Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages

OFF-LINE AND ON-LINE ALGORITHMS FOR DEDUCING EQUALITIES

Peter Downey[+]
The Pennsylvania State University, University Park, Pennsylvania 16802

Hanan Samet
University of Maryland, College Park, Maryland 20742

Ravi Sethi
Bell Laboratories, Murray Hill, New Jersey 07974

Abstract

The classical common subexpression problem in program optimization is the detection of identical subexpressions. Suppose we have some extra information and are given pairs of expressions $e_{i1}=e_{i2}$ which must have the same value, and expressions $f_{j1}\neq f_{j2}$ which must have different values. We ask if as a result, $h_1=h_2$, or $h_1\neq h_2$. This has been called the uniform word problem for finitely presented algebras, and has application in theorem-proving and code optimization. We show that such questions can be answered in $O(n\log n)$ time, where n is the number of nodes in a graph representation of all relevant expressions. A linear time algorithm for detecting common subexpressions is derived. Algorithms which process equalities, inequalities and deductions on-line are discussed.

1. Introduction

Some compilers will recognize that $b\times c$ is a subexpression of both $(a+b)/(b\times c)$ and $b\times c-d$. Such common subexpressions tend to arise when address computations are made explicit in commands like $A[i,j]:=B[i,j]-C[i,j]$. In the context of array references, another interesting phenomenon occurs; this time at the source level. Instead of looking for identical subexpressions, we need to find expressions that are equivalent subject to some conditions.

For example, whenever $a=b\div a$, as it is when $a=2$; $b=4$ or $a=-5$; $b=25$, then it follows that $a\times c=(b\div a)\times c$. A less direct implication is $a=b\div(b\div a)$. Iterating further, $a=b\div(b\div(b\div a))$, and so on.

In inferring $a\times c=(b\div a)\times c$ and $a=b\div(b\div a)$, we do not use any properties of division or subtraction. For that matter, whenever $i=\phi(j,i)$, then $\psi(k,\phi(j,i))=\psi(k,i)$, where ϕ and ψ can be any two functions.

In this paper we suppose we are given certain pairs of expressions $e_{i1}=e_{i2}$, $1\leq i\leq m$, called axioms or equalities, and ask whether these axioms together imply $g=h$, where g and h are some expressions. Also of interest is the case when we are given inequalities $f_{j1}\neq f_{j2}$, $1\leq j\leq n$, and ask whether $e_{i1}=e_{i2}$ for all i, and $f_{j1}\neq f_{j2}$ for all j, together imply either $g=h$ (the equality problem), or $g\neq h$ (the inequality problem).

The above problems arose during investigation by Downey and Sethi [DS] of optimization of programs that manipulate data structures. For example, consider the following simple program:

```
A[i]:=i+j;
A[k-ℓ]:=k-ℓ+j;
m:=A[i];
```

The value assigned to m is given by the conditional *if* $i=k-\ell$ *then* $k-\ell+j$ *else* $i+j$. The assignment to m can safely be replaced by $m:=i+j$; demonstrating this involves showing that whenever $i=k-\ell$, it follows that $k-\ell+j=i+j$. Both equalities and inequalities must be considered while simplifying the nested conditionals which occur with more complicated programs, or with multiple subscripts:

```
A[e1,f1]:=g1;
A[e2,f2]:=g2;
h:=A[e1,f1];
```

Here h is assigned *if* $e_1=e_2\wedge f_1=f_2$ *then* g_2 *else* g_1, which simplifies to g_1 if we can show that assuming $e_1=e_2$ and $f_1=f_2$, it follows that $g_2=g_1$.

Samet [Sal] considers the equivalence of Lisp-based programs where all predicates are tests for equality. Showing that two programs are equivalent again reduces to showing that two nested conditionals (with equality tests) are equivalent, thereby involving the equality and inequality problems. Shostak [Sh] considers a deductive system for solving these problems as a necessary component of a program verification system.

[+]The work of this author was partially supported by the National Science Foundation under Grant Number MCS75-22557.

The interactive deductive systems [Go] and the symbolic execution systems [Ki] used in program verification require efficient methods for deducing equality or inequality between expression instances. These applications require that a data base of "known" equalities be maintained to determine whether a new equality or inequality is consistent with, or deducible from, the current data base [NO1, NO2].

Our object is to give time efficient algorithms for deducing equalities or inequalities of expressions from axioms. We stress that the expressions considered are free of variables, and involve only constant symbols.

