SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

0097-5397/82/1104-0003 $01.00/0

EVALUATION OF ARITHMETIC EXPRESSIONS WITH ALGEBRAIC IDENTITIES*

TEOFILO GONZALEZ† AND JOSEPH JA'JA'‡

Abstract. We consider the problem of evaluating arithmetic expressions under a set of algebraic laws including the distributive law. An arithmetic expression can be represented by a dag and our problem is to find an equivalent dag with the fewest number of interior nodes. We attack the case when it is possible to eliminate common subexpressions and transform the dag into a tree; efficient algorithms to handle different cases of this problem are developed. These algorithms are based on the following strategy: we first transform the dag into a tree, assuming that such a transformation is possible, and we later check to see whether the tree and the given dag are indeed equivalent.

Key words. evaluation of arithmetic expressions, code generation, algorithms, dags

1. Introduction. Several interesting results concerning the problem of code generation for arithmetic expressions have been established by several authors. Extending the work of Anderson [A], Nakata [N], and Redziejowski [R], Sethi and Ullman [SU] have presented an efficient algorithm to generate minimal length codes for a special type of arithmetic expressions, namely those expressions with no common subexpressions. Aho and Johnson [AJ] have found a more general algorithm which allows general addressing features such as indirect addressing, but again restricting themselves to the same type of expressions. The case of arbitrary expressions has been proven to be difficult in a precise sense, i.e., it is NP-complete, even for the class of one-register machines with no algebraic identities allowed (Bruno and Sethi [BSe]). Aho et al. [AJU] have shown that the problem remains NP-complete for dags whose shared nodes are leaves or nodes at level one and have developed heuristic algorithms to generate good codes.

The effect of algebraic laws on code generation has received little attention in the literature. Sethi and Ullman [SU] have discussed the case where some of the operators of an expression tree are associative and commutative, and Breuer [B] used the distributive law to factor polynomials in a manner similar to that of Horner's algorithm. When certain algebraic transformations apply for an arithmetic expression A, we are not required to generate codes for A, but we may generate codes for any equivalent expression A' obtained by successive applications of the algebraic laws. Since the number of arithmetic operations may then vary, the optimality criterion of generated codes should depend on the number of arithmetic operations as well as on the code length. In this paper, we assume that the distributive law holds and consider the problem of minimizing the number of arithmetic operations for single arithmetic expressions which involve only addition and multiplication. We also assume that addition is commutative and associative and that multiplication is associative. This problem has been shown to be NP-hard [GJ1], [GJ2] even for expressions of degree 2 (degree of the arithmetic expression when viewed as a polynomial) and whose

* Received by the editors November 15, 1978, and in final revised form December 15, 1981. Preliminary versions of this paper appear as Technical Reports CS-80-4 and CS-78-13 at Pennsylvania State University.

† Programs in Mathematical Sciences, The University of Texas, Dallas, Texas 75080. The work of this author was supported in part by the National Science Foundation under grant MCS 77-21092.

‡ Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. The work of this author was supported in part by the National Science Foundation under grants MCS 78-06118 and MCS 78-27600.

corresponding graphs are leaf dags. However, in the case when common subexpressions can be eliminated, and the dag can thus be transformed into a tree, we develop efficient algorithms for the following types of dags:

a) the dag is a leaf dag (Theorem 5.15),

b) no term in the expression is repeated (Theorem 5.10),

c) the degree of the expression is bounded by some fixed constant (Theorem 5.16),

d) the level of sharing in the dag is bounded by some fixed constant (comment after Theorem 5.16).

2. Basic definitions. An arithmetic expression can be conveniently represented by a *directed acyclic graph*, referred to as a *dag*, in the same way basic blocks in code optimization are represented (Aho and Ullman [AU]). A dag has an interior node for each operation whose operands are the children of the node; the leaves of the dag represent initial values (variable names). For example, the arithmetic expression $A = (a * b + a * c) + (d * b + d * c)$ can be represented by the dag:

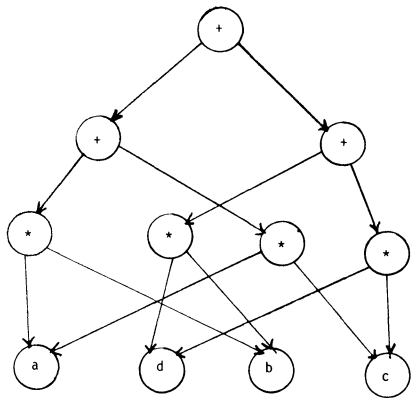


FIG. 2.1

The order of the children of an interior node is important; the leftmost child represents the first operand and the rightmost child represents the last operand.

We will restrict our attention to the following class of objects. Let Σ be a countable set of variable names and let $\theta = \{+, *\}$ be the set of binary operators on Σ such that the following laws hold:

(i) + and * are associative, i.e.,

$$(a + b) + c = a + (b + c),$$

$$(a * b) * c = a * (b * c), \quad \text{for all } a, b, c \in \Sigma;$$

(†) (ii) + is commutative, i.e.,

$$a + b = b + a, \quad \text{for all } a, b \in \Sigma;$$

(iii) * is distributive with respect to +, i.e.,

$$a * (b + c) = a * b + a * c,$$

$$(b + c) * a = b * a + c * a, \quad \text{for all } a, b, c \in \Sigma.$$

Strictly speaking, the above laws do not necessarily hold for actual expressions because of round-off errors in finite precision arithmetic. However, there is a general feeling [JMMW] that, for a given computation, the fewer the arithmetic operations, the less the worst-case round-off errors are, in spite of the fact that there are special situations where the opposite is true.

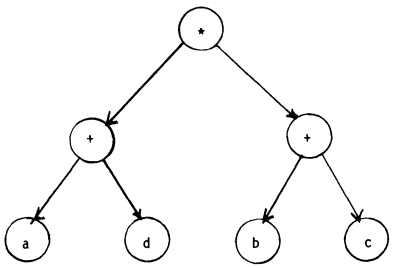
Another remark is concerned with the fact that we have not assumed that $*$ is commutative; the main reason is that the same techniques can be applied to a matrix expression to reduce the number of arithmetic operations. As an example, the expression $AB+AC+DB+DC$, where A, B, C and D are $n \times n$ matrices, is equivalent to $(A+D) \cdot (B+C)$ whose computation requires considerably fewer arithmetic operations than the original expression. This might also have some applications to code generation for parallel computers in which most of the operations are written in matrix form.

As a final remark, we note that identities such as $x+x=2x$, $x*x=x^2$ or $xy+x=x(y+1)$ do not exist.

We now define precisely the class of dags we are interested in. A *σ-dag* is a dag with a single root (i.e., a node without parents), whose interior nodes are either $+$ or $*$ from θ and whose leaves are from Σ. Note that no two leaves will represent the same element in Σ. If D is such a dag and v is a node from D, then the *expression corresponding to v*, denoted by $\exp(v)$, is defined as follows:

1) if v is a leaf, then $\exp(v)=v$,
2) else v corresponds to $\alpha \in \theta$; let v_1 and v_2 be the left and right children respectively, then $\exp(v)=\exp(v_1)\alpha \exp(v_2)$.

The expression corresponding to the dag D is just $\exp(r)$, where r is the root of D. Define two σ-dags D_1 and D_2 to be *equivalent* ($D_1 \equiv D_2$) if there exists a sequence of transformations from (†) which will transform D_1 into D_2. Given a σ-dag D, let $C[D]$ be the class of σ-dags equivalent to D; we are going to investigate the problem of finding a σ-dag $D' \in C[D]$ such that D' has the smallest possible number of interior nodes. If we define a *tree* to be a σ-dag such that none of its nodes has more than one parent, then we will attempt to design an algorithm which finds a tree $T \in C[D]$, whenever such a tree exists. In this case, we call the expression corresponding to T an *expression tree*, and we say that D is *tree-transformable*. If we consider, once again, the dag of Fig. 2.1, then it is easy to see that this dag is equivalent to the following tree:



whose evaluation requires 2 additions and one multiplication compared to 3 additions and 4 multiplications necessary to compute the dag of Fig. 2.1.

It is clear that if a tree T belongs to $C[D]$, for a σ-dag D, then T has the minimal number of interior nodes. Moreover, we can now use the algorithms already available in literature to generate corresponding minimal length codes.

Before closing this section, we make two more definitions and a comment. A *shared node* in a dag is a node with more than one parent; a *leaf dag* is a dag in which every shared node is a leaf (Aho et al. [AJU]). Let D be an arbitrary σ-dag with n total nodes, e edges, n_i interior nodes and v leaves (from Σ). It is easy to check that we always have the following relations:

$$n \geqq 2v-1, \qquad n_i \geqq v-1,$$
$$e = 2n_i = 2(n-v), \qquad \text{i.e.,} \quad e = O(n).$$

3. Motivation of the algorithm. We study, in this section, several properties of expression trees and examine some problems which are encountered in trying to develop an algorithm to transform a dag into a tree, whenever this is possible. We also develop some terminology which will be used in the subsequent sections.

Let A be an arithmetic expression with corresponding dag D; $L(A)$ or $L(D)$ will denote the set of leaves of D. $N(D)$ and $E(D)$ will represent the sets of nodes and edges in D respectively. Note that A can be written as $A = P_1 + \cdots + P_k$, where each P_i is a leaf or can be expressed as a product of arithmetic expressions. We call the P_is, the *product terms* of A or D. An arithmetic expression is in *normal form* (NF) if it is not possible to expand it using the distributive law. The expression $A_1 = (a*b+a*c)+(d*b+d*c)$ is in normal form and has $a*b$ as a product term, while $A_2 = a*(b+c)+d*(b+c)$ is not in normal form and has $a*(b+c)$ as a product term. The product terms of an arithmetic expression in normal form are called *normal terms*. $N_t(D)$ will denote the set of normal terms in the σ-dag D. Transforming a σ-dag into a normal form might correspond to an exponential growth in the number of nodes of the dag. For example, the expression $A = (x_1+x_2)*(x_3+x_4)*\cdots*(x_{2n-1}+x_{2n})$ (up to a fixed order) has a normal form whose dag has more than 2^n edges. We now define two more important terms.

DEFINITION 3.1. Given a σ-dag D, the *left factors* of D consist of the leaves which are the leftmost children of the normal terms of D.

DEFINITION 3.2. Given a σ-dag D with left factors $\{x_i\}_{i=1}^k$, a set of *right products* of D consists of a set of dags $\{P_i\}_{i=1}^k$ such that D is equivalent to the dag

$$D' = x_1 * P_1 + x_2 * P_2 + \cdots + x_k * P_k.$$

The expression $A_1 = (a*b+a*c)+(d*b+d*c)$ has the left factors $\{a, d\}$ with coresponding right products $\{(b+c), (b+c)\}$.

One could solve our problem by using the following divide-and-conquer approach:

(1) Find all the left factors and the corresponding right products of the given dag D.
(2) Recursively transform each right product into a tree.
(3) Combine the common right factors.

It may now seem that once we have efficient algorithms to implement steps (1) and (3), then we have an efficient algorithm to solve our problem. This is not the case since several of the right products could share several subexpressions, each of which will be processed several times. It is not hard to exhibit an example where the above strategy takes exponential time, given that each of steps (1) and (3) could be done in linear time.

It follows that we should avoid processing any subexpression more than once. What makes the problem harder is that two right products might be precisely the same and appear at different stages of the algorithm. On the other hand, it is not

posssible to transform a shared subexpression into a tree because there are subexpressions which are not tree-transformable and yet the expression to which they belong is tree-transformable. However, if a dag is tree-transformable, then we have the following characterization.

