stack machines [BL]. We can derive a similar algorithm for machines with an indirection operator but no register to register operations, such as the Honeywell 6000 or IBM 7090 machines.

There are a number of directions in which the dynamic programming algorithm can be generalized. Throughout this paper we have used program length as the criterion of optimality. However, the algorithm will work with any additive cost function (i.e., for a program PQ, $c(PQ) \geq c(P) + c(Q)$), such as execution time or number of stores. The algorithm can also be extended to handle instructions which compute values directly into memory. The algorithm is capable of taking into account certain algebraic properties of operators, e.g., commutativity; however, it is not obvious how to extend the algorithm efficiently to all common algebraic properties, except for particular machine models as in [B, BL, SU].

We now have a situation where, over a wide range of machines, code generation for trees is linear in difficulty, while for even very simple machines code generation for dags is NP-complete. The question of how many subexpressions can be tolerated and in what form they can appear before the problem becomes hard merits further study. The results in [BS, S, and WS] are useful in this regard.

References

[AHU] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[AU] Aho, A. V., and Ullman, J. D., "Optimization of Straight Line Code," *SIAM J. Computing* **1**:1 (1972), 1-19.

[B] Beatty, J. C., "An Axiomatic Approach to Code Optimization for Expressions," *J. ACM* **19**:4 (1972), 613-640.

[BL] Bruno, J., and Lassagne, T., "The Generation of Optimal Code for Stack Machines," To appear, *J. ACM*.

[BS] Bruno, J., and Sethi, R., "Register Allocation for a One-Register Machine," *Technical Report* No. 157, Computer Science Dept., Pennsylvania State University, October 3, 1974.

[C] Cook, S. A., "The Complexity of Theorem Proving Procedures," *Proc. 3rd Annual ACM Symposium on Theory of Computing* (May 1971), 151-158.

[Ka] Karp, R. M., "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations,* R. E. Miller and J. W. Thatcher (eds.), Plenum Press, New York (1972), 85-103.

[Kn] Knuth, D. E., *Fundamental Algorithms,* second edition, The Art of Computer Programming **1**, Addison-Wesley, Reading, Mass., 1973.

[N] Nakata, I., "On Compiling Algorithms for Arithmetic Expressions," *Comm. ACM* **10**:8 (1967), 492-494.

[R] Redziejowski, R. R., "On Arithmetic Expressions and Trees," *Comm. ACM* **12**:2 (1969), 81-84.

[S] Sethi, R., "Complete Register Allocation Problems," *Technical Report* No. 134, Computer Science Dept., Pennsylvania State University, May, 1974.

[SU] Sethi, R., and Ullman, J. D., "The Generation of Optimal Code for Arithmetic Expressions," *J. ACM* **17**:4 (1970), 715-728.

[W] Wasilew, S. G., "A compiler writing system with optimization capabilities for complex order structures," Ph. D. thesis, Northwestern University, Evanston, Illinois, 1971.

[WS] Walker, S. A., and Strong, H. R., "Characterizations of Flowchartable Recursions," *Proc. 4th Annual ACM Symposium on Theory of Computing,* (May 1972), 18-34.