THEOREM 3.1. *Let* $\{x_i\}_{i=1}^{l}$ *be the set of left factors of a* σ*-dag* D *and let* $\{P_i\}_{i=1}^{l}$ *be the corresponding right products. If* D *is tree-transformable, then, for each* $i_1 \neq i_2$, *either*

(1) $L(P_{i_1}) \cap L(P_{i_2}) = \varnothing$ *or*

(2) *there exist* D_{i_1}, D_{i_2} *and* R *such that*

$$P_{i_1} \equiv D_{i_1} * R, \quad P_{i_2} \equiv D_{i_2} * R, \quad L(D_{i_1}) \cap L(D_{i_2}) = \varnothing.$$

Note that we can distinguish between (1) and (2) above quite easily by checking whether $L(P_{i_1}) \cap L(P_{i_2}) = \varnothing$ or not.

Let $\{P_{i_k}\}_{k=1}^{l}$ be a set of product terms which, we know, should overlap each other. The above characterization suggests that we solve the problem for just one P_{i_k}, say P_{i_1}, and use its right subtree to eliminate the overlap with all the other P_is. For example in the case where we only have two right products P_{i_1} and P_{i_2}, we transform (recursively) P_{i_1} into a tree and write it as a product, say

$$P_{i_1} \equiv (\cdots (Q_m * Q_{m-1}) * \cdots * Q_1), \qquad m \geqq 1.$$

Now if $L(Q_1) \cap L(P_{i_2}) \neq \varnothing$ (which should be true in this case), we somehow factor Q_1, and write P_{i_2} as $P_{i_2} \equiv R_{i_2} * Q_1$. We continue in this way until we have no more overlap; then we apply the procedure recursively to the nonoverlapping parts. We would like to emphasize one more point about this procedure: We assume that Q_1 will be a factor, and find R_{i_2}. This assumption is justfied if the dag is tree-transformable. Otherwise the procedure will construct an inequivalent tree; we will use the equivalent algorithm in § 5 to check whether a given σ-dag and a given tree are equivalent.

Let us summarize the general strategy of the algorithm. It consists of two parts:

(i) *The transformation algorithm* which proceeds and transforms the given σ-dag into a tree assuming the dag is tree-transformable;

(ii) *The equivalence algorithm* which, for a given σ-dag and a given tree, checks if they are indeed equivalent.

4. The transformation algorithm. The different parts of the transformation algorithm will be described in this section, together with the proofs of its correctness and its complexity. We start by discussing the algorithms to find the left factors and the corresponding right products of a general σ-dag D. $|N(D)|$ will denote the number of nodes in D.

The procedure to find the left factors is fairly straightforward. It is just a bottom-up labeling of the dag based on the following observation. Let LF (y) denote the set of left factors of the subdag rooted at y. Then

$$\mathrm{LF}(y) = \begin{cases} \{y\} & \text{if } y \text{ is a leaf,} \\ \mathrm{LF}(s_1) \cup \mathrm{LF}(s_2) & \text{if } y \text{ is a } + \text{ node with children } s_1 \text{ and } s_2, \\ \mathrm{LF}(s_1) & \text{if } y \text{ is a } * \text{ node and } s_1 \text{ is its left child.} \end{cases}$$

In order to have a linear time algorithm, we must avoid visiting nodes more than once. In order to guarantee this, we mark the nodes visited. We use a function tc $(\cdot)$ for this purpose; this function will serve another purpose in the next procedure when initialized properly by the procedure to find the left factors.

Let r be the root of a dag D. Let $\text{tc}(z) = 0$, for every $z \in N(D)$, and let $L = \varnothing$. When LEFT-FACTORS (r) terminates, L will denote the set of left factors of D and $\text{tc}(z)$ will be equal to the size of {w| there is a call to LEFT-FACTORS (w) and (w is a + node with z as one of its children or w is a $*$ node with z as its left child)}. LC (r) and RC (r) denote respectively the left and right children of r. This notation will be used consistently throughout the paper.

```
     procedure LEFT-FACTORS (r)
     begin
     global (tc[·], L)
1.       If tc (r) ≠ 0 then [tc (r) ← tc (r) + 1;
                             return];
3.       case
4.       :r is a leaf: [L ← L ∪ {r}];
5.       :r is a * node: [call LEFT-FACTORS (LC (r))];
6.       :r is a + node: [call LEFT-FACTORS (LC (r));
7.                        call LEFT-FACTORS (RC (r))];
8.       endcase
9.       tc (r) ← 1;
10.      return
11.  end LEFT-FACTOR
```

Let r be the root of a σ-dag D. The set of left factors of the σ-dag with root $y \in N(D)$ is denoted by L_y. Consider now any call to LEFT-FACTORS (y); let $L' = L$ just before the call and let $L'' = L$ after the procedure LEFT-FACTORS (y) terminates, where L is the set of left factors computed by the algorithm.

LEMMA 4.1. *Let r, y, L', L'' and L_y be as defined above.*

a) *If* $\text{tc}(y) \geqq 1$ *before the call to* LEFT-FACTORS (y), *then* $L_y \subseteq L'$.

b) *If* $\text{tc}(y) = 0$ *before the call to* LEFT-FACTORS (y), *then when the procedure terminates* $\text{tc}(y) = 1$ *and*

$$L'' = L' \cup (L_y - L').$$

Proof. The proof for part a) follows from part b) and the one for part b) is by induction on the height of the σ-dag with root y. □

LEMMA 4.2. LEFT-FACTORS (r) *correctly finds the left factors of D, i.e., $L = L_r$.*

Proof. The proof follows from Lemma 4.1 together with the initial condition $L = \varnothing$ and $\text{tc}(y) = 0$, for every $y \in N(D)$. □

The algorithm to find a set of right products of a given σ-dag D is discussed now. It is easy to design a bottom-up algorithm to do the job, but for efficiency reasons, our algorithm will process the dag top-down. Before presenting the algorithm, we introduce some notation. $\text{pt}(y)$ will denote a pointer from a node y of D to a dag written as a right-hand side of an assignment statement; e.g., $\text{pt}(y) \leftarrow \varnothing$ should be interpreted as a pointer from y to the empty dag. Another convention is that the assignment

$$\text{pt}(N_0) \leftarrow \text{pt}(N_1)\ \theta\ \text{pt}(N_2),$$

where $\theta = \{+, *\}$, is understood to mean the assignment of Fig. 4.1. If one of $\text{pt}(N_1)$ or $\text{pt}(N_2)$ happened to be $\varnothing$, then the above statement is interpreted as $\text{pt}(N_0) \leftarrow \text{pt}(N_2)$ or $\text{pt}(N_0) \leftarrow \text{pt}(N_1)$, respectively.

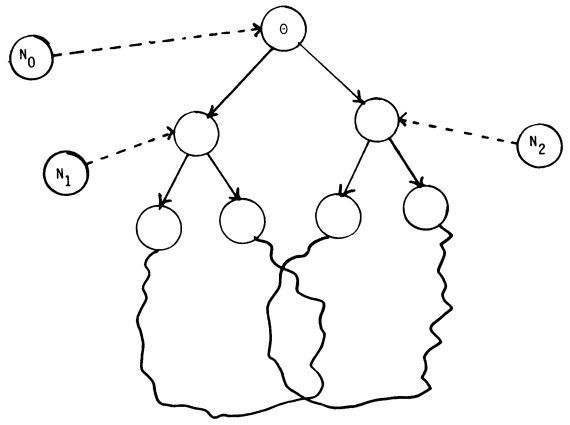


FIG. 4.1

We are now ready for the procedure. The main idea of the algorithm is a top-down traversal of the dag which makes nodes point to subdags in such a way that the left factor nodes will point to the corresponding right products. If we are visiting node y with s_1 and s_2 as its left and right children, then the following changes are made:

a) y is a + node,

$$\text{pt}(s_1) \leftarrow \text{pt}(y) \boxplus \text{pt}(s_1)$$

$$\text{pt}(s_2) \leftarrow \text{pt}(y) \boxplus \text{pt}(s_2).$$

b) y is a $*$ node,

$$\text{pt}(s_1) \leftarrow \text{pt}(s_1) \boxplus (s_2 \boxtimes \text{pt}(y)).$$

We cannot proceed and visit any of the children unless all of its parents have been visited. That is why we use the function tc( ) in our procedure below. In order to make inductive assertions about the algorithm, we use a function $w: N(D) \leftarrow \{0, 1\}$, which is initialized by $w(x) = 0$, for all $x \in N(D)$ unless x is the root r in which case $w(r) = 1$. This function serves no other purpose.

```
     procedure RIGHT-PRDS (r)
     begin
     global (tc[·], pt[·], w[·]);
1.       tc(r) ← tc(r) − 1
2.       If tc(r) ≠ 0 then [return];
3.       case
4.          :r is a leaf: [return];
5.          :r is a ⊕ node:
6.                   [w(r) ← 0; w(LC(r)) ← w(RC(r)) ← 1;
7.                   pt(LC(r)) ← pt(LC(r)) ⊞ pt(r);
8.                   pt(RC(r)) ← pt(RC(r)) ⊞ pt(r);
9.                   call RIGHT-PRDS (LC(r));
10.                  call RIGHT-PRDS (RC(r));
11.                  return];
12.         :r is a ⊛ node:
13.                  [w(r) ← 0; w(LC(r)) ← 1;
14.                  pt(LC(r)) ← pt(LC(r)) ⊞ (RC(r) ⊠ pt(r));
```

15. **call** RIGHT-PRDS (LC (r));
16. **return**];
17. **endcase**
18. **end of procedure** RIGHT-PRDS (r)

To clearly illustrate the usefulness of tc [·], we introduce an example.

Example 4.1. Consider the dag of Fig. 4.2. At the end of LEFT-FACTORS (N_0), we have

$$\text{tc}(N_0) = \text{tc}(N_1) = \text{tc}(N_2) = \text{tc}(N_4) = \text{tc}(a) = 1,$$
$$\text{tc}(N_3) = 2 \quad \text{and} \quad L = \{a\}.$$

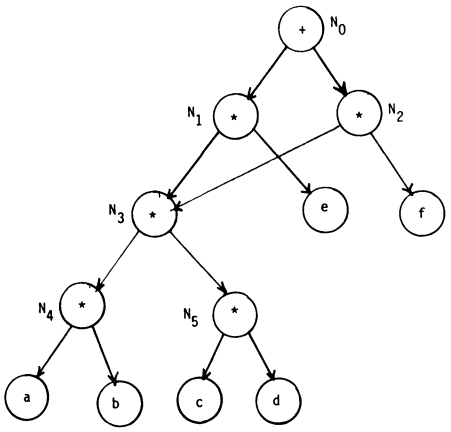


FIG. 4.2

Let us now apply RP (N_0).

Recursive call	Result
RP (N_0)	RP (N_1), RP (N_2)
RP (N_1)	pt (N_3) ← e, RP (N_3)
RP (N_3)	tc (N_3) ← 1
RP (N_2)	pt (N_3) ← $e+f$, RP (N_3)
RP (N_3)	pt (N_4) ← $N_5 * (e+f)$, RP (N_4)
RP (N_4)	pt (a) ← $b * (N_5 * (e+f))$, RP (a)
RP (a)	–

Note that the original expression $(a * b) * (c * d) * e + (a * b) * (c * d) * f$ is indeed equivalent to $a * \text{pt}(a) = a * (b * ((c * d) * (e+f)))$. We now proceed to prove the correctness of the above algorithm.

Let D be a tree-transformable σ-dag with root r. Just before a call to RIGHT-PRDS (y), let $R = \{z \mid z \in N(D)$ and $w(z) = 1\}$ and $D = \sum_{z \in R} z * \text{pt}(z)$. After the procedure RIGHT-PRDS (y) terminates, let $R' = \{z \mid z \in N(D)$ and $w(z) = 1\}$ and $D' = \sum_{z \in R'} z * \text{pt}(z)$.

LEMMA 4.3. *Let R, R', D and D' be as defined above. Assume $y \in R$. If there is a call to* RIGHT-PRDS (y), *then after the procedure terminates, $D \equiv D'$.*

Proof. There are two cases depending on the value of tc (y) at the time of the call.

Case 1. tc (y) $\neq 1$. It is simple to verify that in the procedure none of the values of $w(z)$ or pt (z), for $z \in N(D)$, is modified. Hence $R' = R$ and $D' \equiv D$.

Case 2. $\mathrm{tc}(y) = 1$. The proof is by induction on the height $h \geqq 0$ of y. If $h = 0$, then y must be a leaf. It is simple to verify that in the procedure none of the values $w(z)$ or $\mathrm{pt}(z)$, for $z \in N(D)$, is modified. Hence $R' = R$ and $D' \equiv D$. Suppose now that the height of y is $h+1 > 1$. As $h+1 > 1$, it must be that y is a $\circledast$ or a $\oplus$ node. By assumption $\mathrm{tc}(y) = 1$, so step 3 is executed and as $y \in R$ then

$$D = \sum_{z \in R - \{y\}} z * \mathrm{pt}(z) + y * \mathrm{pt}(y).$$

There are two subcases.

Subcase a. *r is a* $\circledast$ *node.*

a.1. $w(\mathrm{LC}(y)) = 1$ *before line* 3. At this point D can be written as

$$D = \sum_{z \in R - \{y, \mathrm{LC}(y)\}} z * \mathrm{pt}(z) + y * \mathrm{pt}(y) + \mathrm{LC}(y) * \mathrm{pt}(\mathrm{LC}(y)).$$

By definition, $y = \mathrm{LC}(y) * \mathrm{RC}(y)$. Applying the distributive law, we obtain

$$D = \sum_{z \in R - \{y, \mathrm{LC}(y)\}} z * \mathrm{pt}(z) + \mathrm{LC}(y) * (\mathrm{pt}(\mathrm{LC}(y)) + \mathrm{RC}(y) * \mathrm{pt}(y)).$$

After line 14, $D = \sum_{z \in R - \{y\}} z * \mathrm{pt}(z)$. In line 13, $w(y)$ is set to zero. Let $R'' = \{z \mid z \in N(D) \text{ and } w(z) = 1\}$ at this point. Clearly $R'' = R - \{y\}$ so $D'' \equiv D$, where D'' is defined using R''. Since the height of the σ-dag with root $\mathrm{LC}(r)$ is $\leqq h$ and $\mathrm{LC}(r) \in R''$, it then follows by induction that after line 15,

$$D \equiv D'' \equiv D' = \sum_{z \in R'} z * \mathrm{pt}(z).$$

The procedure terminates (line 16) with

$$D \equiv D' = \sum_{z \in R'} z * \mathrm{pt}(z).$$

a.2. $w(\mathrm{LC}(y)) = 0$ *before line* 3. The proof is similar to that for a.1.

Subcase b. *r is a* $\oplus$ *node.* The proof is similar to that for subcase a and will be omitted. □

We collect all the above facts about the procedures LEFT-FACTORS and RIGHT-PRDS in the following theorems.

THEOREM 4.4. *Let D be a tree-transformable* σ*-dag with root r. Then* LEFT-FACTORS (r) *and* RIGHT-PRDS (r) *will generate an equivalent* σ*-dag* D'' *of the form*

$$D \equiv D'' = \sum_{x \in L_r} x * \mathrm{pt}(x),$$

where L_r *is the set of left factors of D.*

Proof. Using the initial conditions $w(y) = 0$ for every $y \in N(D)$, $w(r) = 1$ and $D = r$ together with Lemma 4.3, it then follows that after procedure RIGHT-PRDS (r) terminates, $D' = \sum_{z \in R'} z * \mathrm{pt}(z)$. Furthermore, $D' \equiv D = r$. So, $D' \equiv D''$. To complete the proof it is required to show that $R' = L_r$. Both LEFT-FACTORS (r) and RIGHT-PRDS (r) will make the same recursive calls. So, it must be that for each $y \in N(D)$, if there were l calls to RIGHT-PRDS (y), then there must have been l calls to LEFT-FACTORS (y). As $\mathrm{tc}(y)$ is the total number of calls to LEFT-FACTORS (y) after LEFT-FACTORS (r) terminates, then $\mathrm{tc}(y) = 0$ for all $y \in N(D)$ after RIGHT-PRDS (r) terminates. So, after RIGHT-PRDS (r) terminates, all internal nodes y of D will have $w(y) = 0$ (see lines 6 and 13) and all leaves visited will have $w(\cdot) = 1$. Hence, $y \in R'$ if and only if y is a leaf visited from RIGHT-PRDS (r). From Lemma

4.2 we have that $L = L_r$ when LEFT-FACTORS (r) terminates. By inspection of procedure LEFT-FACTORS, $y \in L$ if and only if y is a leaf visited from LEFT-FACTORS (r). As the same leaves are visited by both procedures, then $R' = L_r$.

This completes the proof of the theorem. □

THEOREM 4.5. *The time complexity of* LEFT-FACTORS (r) *and* RIGHT-PRDS (r) *is* $O(|N(D)|)$.

Proof. The proof follows from the observations that no edge in D is traversed more than once. □

We will now establish a relationship between the number of nodes in the dag constructed by LEFT-FACTORS and RIGHT-PRDS and the number of nodes in the original graph (Theorem 4.7). This result is used in the proof of Theorem 4.9. Beforehand, we need the following lemma.

LEMMA 4.6. *Let* $p(y)$ *be the number of parents of a node* y, $y \in N(D)$. *If, after the execution of* LEFT-FACTORS (r), *there exists* $z \in N(D)$ *such that* $0 < \text{tc}(z) < p(z)$, *then* D *is not tree-transformable.*

Proof. Note that since $\text{tc}(z) < p(z)$, z must be a right descendant of a ⊛ node. On the other hand, $\text{tc}(z) > 0$ implies that the left factors of z are also left factors of the dag D. These two observations imply that D is not tree transformable. □

Our procedure does not compute $p(z)$; however, it is trivial to design a procedure which computes $p(z)$. Another procedure could also be constructed to verify that, for each node $z \in N(D)$, either $\text{tc}(z) = p(z)$ or $\text{tc}(z) = 0$. Both procedures would run in time $O(|N(D)|)$. In what follows, we assume that these two procedures are actually executed in between LEFT-FACTORS (r) and RIGHT-PRDS (r) and therefore the situation of Lemma 4.6 cannot arise.

Let D be a σ-dag with root r and let n_1 be the number of nodes in the g product terms of D. Clearly, $|N(D)| = n_1 + g - 1$. LEFT-FACTORS (r) and RIGHT-PRDS (r) generate an equivalent σ-dag of the form $D' = \sum_{i=1}^{k} x_i * \text{pt}(x_i)$, where the summation is in some fixed order.

THEOREM 4.7. *Let* D, n_1, g, D' *and* k *be as defined above. Then*

$$|N(D')| \leqq |N(D)| + k - g.$$

Proof. Let m' and p' be the numbers of ⊛ and ⊕ nodes respectively whose $\text{tc}(\cdot) \neq 0$ after procedure LEFT-FACTORS (r) terminates. Clearly k is the number of leaves with $\text{tc}(\cdot) \neq 0$ (i.e., the left factors). We try now to account for all the new nodes we create in D'. Each ⊕ node with $\text{tc}(\cdot) \neq 0$ may generate two plus nodes in lines 7 and 8 of the procedure RIGHT-PRDS (r) except in the case where the pointer of the node is ∅. It is easy to see that the ⊛ nodes could generate at most $2(p' - (g-1))$ new nodes. On the other hand, each ⊛ node with $\text{tc}(\cdot) \neq 0$ creates at most one ⊕ node and one ⊛ node in line 14 of RIGHT-PRDS (r). However, as before, whenever the pointer of the ⊛ node is ∅, no new ⊛ node will be generated; therefore, the m' ⊛ nodes can generate at most $2m' - g$ new nodes. Since $\text{pt}(y)$ is initialized to ∅, for each $y \in N(D)$, the first time lines 7, 8 and 14 are executed, the corresponding ⊕ nodes are not introduced. Therefore, the total number of new nodes in D' is

$$\begin{aligned} &2(p' - (g-1)) + 2m' - g - ((p' - (g-1)) + (m' - g) + k) \\ &\quad = p' + m' - g - k + 1. \end{aligned}$$

Constructing the expression $x_1 * \text{pt}(x_1) + \cdots + x_k * \text{pt}(x_k)$ requires the introduction of at most $2k - 1$ new nodes. It follows that the total number of new nodes is at most $(p' + m' - g - k + 1) + (2k - 1) = p' + m' + k - g$.

On the other hand, all interior nodes with $\mathrm{tc}(\cdot) \neq 0$, after procedure LEFT-FACTORS (r) terminates, will not appear in D'. Therefore

$$|N(D')| \leqq |N(D)| - (m' + p') + (p' + m' + k - g), \quad \text{i.e.,}$$

$$|(D')| \leqq |N(D) + k - g.$$ □

COROLLARY. *Let D, n_1, D' and k be as defined above. Then $|N(D')| \leqq n_1 + k - 1$.*

Before giving the precise overall transformation algorithm, we outline its general strategy. Let D be a given σ-dag.

(1) Identify the left factors and the set of right products $\{P_i\}_{i=1}^{l}$ of D using the procedures described above.
(2) Split the right products into nonoverlapping sets of dags $\{S_i\}_{i=1}^{k}$.
(3) Suppose the set S_i consists of $\{i_1, i_2, \cdots, i_t\}$. Recursively, transform P_{i_1} into a tree, say T_i; T_i can be written as $T_i = ((\cdots (Q_{i,m} * Q_{i,m-1}) * \cdots) * Q_{i,1})$, $m \geqq 1$, where each $Q_{i,j}$ is a leaf or a tree with a $\oplus$ root. Figure out the overlap between T_i and $x_{i_l} * P_{i_l}$, $l > 1$, and the missing factor in $x_{i_l} * P_{i_l}$. Transform the missing factor in $x_{i_l} * P_{i_l}$ into a tree.
(4) Combine the subtrees.

Step (2) is very easy to do: just find the connected components of the graph induced by the set of right products $\{P_i\}_{i=1}^{l}$. Step (4) is also quite straightforward. Step (3) is a bit harder and can be done as follows. For each $1 \leqq s \leqq m$, obtain a normal term f_s of $Q_{i,s}$. For each z, let r_z be the maximum integer such that $Q_{i,1}, Q_{i,2}, \cdots, Q_{i,r_z}$ appear in $x_{i_z} * P_{i_z}$, i.e.,

$$x_{i_z} * P_{i_z} = R_{i_z} * ((\cdots ((Q_{i,r_z} * Q_{i,r_z-1}) * \cdots) * Q_{i,2}) * Q_{i,1}),$$

where $R_{i_z} \cap Q_{i,j} = \varnothing$, for all $j > r_z$ (such an $r_z \geqq 1$ exists by virtue of Theorem 3.1). Let $N_t(Q_{i,j})$ be the set of normal terms of $Q_{i,j}$. Then

$$Q_{i,j} = \sum_{q \in N_t(Q_{i,j})} q = f_j + \sum_{\substack{q \in N_t(Q_{i,j}) \\ q \neq f_j}} q.$$

Therefore,

$$x_{i_z} * P_{i_z} = R_{i_z} * f_{r_z} * \cdots * f_2 * f_1 + R_{i_z} * \Gamma,$$

where

$$N_t(\Gamma) = \sum_{\substack{q \in N_t(Q_{i,r_z} * \cdots * Q_{i,1}), \\ q \neq f_{r_z} * \cdots * f_2 * f_1}} q.$$

Assume without loss of generality that y_0 and $y_1 \notin \Sigma$, i.e., no leaf in the σ-dag contains the symbols y_0 or y_1. Now, let us substitute in the σ-dag for $x_{i_z} * P_{i_z}$, the symbol y_1 for all the leaves in $L(f_{r_z} * \cdots * f_2 * f_1)$ and the symbol y_0 for all the leaves in $L(Q_{i,r_z} * \cdots * Q_{i,2} * Q_{i,1}) - L(f_{r_z} * \cdots * f_1)$.

The reader should associate the symbol y_0 with the value zero and y_1 with the value one. The σ-dag obtained by *partially evaluating* a σ-dag with root y, $\mathrm{pe}(y)$; is defined in terms of the normal terms it contains:

$$N_t(\mathrm{pe}(y)) = \begin{cases} y_1 & \text{if } \exists c \in N_t(y) \text{ such that } L(c) \subseteq L(y_1) \text{ and } \forall c \in N_t(y) \\ & \text{either } L(c) \subseteq L(y_1) \text{ or } y_0 \in L(c), \\ y_0 & \forall c \in N_t(y),\ y_0 \in L(c), \\ w & \exists c \in N_t(y) \text{ such that } y_0 \notin L(c) \text{ and } L(c) \cap \Sigma \neq \varnothing, \end{cases}$$

where $w = \{h | c \in N_t(y), y_0 \notin L(c), L(c) \cap \Sigma \neq \emptyset$ and h is c after deleting all the y_1s$\}$. From the definition of pe (y) it should be clear that pe $(x_{1_z} \circledast P_{i_z})$ is R_{i_z}.

The computation of pe (y) for all $y \in N(x_{i_z} \circledast P_{i_z})$ is carried out bottom-up in the obvious way after initializing the leaves in $Q_{i,r_z}, \cdots, Q_{i,1}$, as mentioned above. We actually compute all the pe (y) for all the $x_{i_z} \circledast P_{i_z}$ simultaneously by first constructing the dag for $rr = x_{i_2} \circledast P_{i_2} \boxplus x_{i_3} \circledast P_{i_3} \boxplus \cdots \boxplus x_{i_z} \circledast P_{i_z}$; then initializing the leaves in $Q_{i,m}, \cdots, Q_{i,1}$ and $z_q, \cdots, z_1$, where $z_q \circledast \cdots \circledast z_1$ is one normal term in $Q_{i,m} * \cdots * Q_{i,1}$; then computing pe (y) for all $y \in N(rr)$ as outlined above and finally extracting pe $(y) = R_{i_z}$, where y is the root of $x_{i_z} \circledast P_{i_z}$, for all z.

The formal algorithm is given below. If r is the root of D, TRANSFORM (r) will generate the tree. $D(y)$ will denote the subdag rooted at y.

procedure TRANSFORM (r)
begin
1. Let $x_1 * P_1, x_2 * P_2, \cdots, x_h * P_h$ be the set of left factors and right products of $D(r)$;
2. Let $A = \{i | P_i = \emptyset\}$ and $B = \{i | P_i \neq \emptyset\}$;
   //t will be the root of the tree constructed//
3. $t \leftarrow \emptyset$;
   //the sum of the left factors with empty right products is constructed//
4. **for** each $i \in A$ **do** $t \leftarrow t \boxplus x_i$ **endfor**;
   //Partition the set B in such a way that α and β belong to the same set iff $L(P_\alpha) \cap L(P_\beta) \neq \emptyset$.//
5. Let R be the equivalence relation defined over the elements of set B in such a way that $\alpha R \beta$ iff $L(P_\alpha) \cap L(P_\beta) \neq \emptyset$. Partition B into the sets of equivalence classes $S_1, S_2, \cdots, S_k$ under R.
   //Clearly $\sum_{i \in A} x_i + \sum_{i \in B} x_i * P_i \equiv t + \sum_{j=1}^{k} \sum_{i \in S_j} x_i * P_i$//
   //for each S_i, find an equivalent tree (t')//
6. **for** $i = 1$ **to** k **do**
7. Let $S_i = \{j_1, j_2, \cdots, j_m\}$;
8. $n_0 \leftarrow$ TRANSFORM (P_{j_1}); //construct a tree equivalent to P_{j_1}//
   case
9. $:m = 1$: $[t' \leftarrow x_{j_1} \circledast n_0$; //$|S_i| = 1$//];
10. $:m > 1$: $[rr \leftarrow x_{j_2} \circledast P_{j_2} \boxplus x_{j_3} \circledast P_{j_3} \boxplus \cdots \boxplus x_{j_m} \circledast P_{j_m}$;
11. Let $Q_w, Q_{w-1}, \cdots, Q_1$ be the factors in n_0, i.e., $n_0 = ((\cdots (Q_w \circledast Q_{w-1}) \circledast \cdots) \circledast Q_2) \circledast Q_1$, where each Q_p is a leaf or a subtree with a $\oplus$ root.
12. Compute label (y) for each $y \in N(rr)$;
    //if the root of $x_{i_a} * P_{i_a}$ is labeled s then $x_{i_a} * P_{i_a}$ can be written as $R_{i_a} * Q_s * \cdots * Q_1$ and R_{i_a} does not overlap with $Q_w, \cdots, Q_{s+1}$//
    //take a normal term from $Q_w \circledast Q_{w-1} \circledast \cdots \circledast Q_1$//
13. $(z_q \circledast z_{q-1} \circledast \cdots \circledast z_1) \leftarrow$ NORMAL-TERM $(Q_w \circledast Q_{w-1} \circledast \cdots \circledast Q_1)$;
    //Transform each $x_{i_a} * P_{i_a}$ into $R_{i_a} * Q_s * \cdots * Q_1$ where s is the label of the root of $x_{i_a} * P_{i_a}$. This operation is performed by partially evaluating $x_{i_a} \circledast P_{i_a}$ (actually we partially evaluate all the $x_{i_a} \circledast P_{i_a}$s at the same time by partially evaluating rr)//
14. **for** each $y \in N(rr)$ **do**
    compute pe (y) after initializing the leaves in $L(z_q \circledast \cdots \circledast z_1)$ to y_1 and the leaves in $L(Q_w \circledast \cdots \circledast Q_1) - L(z_q \circledast \cdots \circledast z_1)$ to y_0.

```
            endfor
15.         RS_l ← ∅ for l = 1, 2, ···, w;
16.         for l = 2 to m do
17.           Let y be the root of x_{i_l} ⊡ P_{i_l};
18.           RS_{label(y)} ← RS_{label(y)} ⊞ pe(y);
19.         endfor
20.       t' ← ((··· ((TRANSFORM (RS_w) ⊞ x_{j_1}) ⊡ Q_w ⊞
                TRANSFORM (RS_{w-1})) ⊡ Q_{w-1} ⊞ ··· ⊞ TRANSFORM (RS_2)) ⊡ Q_2 ⊞
                TRANSFORM (RS_1)) ⊡ Q_1];
        endcase
21.     t ← t ⊞ t';
22. endfor
23. return (t);
    end of procedure TRANSFORM
```

We now describe the procedure NORMAL-TERM $(\cdot)$ which was used in the body of the above procedure.

```
procedure NORMAL-TERM (n_0)
begin
    case
      :n_0 is a leaf: [return (n_0)];
      :n_0 is a ⊞: [return (NORMAL-TERM (RC (n_0))];
      :else: [return (NORMAL-TERM (LC (n_0))
            ⊡ NORMAL-TERM (RC (n_0)))];
    endcase
end of procedure NORMAL-TERM
```

Note that procedure NORMAL-TERM (n_0) does not mark the nodes of n_0. However, when n_0 is the root of a tree, the procedure takes linear time with respect to the number of nodes in the tree. If the dag we wish to transform is tree-transformable, then all calls made from procedure TRANSFORM to procedure NORMAL-TERM will involve trees.

We now consider an example. Let $A = ((a + b * c) * f + b * (c * e + d * f)) + (a + b * d) * e$. In this case, it is easy to see that the left factors are given by $x_1 = a$ and $x_2 = b$ with corresponding right products $\text{pt}(a) = (f + e) = P_1$ and $\text{pt}(b) = [c * (f + e) + d * (f + e)] = P_2$. Thus P_1 and P_2 are in the same connected component. At step 8, we will have $n_0 \leftarrow (f + e)$ which has one factor $Q_1 = (f + e)$. At step 12, the root of $b * P_2$ will be labeled 1 at step 13 NORMAL-TERM (Q_1) will return e. The partial evaluation of rr will produce the expression $b * (c + d)$. In lines 15–19 RS_1 is set to be $b * (c + d)$. Finally, at step 20, we get (TRANSFORM $(b * (c + d)) + a) * (f + e)$, i.e., $(b * (c + d) + a) * (f + e)$, which is indeed an equivalent tree.

We are now ready to prove the main theorem of this section.

THEOREM 4.8. *Let D be a tree-transformable σ-dag with root r. Then* TRANSFORM (r) *generates an equivalent tree T with root t.*

Before giving the proof, we make the following definition.

DEFINITION 4.3. The *degree* d of a dag D (or of the corresponding expression) is the maximum number of variables in any normal term of D.

Proof of Theorem 4.8. The proof is by induction on the degree d of D.

Assume $d = 1$. In line 1 the set of left factors and corresponding right products of D is obtained using LEFT-FACTORS (r) and RIGHT-PRDS (r). From Theorem

4.4, it follows that $D \equiv \sum_{i=1}^{h} x_i * P_i$. Since $d = 1$, D consists of simple variables, i.e., $P_i = \varnothing$, for $1 \leqq i \leqq h$. Therefore $A = \{1, 2, \cdots, h\}$ and $B = \varnothing$ in line 2; executing line 4 produces a tree $t \equiv D$. All the other parts of the algorithm will be skipped because $B = \varnothing$ and $k = 0$.

Suppose now $d > 1$. As before, the execution of line 1 produces an equivalent dag $D \equiv \sum_{i=1}^{h} x_i * P_i$. In line 2, the set $\{1, 2, \cdots, h\}$ is partitioned into two sets A and B such that $D \equiv \sum_{i \in A} x_i + \sum_{i \in B} x_i * P_i$. Line 4 sets t to $t = \sum_{i \in A} x_i$. Line 5 partitions B into disjoint sets in such a way that if i and j belong to the same set, then $L(P_i) \cap L(P_j) \neq \varnothing$. It follows that $D \equiv t + \sum_{i=1}^{k} U_i$, where $U_i = \sum_{j \in S_i} x_j * P_j$, $1 \leqq i \leqq k$. This partition is justified by Theorem 3.1.

Loop 6–22 transforms each U_i into a tree t' which is added to t in line 21. To complete the proof, it is only required to show that, after the execution of lines 7–20, $t' \equiv U_i$.

CLAIM. *For each* $1 \leqq i \leqq k$, *the tree* t' *generated by lines* 7–20 *is equivalent to* U_i.

Proof of the claim. Since S_i consists of the elements $\{j_1, j_2, \cdots, j_m\}$, U_i is the dag $x_{j_1} * P_{j_1} + \cdots + x_{j_m} * P_{j_m}$ (for some fixed order). It is clear that since the degree of P_{j_1} is $\leqq d - 1$, it follows by the induction hypothesis that n_0 is a tree equivalent to P_{j_1}. Thus $U_i \equiv x_{j_1} * n_0 + \sum_{z=2}^{m} x_{j_z} * P_{j_z}$, after line 8. Note that if $m = 1$, then we are done. Therefore, let's assume that $m > 1$. After lines 10 and 11, we obtain $U_i \equiv x_{j_1} * ((\cdots (Q_w * Q_{w-1}) * \cdots) * Q_2) * Q_1 + rr$, where each Q_i is a leaf or a tree with a $\oplus$ root.

Label(y) is computed for each $y \in N(rr)$ (line 12) in such a way that if the root of $x_{i_a} * P_{i_a}$ is labeled b then $x_{i_a} * P_{i_a}$ can be written as $R_{i_a} * Q_b * \cdots * Q_1$ and $L(R_{i_a}) \cap L(Q_w * \cdots * Q_{b+1}) = \varnothing$. In line 13 we extract a normal term from $Q_w * Q_{w-1} * \cdots * Q_1$. pe ($y$) is computed in line 14 in such a way that if y is the root of $x_{i_a} * P_{i_a}$ then pe (y) $= R_{i_a}$ as defined above. After the execution of lines 15–19 we have:

$$U_i \equiv x_{j_1} * P_{j_1} + \sum_{a=2}^{w} x_{j_a} * P_{j_a}$$

$$\equiv ((\cdots ((RS_w \boxplus x_{j_1}) \circledast Q_w \boxplus RS_{w-1}) \circledast Q_{w-1} \boxplus \cdots \boxplus RS_2) \circledast Q_2 \boxplus RS_1) \circledast Q_1.$$

Procedure TRANSFORM is used in step 20 to obtain equivalent trees for $RS_w, \cdots, RS_1$. Since the degree of each RS_l is $< d$, it then follows by induction that t', as constructed by line 20, is equivalent to U_i. This completes the proof of the claim and the theorem. □

Let D be a tree-transformable σ-dag with root r and of degree d. Let n be the number of nodes in the g product terms and let v be the number of leaves in D. Let $T(n, d)$ be the time complexity of procedure TRANSFORM (r).

THEOREM 4.9. *Let* d, n, v, r, g *and* D *be as defined above. Then* $T(n, d) \leqq C_1 dn$, *where* C_1 *is some fixed constant.*

Proof. First of all, let us determine the time complexity of steps 1–5. Line 1 takes $\leqq C_2(n + g)$ time since D has exactly $n + g - 1$ nodes and procedures LEFT-FACTORS and RIGHT-PRDS take time $O(|N(D)|)$ (see Theorem 4.5). Lines 2–4 take time $\leqq C_2 h$. Line 5 can be implemented by finding the connected components of a graph for P_j, $j \in B$, (of course, we ignore the direction of the edges) which can be easily carried out in $\leqq C_2 m$ steps, where m is the number of nodes in the graph for P_j, $j \in B$. Since the number of nodes in $\sum_{i=1}^{h} x_i * P_i$ is $\leqq n + h - 1$ (see Theorem 4.7), then line 5 takes time $\leqq C_2(n + h)$. Clearly $g \leqq n$ and $h \leqq n$. Hence, lines 1–5 take time $\leqq 5C_2 n$.

In what follows, we prove by induction on the degree $d \geqq 1$ of the σ-dag with root r, that $T(n, d) \leqq C_1 dn$, where $C_1 = 12C_2$.

For $d = 1$, we know that B must be empty. Therefore, loop 6–22 is not executed. By assumption $C_1 > 5C_2$. Hence, $T(n, 1) \leqq C_1 n \leqq C_1 dn$.

Suppose now the degree of D is $d > 1$. In this case loop 6–22 has to be considered. Let U_i be the σ-dag corresponding to S_i (see Theorem 4.8). Let g_i be the total number of product terms in U_i and let n_i be the number of nodes in the descendants of the product terms of U_i. Let v_i be the number of leaves in U_i and let d_i be the degree of U_i. Clearly, $d_i \leqq d$. Since $N(U_i) \cap N(U_j) = \varnothing \ \forall i \neq j$ (see Theorem 4.8) and since the number of nodes in $\sum_{i=1}^{h} x_i P_i$ is $\leqq n + h - 1$ (see Theorem 4.7), it follows that $\sum n_i \leqq n$. Let $T''(n_i, d_i)$ be the time required by loop 6–22 when processing U_i. Then the overall time complexity for TRANSFORM (r) is $\leqq 5C_2 n + \sum_i T''(n_i, d_i)$. We now claim the following:

CLAIM.

$$5C_2 n_i + T''(n_i, d_i) \leqq C_1(d_i) n_i.$$

Once we prove this claim it will follow that $T(n, d) \leqq C_1 dn$, which will complete the proof of the theorem.

Proof of claim. We treat the following two cases separately.

Case 1. $m = 1$. Lines 7 and 9 take constant time, and by the induction hypothesis, line 8 takes time $\leqq C_1(d_i - 1)n_i$. Hence, $T''(n_i, d_i) \leqq C_1(d_i - 1)n_i + 2C_2$. In this case, the proof of the claim follows from the assumption that $7C_2 < C_1$.

Case 2. $m > 1$. Line 7 takes $\leqq C_2 g_i$ time and by induction, line 8 takes time $\leqq C_1(d'')(n_i'')$, where d'' is the degree of P_{j_1}; n_i'', v_i'' are the number of nodes and leaves in P_{j_1} respectively. The execution time of lines 10 and 11 can be easily shown to be $C_2 g_i$ and $C_2 v_i''$, respectively. Hence, the time taken by steps 7–11 is

$$\leqq C_1(d'')n_i'' + 2C_2(g_i + v_i'').$$

Since $g_i \leqq n_i$ and $v_i'' \leqq n_i'' \leqq n_i$, we have that the above inequality is

$$\leqq C_1(d'')n_i'' + 4C_2 n_i \leqq C_1(d_i - 1)n_i'' + 4C_2 n_i.$$

Lines 12–19 can be easily shown to take time $\leqq C_2 n_i$. Let g_l' be the number of product terms in RS_l with n_l' descendants and v_l' leaves. Step 20 can be easily shown to take time $\leqq C_2 n_i + \sum_{l=1}^{w} C_1(d_i - 1)n_l'$, since the degree of each RS_l is $< d_i$. Line 21 takes time $\leqq C_2$. Collecting all the above facts we obtain:

$$\begin{aligned} T''(n_i, d_i) \leqq\ & C_1(d_i - 1)n_i'' + 4C_2 n_i && \text{(lines 7–11)} \\ & + c_2 n_i && \text{(lines 12–19)} \\ & + C_2 n_i - \sum_{l=1}^{w} C_1(d_i - 1)n_l' && \text{(line 20)} \\ & + C_2 && \text{(line 21).} \end{aligned}$$

Since $C_1 = 12C_2$, we have that

$$T''(n_i, d_i) + 5C_2 n_i \leqq C_1(d_i - 1)n_i'' + C_1 n_i + \sum_{l=1}^{w} C_1(d_i - 1)n_l'.$$

A straightforward implementation of line 14 can be used to show that $n_i'' + \sum_{l=1}^{w} n_l' \leqq n_i$. Hence, $T''(n_i, d_i) + 5C_2 n_i \leqq C_1(d_i - 1)n_i + C_1 n_i \leqq C_1 d_i n_i$.

This completes the proof of the claim and the theorem. □

Let us finally remark that the above transformation algorithm could output a tree which is not equivalent to the input dag, as the following example shows.

Example 4.2. Suppose we are given the expression

$$E = (((x * b) * c + (a * b) * c) + x * (e + d)) + a * (b * c + d),$$

whose dag is drawn in Fig. 4.3. In this case, the left factors are the variables x and a with corresponding right products $P_1 = b * c + (e + d)$ and $P_2 = b * c + (b * c + d)$. Therefore we have one connected component consisting of $\{P_1, P_2\}$. At step 8 of the procedure TRANSFORM, we have $n_0 \leftarrow b * c + (e + d)$ in which case n_0 has one factor Q_1. Now NORMAL-TERM $(Q_1) = d$ and $rr = a * (b * c + (b * c + d))$. The label assigned in line 12 to $a * P_2$ is 1 and the partial evaluation of rr produces $rr = a$. Therefore, the output will be $(a + x) * (b * c + (e + d))$, which is not equivalent to E.

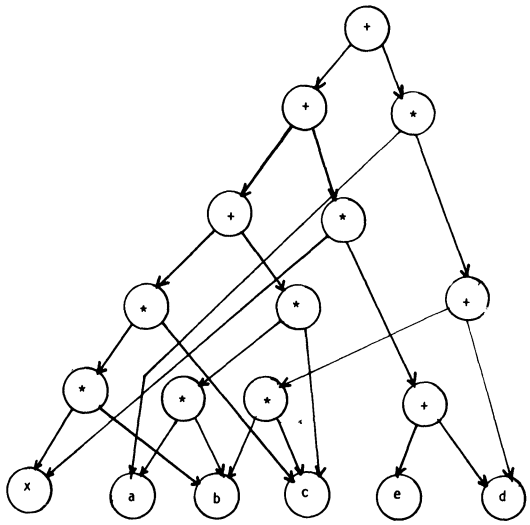


FIG. 4.3

5. The equivalence algorithm. As we have seen in the previous section, the transformation algorithm might generate a tree from a σ-dag which is not tree-transformable. Algorithms to check whether a given tree T and a given σ-dag D are equivalent are developed for several cases in this section. One way to solve this problem would be to find the normal terms of T and D and compare them; this is not efficient since the number of normal terms could be an exponential function of the number of nodes in D. The approach we take here is based upon the following characterization.

THEOREM 5.1. *A σ-dag D is equivalent to a tree T if and only if the following conditions are satisfied*:

(i) *Every normal term of D is a normal term of T.*

(ii) *The number of normal terms of D is equal to the number of normal terms of T.*

(iii) *No two normal terms of D are equal.*

In the rest of this section, we will examine the problem of designing efficient algorithms to check each of the above properties separately.

To handle (i), we will associate a graph with T, denoted by $G(T)$ in which a normal term induces a path and every path corresponds to a normal term. Before

doing so, we label the nodes of the tree by the following procedure. Initially, n_o represents the root of T, mark $(n_0) \leftarrow (0, 1)$, left $\leftarrow 0$, right $\leftarrow 1$ and next $\leftarrow 2$.

```
procedure LABEL-TREE (n0, left, right)
begin
global (next, mark [·]);
  case
    :n0 is a leaf [return];
    :n0 is a ⊛: [m ← next;
                 next ← next + 1;
                 mark (LC (n0)) ← (left, m);
                 call LABEL-TREE (LC (n0), left, m);
                 mark (RC (n0)) ← (m, right);
                 call LABEL-TREE (RC (n0), m, right);
                 return];
    :n0 is a ⊕: [mark (LC (n0)) ← mark (RC (n0)) ← (left, right);
                 call LABEL-TREE (LC (n0), left, right);
                 call LABEL-TREE (RC (n0), left, right);
                 return];
  endcase
end of procedure LABEL-TREE
```

To illustrate the ideas, we consider the expression $A = ((a + b * c) * f + b * (c * e + d * f)) + (a + b * d) * e$.

The tree generated by algorithm TRANSFORM is given in Fig. 5.1. The labels are as assigned by the above algorithm. With the labels given in Fig. 5.1, we can associate the following graph:

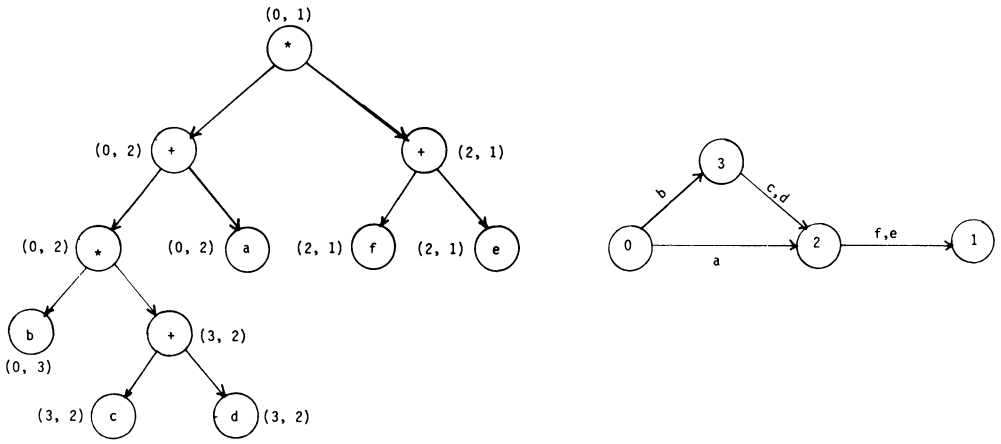


FIG. 5.1

Note that each normal term of the tree can be viewed as representing a path from 0 to 1 and vice versa.

In general, we can construct the graph $G(T)$ associated with a tree T as follows: suppose all of the leaves of T have been marked by the procedure LABEL-TREE. Create two nodes 0 and 1. If a leaf v is labeled (α, β), create nodes α and β (if

necessary) and draw a directed edge from α to β with label v; if such an edge already exists, simply attach the label v to it.[1]

The above graph could be thought of as the transition graph of a finite automaton with 0 as the initial state and 1 as the accepting state; the alphabet consists of the set of leaves of T. A word is accepted by this automaton if and only if it represents a normal term of the tree T.

We are ready to prove the following lemma.

LEMMA 5.2. *Let y be a node of the tree T whose descendants form a subtree T_1 and such that* mark $(y) = (\alpha, \beta)$. *Then, every normal term of T_1 is represented by a path from α to β in $G(T_1)$ and, conversely, every path form α to β in $G(T_1)$ represents a normal term of T_1.*

Proof. By induction on the height h of T_1.

If $h = 0$, y is a leaf and the result is obvious from the definition of $G(T_1)$.

Suppose now $h \geqq 1$. Two cases might arise:

(1) y is a $\oplus$ node with children generating subtrees C_1 and C_2, as shown in Fig. 5.2. Note that every normal term of T_1 is a normal term of either C_1 or C_2. Moreover, it is easy to see (from the definition of $G(T_1)$) that each edge in $G(T_1)$ is an edge in either $G(C_1)$ or $G(C_2)$, and conversely. Furthermore, no two edges of $G(C_1)$ and $G(C_2)$ are the same; therefore every path[2] from α to β in $G(T_1)$ is a path in either $G(C_1)$ or $G(C_2)$, and conversely. The proof follows now by induction for this case.

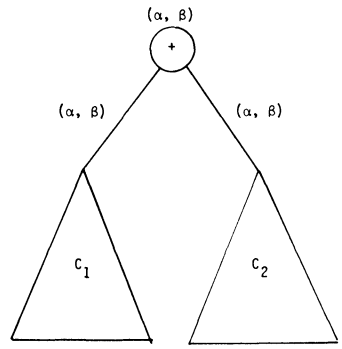


FIG. 5.2

(2) y is a $\circledast$ node with subtrees C_1 and C_2 (Fig 5.3(a)). Note that $G(T_1) = G(C_1) \cup G(C_2)$ as shown in Fig. 5.3(b) and where the edges of $G(C_1)$ are distinct from those of $G(C_2)$. Let $N_t(T)$ designate the set of normal terms of a tree T. Assume

$$N_t(C_1) = \{p_i | i = 1, \cdots, k_1\} \quad \text{and}$$

$$N_t(C_2) = \{q_j | j = 1, \cdots, k_2\}.$$

Then

$$N_t(T_1) = \{p_i * q_j | 1 \leqq i \leqq k_1, 1 \leqq j \leqq k_2\}.$$

[1] Note that we are actually constructing a series-parallel graph $G(T)$ from T. Every $\oplus$ causes a parallel connection and every $\circledast$ causes a series connection.

[2] As a sequence of edges.

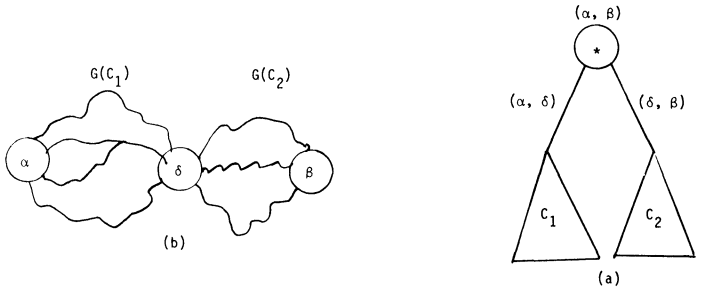


FIG. 5.3

Therefore each normal term of T_1 is of the form $p_i * q_j$; by induction, p_i is represented by a path from α to δ in $G(C_1)$ and q_j is represented by a path from δ to β in $G(C_2)$. It follows that $p_i * q_j$ can be represented by a path from α to β in $G(T_1)$.

Conversely, every path from α to β in $G(T_1)$ consists of two paths P_1 and P_2, where P_1 is a path from α to δ in $G(C_1)$ and P_2 is a path from δ to β in $G(C_2)$. The proof follows now by induction. □

COROLLARY 5.2.1. *Let $G(T)$ be the graph associated with a tree T. Every normal term of T is represented by a path from 0 to 1 in $G(T)$, and conversely, every such path represents a normal term of T.*

Suppose now we use the label of the leaves of T and assign them to the corresponding leaves of the dag D. If a node y of D has two children labeled, say, (α, β) and (α', β'), this means that the induced paths in $G(T)$ should match so that each normal term of y will be a consistent part of a normal term in T. It follows that if y is a $\oplus$ node, we must have $\alpha = \alpha'$ and $\beta = \beta'$; else, we must have $\beta = \alpha'$. Moreover, if every normal term of D is a normal term of T, then the root of the dag should get the label (0, 1). Formal proofs of these facts will be given after we present the procedure which implements the above policy.

```
procedure LABEL-DAG (n0)
begin
global mark ([·]);
  If n0 has been labeled then [return];
  If n0 is a leaf then [stop];
  call LABEL-DAG (LC (n0));
  call LABEL-DAG (RC (n0));
  (LX, LY) ← mark (n0));
  (RX, RY) ← mark (RC (n0));
  case
    :n0 is a ⊕ node: [If RX = LX and LY = RY
                        then mark (n0) ← (LX, LY);
                        else stop;
                     return];
    :n0 is a ⊛ node; [If RX = LY then mark (n0) ← (LX, RY);
                        else stop;
                     return];
  endcase
end of procedure LABEL-DAG
```

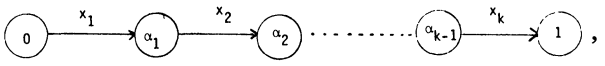
LEMMA 5.3. *Let $G(T)$ be the graph of a tree T. Suppose the leaves of a dag D are initialized with the same labels as those of T. If the dag could be labeled consistently up to a node p, with* mark $(p) = (\alpha, \beta)$, *then each normal term of p corresponds to a path from α to β in $G(T)$.*

Proof. By induction on the height h of the subdag induced by p. The case $h = 1$ is trivial. Suppose $h > 1$. We consider again two cases depending on whether p is a $\oplus$ node or a $\circledast$ node. The proof follows the same line as that of Lemma 5.2. □

COROLLARY 5.3.1. *If D could be labeled such that* mark $(r) = (0, 1)$, *where r is the root of D, then each normal term of D is a normal term of T.*

LEMMA 5.4. *Suppose that each normal term of D is a normal term of T, then* LABEL-DAG (r) *will terminate with* mark $(r) = (0, 1)$, *where r is the root of D.*

Proof. We will only prove that each product term of D will be labeled $(0, 1)$. Let P be a product term in D. Suppose that $w = x_1 * x_2 * \cdots * x_k$ (up to a fixed order) is a normal term in P, $x_i \in \Sigma$, $1 \leq i \leq k$. It follows that w is a normal term in T and hence there exists a path from 0 to 1 in $G(T)$ which represents w. Suppose this path is given by



i.e., mark $(x_i) = (\alpha_{i-1}, \alpha_i)$, $1 < i < k$, and mark $(x_1) = (0, \alpha_1)$, mark $(x_k) = (\alpha_{k-1}, 1)$.

Since w is a normal term in P, P must be of the following form: $P = (x_1 + D_1) * (x_2 + D_2) * \cdots * (x_k + D_k)$ up to a fixed order, where the D_is could be arbitrary subdags of D. We assume that the order of multiplications is as shown in Fig. 5.4; the same argument will hold for any other ordering. Now since x_1 is labeled

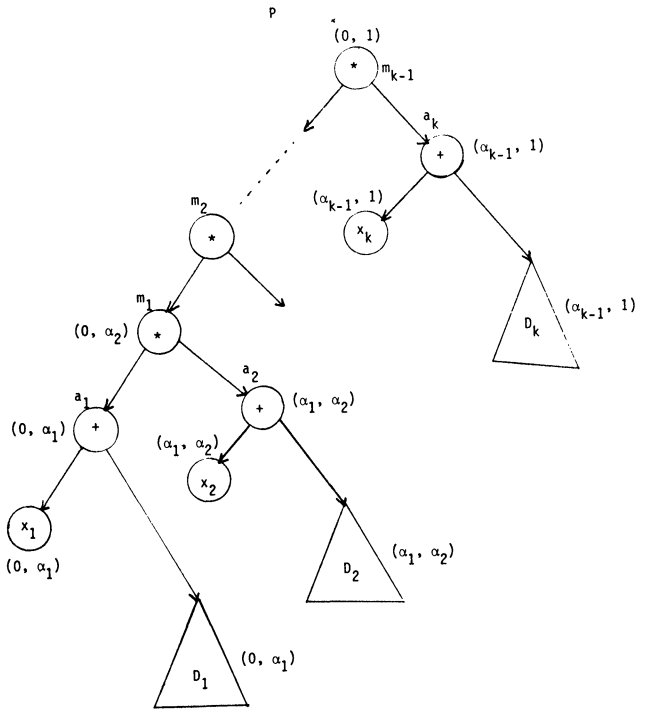


FIG. 5.4

$(0, \alpha_1)$, the root of D_1 must have the same label $(0, \alpha_1)$ and thus a_1 will have the label $(0, \alpha_1)$. Similarly, a_i will have the label (α_{i-1}, α_i), $1 < i < k$, where a_i is the root of the dag $x_i + D_i$; a_k gets the label $(\alpha_{k-1}, 1)$. Now m_1 is a multiplication node with children a_1 and a_2; thus it gets the label $(0, \alpha_2)$.
Using the argument $k-1$ times, we get that m_{k-1} has the label $(0, 1)$ and thus P has the label $(0, 1)$. This completes the proof of the lemma. □

We now collect the above facts in the following theorem.

THEOREM 5.5. *Given a tree T and a σ-dag D, it is possible to check whether each normal term of D is a normal term of T in $O(n)$ time, where n is the number of nodes in D.*

We now consider property (ii) of Theorem 5.1, namely checking whether the number of normal terms of T is the same as that of D. This is fairly easy(?) and the counting of normal terms in a dag D could be done by the procedure COUNT (r), where r is the root of D. Initially, all the nodes are not marked.

```
procedure COUNT (n0)
begin
global (C [·]);
    If n0 is marked then [return (C(n0))];
                  else mark n0;
    case
      :n0 is a leaf: [C(n0) ← 1];
      :n0 is a ⊛: [C(n0) ← COUNT (LC (n0)) * COUNT (RC (n0))];
      :n0 is a ⊕: [C(n0) ← COUNT (LC (n0)) + COUNT (RC (n0))];
    end case
    return (C(n0));
end of procedure COUNT
```

It is easy to prove the following lemma.

LEMMA 5.6. *Let D be any σ-dag with root r. Then* COUNT (r) *correctly computes the number of normal terms in D.*

As for the complexity, we have $O(n)$ steps, where n is the number of nodes in D. However, some steps might involve the multiplication of two large numbers, each of which might take considerably more than one "unit time." Before finding the number of bit operations required by the above algorithm, we establish an upper bound on the magnitude of the numbers used in COUNT.

LEMMA 5.7. *Let T be a tree with root r and such that*[3] $|L(T)| = v$. *Then $C(x) \leqq 2^{v/2}$, for all nodes x in T.*

Proof. By induction on the height h of T. □

COROLLARY 5.7.1. *Each $C(x)$ requires at most $v/2$ bits.*

Note that if we are considering the dag D and if at any one point we need more than $v/2$ bits to store any number, then we halt and declare that the numbers of normal terms are not equal. Therefore, the above upper bound holds true for D.

We are now ready to establish the complexity of the procedure COUNT.

THEOREM 5.8. *Let D be a σ-dag with root r. Then* COUNT (r) *takes $O(nv \log v \log \log v)$ bit operations, where n and v are respectively the numbers of nodes and leaves in D.*

[3] Recall that $L(T)$ represent the set of leaves in T.

Proof. The largest number in COUNT (r) require at most $v/2$ bits; adding such numbers could be done in $O(v)$ bit operations. Multiplying two such numbers could be done in $O(v \log v \log\log v)$ bit operations by using the Schönhage–Strassen integer-multiplication algorithm [SS]. Therefore COUNT (r) requires at most $O(nv \log v \log\log v)$ bit operations. □

Strangely enough, the above (rough) bound cannot be improved (under the assumption that multiplying two $k \times k$ bit numbers takes $k \log k \log\log k$ bit operations), i.e., there exist tree-transformable dags which will require $\Omega(nv \log v \log\log v)$ bit operations, as the following example shows.

Let

$$P_1 = (Z_1 + Z_2) * (Z_3 + Z_4) * \cdots * (Z_{2p-1} + Z_{2p}),$$

$$P_2 = (Y_1 + Y_2) * (Y_3 + Y_4) * \cdots * (Y_{2p-1} + Y_{2p}), \qquad Z_i, Y_i \in \Sigma.$$

Construct now the following dag D (Fig. 5.5).

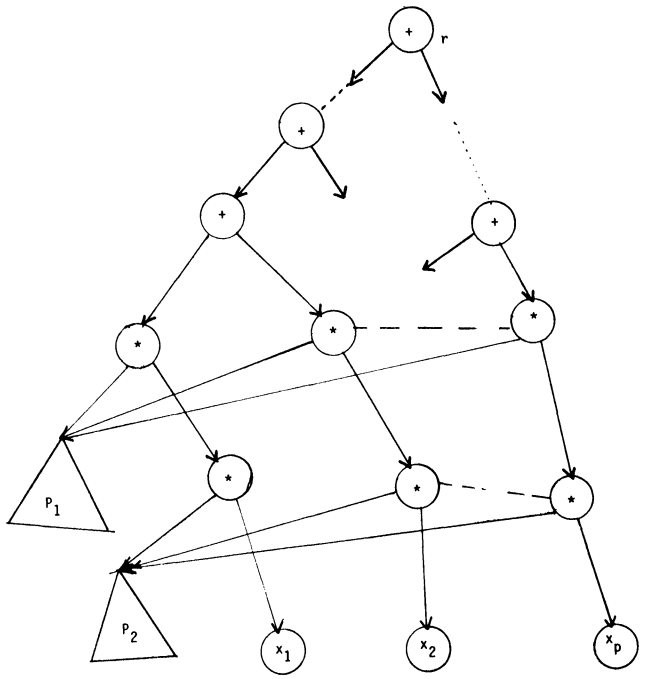


FIG. 5.5

In this case $|N(D)| = O(p)$, $|L(D)| = 5p$. However, COUNT (r) will have p multiplications, each of which occurs between two p-bit numbers. Therefore COUNT (r) requires at least $\Omega(nv \log v \log\log v)$ bit operations in this case.

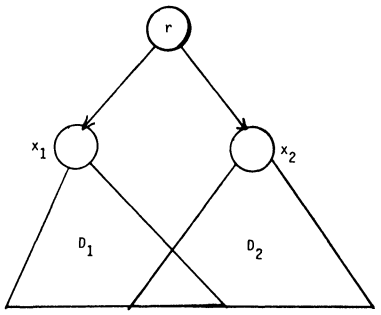
One might be tempted to say that it is possible to design another algorithm which does not multiply the same pair of numbers more than once and which will solve our problem in $O(n^2)$ time; however we can give another example where the multiplications involved are all between different numbers and yet the algorithm requires $\Omega(nv \log v \log\log v)$ bit operations. Let's remark that the execution time of this procedure dominates all the other parts of the equivalence algorithm.

We now consider the performance of COUNT on a special class of dags which will be considered later in more detail, namely that of leaf dags. COUNT is faster for this class even if we use the naive integer-multiplication algorithm as the following theorem shows.

THEOREM 5.9. *Let D be a leaf dag with root r. Then the execution time of* COUNT (r) *is of $O(n^2)$, where n is the number of nodes in D.*

Proof. Let e be the number of edges in D. Since $e = O(n)$, it follows that it is enough to prove that COUNT (r) takes $O(e^2)$ time. The proof is by induction on e, being trivial for $e = 1$.

Suppose $e > 1$. Let x_1 and x_2 be the left and right children, respectively, of r. x_1 and x_2 generate two dags D_1 and D_2 whose edges don't overlap. Let e_1 and e_2 be the numbers of edges in D_1 and D_2, respectively. Then $e = e_1 + e_2 + 2$. Thus the execution time $T(e)$ of the algorithm satisfies



$$T(e) \leqq T(e_1) + T(e_2) + O(e_1 e_2),$$

if we use the naive algorithm to multiply the number of normal times in x_1 by the ones in x_2. It follows that $T(e) = O(e^2)$. □

To terminate the equivalence algorithm, the problem of whether a given σ-dag has two identical normal terms will be investigated now. This is the hardest part of the equivalence algorithm and its complexity seems to depend crucially on two parameters: the degree of the expression and the type of sharing in the dag. If we restrict either one of these parameters, the problem becomes relatively easy and corresponding efficient algorithms can be developed. However, in the general case, the problem looks difficult and we feel that the general problem might be NP-complete. Therefore, we will attack this problem for two special cases: (i) the degree of the corresponding expression is bounded by a constant d and (ii) the given σ-dag is a leaf dag.

Before proceeding, we state the main result which has been obtained so far.

THEOREM 5.10. *Let D be an arbitrary σ-dag with no identical normal terms. Then checking whether D is tree transformable and obtaining an equivalent tree, whenever possible, could be done in $O(nv \log v \log\log v)$ time, where n and v are respectively the number of nodes and leaves in D.*

Proof. Immediate from Theorems 4.8, 4.9, 5.5 and 5.8. □

We now discuss the problem of identifying identical normal terms for leaf dags. We first transform the dag D into an equivalent dag D' which is *left-justified*, i.e., every ⊛ node of D' has a leaf as its left child. Figure 5.6 shows an example of a dag D with an equivalent dag D' which is left-justified.

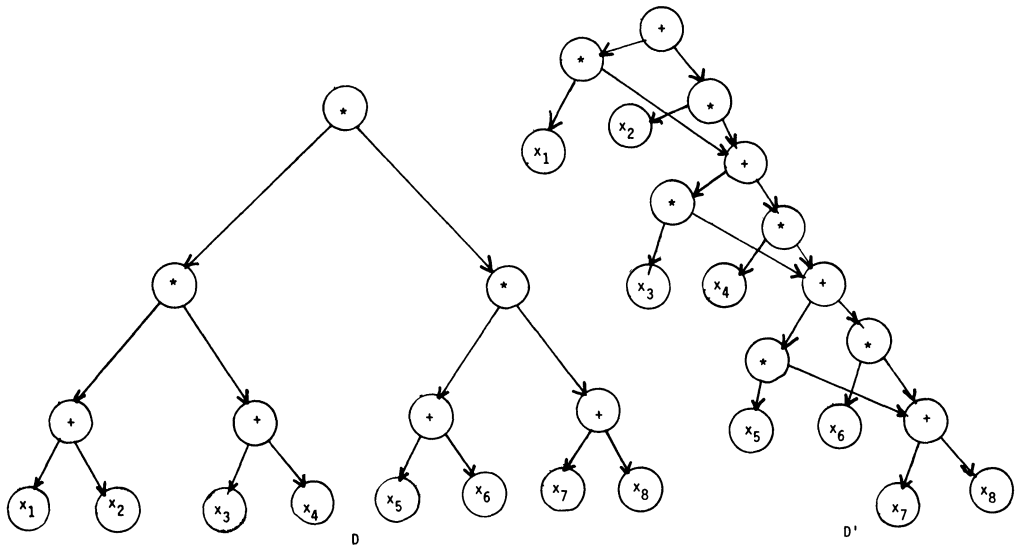


FIG. 5.6

We remark that in the above transformation no $\oplus$ node will be modified unless it is a left child of a $\circledast$ node. That is why the above transformation will increase the number of edges by, at most, a factor of 2, as we will later prove. The procedure to implement the above transformation is given below; pt ( ) has the same meaning as in § 4 and it is initialized to the empty dag, for all nodes of D. OP $(\cdot)$ denotes the operator of a node. The assignment $n_0 \leftarrow$ GETNODE means that a new node n_0 is created and PUTNODE (n_0) means that the node n_0 has been destroyed.

```
procedure LEFT-JUST (r)
begin //Left justify the leaf dag with root r which is multiplied by the left justified
dag pointed at by pt (r)//
    If r = leaf then [return]
  loop
    n2 ← LC (r);
    n3 ← RC (r);
  case
        :r is a ⊕: [If n2 = leaf then [If pt (r) ≠ ∅ then [n0 ← GETNODE;
                                                           OP (n0) ← '*';
                                                           LC (n0) ← n2;
                                                           RC (n0) ← pt (r);
                                                           LC (r) ← n0]]
                                   else [pt (n2) ← pt (r);
                                         call LEFT-JUST (n2)]
                   If n3 = leaf then [If pt (r) ≠ ∅ then [n0 ← GETNODE;
                                                          OP (n0) ← '*';
                                                          LC (n0) ← n3;
                                                          RC (n0) ← pt (r);
                                                          RC (r) ← n0;
                                                          return]
                                                  else [return]]
```

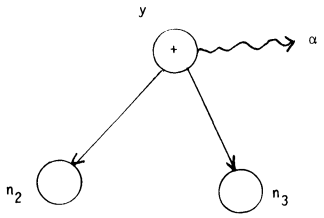
```
                    else [pt (n3) ← pt (r);
                          call LEFT-JUST (n3);
                          return]
                ];
:r is a ⊛: [If (n2 and n3 are leaves) then
              [If pt (r) = ∅ then [return];
              n0 ← GETNODE;
              OP (n0) ← '*';
              LC (n0) ← n3;
              RC (n0) ← pt (r);
              RC (r) ← n0;
              return]
           If (n2 is a leaf) then
                 [pt (n3) ← pt (r);
                 call LEFT-JUST (n3);
                 return]
           If (n3 is a leaf) then
                [OP (r) ← OP (n2);
                LC (r) ← LC (n2);
                RC (r) ← RC (n2);
                If pt (r) = ∅ then
                                [PUTNODE (n2);
                                pt (r) ← n3]
                           else
                                [OP (n2) ← '*';
                                LC (n2) ← n3;
                                RC (n2) ← pt (r);
                                RC (n2) ← pt (r);
                                pt (r) ← n2]
                      ]

           else
             [pt (n3) ← pt (r);
             call LEFT-JUST (n3);
             OP (r) ← OP (n2);
             LC (r) ← LC (n2);
             RC (r) ← RC (n2);
             pt (r) ← n3;
             PUTNODE (n2)]];

    endcase
  forever
end of procedure LEFT-JUST
```

The next lemma essentially establishes the correctness of the above procedure.

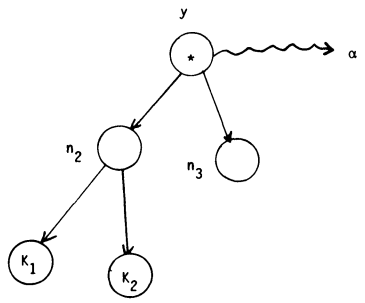
LEMMA 5.11. *Let D be a leaf dag and let y be any node of D such that* pt $(y) = \alpha$ *is left-justified. Then* LEFT-JUST (y) *will return a left-justified dag equivalent to the dag whose root is a* ⊛ *node with y and α as the left and right children, respectively.*

Proof. By induction on the depth of the recursive call. The proof involves eight different cases; we will only consider two typical cases, since all the others are similar to one or the other of the two.

1) y is a $\oplus$ node such that none of its children is a leaf. Two subcases have to be discussed separately.
   1.a) $\alpha = \varnothing$, then LEFT-JUST (y) returns by calling LEFT-JUST (n_2) and LEFT-JUST (n_3). By induction, the dags induced by n_2 and n_3 will be transformed into equivalent left-justified dags. It is easy to see that the dag with root y will then be left-justified.
   1.b) $\alpha \neq \varnothing$, then LEFT-JUST (y) makes the assignments pt $(n_2) \leftarrow$ pt $(n_3) \leftarrow \alpha$ and calls LEFT-JUST (n_2) and LEFT-JUST (n_3). By the induction hypothesis, we will have two left-justified dags equivalent to $(n_2 \circledast \alpha)$ and $(n_3 \circledast \alpha)$. Using the distributive law, it is easy to see that LEFT-JUST (y) will return a left-justified dag equivalent to $(y \circledast \alpha)$.



2) y is a $\circledast$ node such that none of its children is a leaf. Again, we will discuss two subcases separately. Note that the dag which is being transformed into an equivalent left-justified dag is given by $(n_2 \circledast n_3) \circledast \alpha$.
   2.a) $\alpha = \varnothing$, similar to 1.a) and 2.b).
   2.b) $\alpha \neq \varnothing$, LEFT-JUST (y) will first make the assignment pt $(n_3) \leftarrow \alpha$ and call LEFT-JUST (n_3). By the induction hypothesis, LEFT-JUST (n_3) returns a left-justified dag which is equivalent to $(n_3 \circledast \alpha)$. LEFT-JUST (y) will then make n_2 as the new node y with pt $(y) \leftarrow n_3$ and the infinite loop behaves as if it were really a recursive call to n_2 with pt $(n_2) = n_3$ (the "new" n_3 after it was modified by LEFT-JUST (n_3)). By the induction hypothesis, this should return a left-justified dag equivalent to $n_2 \circledast n_3$. □



COROLLARY. *Let D be a leaf dag with root r such that* pt (r) *is initialized to the empty dag. Then* LEFT-JUST (r) *returns a left-justified dag D' which is equivalent to D.*

The following theorem establishes the possible growth in the number of edges as well as the complexity of the procedure LEFT-JUST.

THEOREM 5.12. *Let D be a leaf dag with root r such that* pt (r) *is initialized to the empty dag. Then* LEFT-JUST (r) *returns a left-justified dag D' such that $|E(D')| \leqq 2|E(D)|$. Moreover, the execution time of this procedure is linear in the number of nodes in D.*

Proof. The proof is straightforward and will be left to the reader. □

Once we transform the given dag into an equivalent left-justified dag, the problem of checking the existence of two identical normal terms becomes easy. The main procedure, REPEATED-TERM, which, at each ⊕ node, calls another procedure EQUAL to check whether this node has two identical normal terms, is given below.

```
procedure REPEATED-TERM (n0)
begin
global (rmark [·])
    case
      :n0 has been rmarked: [return (rmark (n0))];
      :n0 is a leaf: [rmark (n0) ← false];
      :else: [rmark (n0) ← REPEATED-TERM (LC (n0))
                          or REPEATED-TERM (RC (n0));
            If rmark (n0) = false and n0 is a ⊕
                  then rmark (n0) ← EQUAL (LC (n0), RC (n0))];
    endcase
    return (rmark (n0));
end of procedure REPEATED-TERM
```

This procedure is fairly straightforward; it rmarks false all the leaves and proceeds from the bottom up, marking all nodes false until it meets a ⊕ node. It then calls the procedure EQUAL to check whether this ⊕ node has two identical normal terms in which case rmark (n_0) will be set true and this truth assignment will propagate to the root of the dag. Otherwise, everything will be set false and REPEATED-TERM returns false. We describe precisely the procedure EQUAL below.

```
procedure EQUAL (x, y)
begin
global (equal [·, ·]);
   If x = y then [return (true)]
   If (x, y)has been previously checked
            then [return (equal (x, y))]
   case
     :x and y are leaves: [equal (x, y) ← false];
     :x is a ⊕ or y is a ⊕:
       [Let w be one of the ⊕ nodes and let z the other node;
       equal (x, y) ← EQUAL (LC (w), z) or EQUAL (RC (w), z)];
     :x and y are ⊛:
       [If LC (x) ≠ LC (y) then equal (x, y) ← false;
            else equal (x, y) ← EQUAL (RC (x), RC (y))];
     :else: [equal (x, y) ← false];
   endcase
   return (equal (x, y));
end of procedure EQUAL
```

Before proving the correctness of the above procedure, we discuss the example mentioned at the end of § 4. The original expression $E = (((x * b) * c +$

$(a * b) * c) + x * (e + d)) + a * (b * c + d)$ has been transformed into the tree $T = (a + x) * (b * c + (e + d))$. Moreover, E could be transformed into an equivalent dag E' which is left-justified (Fig. 5.7). Note that n_0 and n_0' are the nodes created by the procedure LEFT-JUST.

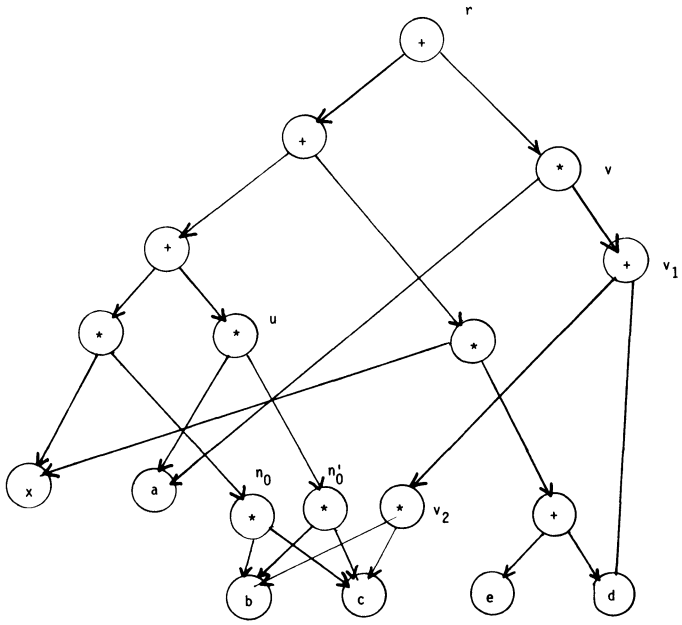


FIG. 5.7

Let us apply the algorithm EQUAL to the nodes u and v indicated in Fig. 5.7. Note that EQUAL (u, v) will be called at some point when REPEATED-TERM is applied to the above dag.

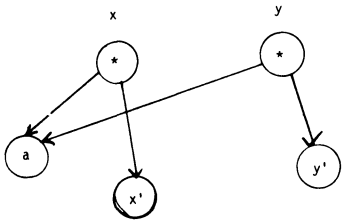
Recursive call	Result
EQUAL (u, v)	EQUAL (n_0', v_1)
EQUAL (n_0', v_1)	EQUAL (n_0', v_2) OR EQUAL (n_0', d)
EQUAL (n_0', v_2)	EQUAL (c, c)
EQUAL (c, c)	**true**
–	equal $(n_0', v_1) \leftarrow$ **true**
–	equal $(u, v) \leftarrow$ **true**

Therefore, EQUAL (u, v) returns true and REPEATED-TERM will also return true. Indeed, E does contain two duplicates of the normal term $a * b * c$.

The correctness of the above procedures is established in the next theorem.

THEOREM 5.13. *Let r be the root of the left-justified σ-dag D. Then* REPEATED-TERM (r) *returns true if, and only if, D has two identical normal terms.*

Proof. Suppose REPEATED-TERM (r) returns true, then it is easy to see by inspection that there exist two ⊛ nodes x and y such that EQUAL (X, Y) is true and



mark (X) = mark (Y).[4] The proof is now by induction on $h_x + h_y$, where h_x and h_y are the heights of x and y, respectively.

If $h_x + h_y = 2$, the proof follows easily.

Suppose $h_x + h_y > 2$. Then x and y must have the same left child, say a, and, moreover, EQUAL (x', y') must be true, where x' and y' are the right children of x and y, respectively. It is easy to check (since EQUAL (x, y) is true) that either the descendants[5] of both x' and y' contain a $\circledast$ node or none of the descendants of x' or y' is a $\circledast$ node. In the latter case, it is easy to check that the algorithm is correct. Thus, suppose the descendants of x' and y' contain $\circledast$ nodes, there exist two $\circledast$ nodes u and v such that EQUAL (u, v) is true and h_u and h_v are maximal among the $\circledast$ nodes which are descendants of x' and y', respectively. The rest of the proof follows by the induction hypothesis.

Suppose now that D has two identical normal terms. The proof is similar to that of Lemma 5.4, taking into consideration the fact that D is left-justified. □

THEOREM 5.14. *If D is a left-justified dag with root r such that n is the total number of nodes in D, then the execution time of* REPEATED-TERM (r) *is of* $O(n^2)$.

Proof. Note that if x and y are two nodes of D such that x has n_1 descendants and y has n_2 descendants, then EQUAL (x, y) takes at most $O(n_1 n_2)$ time. Moreover, for each pair of nodes x and y, EQUAL (x, y) is called at most once. The proof of the theorem follows from these observations. □

We collect all the facts we have established about leaf dags in the following theorem.

THEOREM 5.15. *Let D be a leaf dag with n nodes. Then it is possible to check whether D is tree-transformable or not, and to find an equivalent tree, whenever possible, in $O(n^2)$ time.*

This settles the case of leaf dags. Consider the case where the degree of the dag is bounded by a constant d. Then there are at most v^d normal term, where $v = |L(D)|$. Thus checking whether D has two identical normal terms could be done in $O(v^d)$ time. Note that, in this case, all the previous procedures run in linear time. Therefore, we have the following.

THEOREM 5.16. *Let D be an arbitrary σ-dag whose degree is bounded by a constant d. Then transforming D into an equivalent tree, whenever possible, could be done in $O(\max(|N(D)|, |V(D)|^d))$ time.*

Let us remark that if the level of sharing is a fixed constant then it is possible to transform the dag into a leaf dag in polynomial time. Therefore the corresponding problem can be solved efficiently in this case too.

Acknowledgment. We would like to thank the referees for their careful reading of the manuscript and for their constructive comments.

[4] REPEATED-TERM (r) returns also true if a $\oplus$ node has a leaf v as its left and right child. This case will be ruled out by the preceding algorithms.

[5] Including x' and y'.

REFERENCES

[AHU] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[AJ] A. V. AHO AND S. C. JOHNSON, *Optimal code generation of expression trees*, J. Assoc. Comput. Mach., 23 (1976), pp. 488–501.

[AJU] A. V. AHO, S. C. JOHNSON AND J. D. ULLMAN, *Code generation for expressions with common subexpressions*, J. Asssoc. Comput. Mach., 24 (1977), pp. 146–160.

[AU] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling. Vol. II: Compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[A] J. P. ANDERSON, *A note on some compiling algorithms*, Comm. ACM, 7 (1964), pp. 149–150.

[B] M. A. BREUER, *Generation of optimal codes for expression via factorization*, Comm. ACM, 12 (1969), pp. 333–340.

[BSe] J. L. BRUNO AND R. SETHI, *Code generation for a one-register machine*, J. Assoc. Comput. Mach., 23 (1976), pp. 502–510.

[DS] P. J. DOWNEY AND R. SETHI, *Variations on the common subexpression problem*, unpublished manuscript, 1977.

[GJ1] T. GONZALEZ AND J. JA'JA', *On the complexity of computing bilinear forms with {0, 1} constants*, J. Comput. Systems Sci., 20 (1980), pp. 77–95.

[GJ2] ———, *Computing arithmetic expressions with algebraic identities is hard*, in Proc. 1979 Conference on Information Sciences and Systems, March 1979, pp. 167–173.

[JMMW] D. B. JOHNSON, W. MILLER, B. MINNIHAN AND C. WRATHALL, *Reducibility among floating-point graphs*, J. Assoc. Comput. Mach., 26 (1979), pp. 739–760.

[K] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[N] I. NAKATA, *On compiling algorithms for arithmetic expressions*, Comm. ACM, 10 (1967), 492–494.

[R] R. R. REDZIEJOWSKI, *On arithmetic expressions and press*, Comm. ACM, 12 (1969), 81–84.

[SS] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation grosser Tählen*, Computing, 7 (1971), pp. 281–292.

[SU] T. SETHI AND J. D. ULLMAN, *The generation of optimal code for arithmetic expressions*, J. Assoc. Comput. Mach., 17 (1970), pp. 715–728.