The following examples suggest the scope of the problems.

Example 1: Given that $x = y$, it is immediate that $\sin(x) = \sin(y)$. However, given $\sin(\pi) = \sin(3\pi)$, we cannot "climb down" and infer $\pi = 3\pi$. Similarly, given that $a \neq b \div a$, we cannot "climb up" and infer $a \times c \neq (b \div a) \times c$ since c may be 0. But given $a \times c \neq (b \div a) \times c$, it follows that $a \neq b \div a$. Using tests by contradiction, we can turn inequality questions into equality questions.

Suppose for example we are given $i \times k \neq (j \div i) \times k$, and are asked if as a result, $i \neq j \div i$?

For a test by contradiction, suppose that $i \neq j \div i$ is false i.e. in fact $i = j \div i$. It is immediate from $i = j \div i$ that $i \times k = (j \div i) \times k$, which contradicts the given statement $i \times k \neq (j \div i) \times k$. Consequently, our supposition $i = j \div i$ must be false i.e. $i \neq j \div i$ is true. □

As in Example 1, we show (Appendix A) that the inequality problem reduces to the equality problem. Furthermore, the given inequalities $f_{j1} \neq f_{j2}$ play a very minor role, so the equality problem reduces to the uniform word problem defined by: given axioms $e_{i1} = e_{i2}$, $1 \leq i \leq m$, does it follow that $g = h$?†

Example 2: Conceptually, the basic technique for testing equality is substitution of expressions. Given the following set of axioms, suppose we are asked whether $c \times d = (b \div a - d) \times d$:

$$c = (c \div b - a) \div a - d$$
$$b = c \div b - (d \div c - b)$$
$$a = d \div c - b$$

Substituting a for $d \div c - b$ in the second axiom, we infer $b = c \div b - a$. Substituting b for $c \div b - a$ in the first axiom yields $c = b \div a - d$. One final substitution of c for $b \div a - d$ in the expression $(b \div a - d) \times d$ yields $c \times d$, thereby answering "$c \times d = (b \div a - d) \times d$?" in the affirmative. □

The implementation of substitution raises a number of algorithmic issues, since we will employ some bookkeeping device to keep track of equivalent expressions rather than performing actual substitutions. The nature of the bookkeeping device depends critically on the representation of expressions.

Before discussing the merits of various representations, let us review what is known about the uniform word problem. The problem has long been observed to be decidable; Ackermann [A] gave an exponential decision algorithm. Kozen [Ko1] shows that the uniform word problem is logspace complete for P, the class of polynomial time recognizable languages, and gives a polynomial time algorithm. Kozen [Ko2] also shows that the problem requires $O(n/\log n)$ space in one "natural" proof system. Nelson and Oppen [NO1] have independently discovered an $O(n \cdot e)$ time algorithm using the methods of Section 5 below, have discussed the connection of the problem with theories of LISP-like data structures, and have implemented a simple polynomial-time algorithm for program verification studies.

In this paper, our object is to give improved time efficient algorithms for the uniform word problem.

In designing an algorithm there are three choices that need to be made: the representation of expressions, the representation of equality information between expressions, and the mechanism for deciding when an expression f can be derived from an expression e by substituting for a subexpression.

In Example 2, in order to infer $c \times d = (b \div a - d) \times d$, we need to establish the equality of the subexpressions c and $b \div a - d$, as well as b and $c \div b - a$. In general, even though we may be interested in testing the equality of expressions g and h only, it will be necessary to examine all subexpressions of g, h and the axioms. A tree representation makes it easy to refer to subexpressions. Each leaf in the tree represents an input name, and each non-leaf node represents an operator.

The representation of expressions we will actually use is a generalization of a tree called a directed acyclic graph (dag). Dags are formed from trees by collapsing identical subtrees. Figure 1 gives an example of a dag. In the context of dags we will refer to nodes rather than to expressions. Both nodes u and x represent the subexpression $b \div a$.

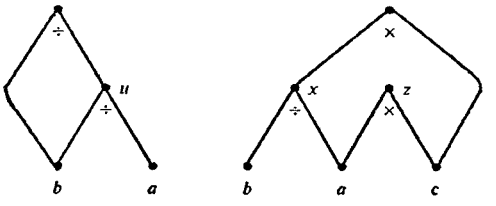

Figure 1: Collapsing identical subtrees yields a directed acyclic graph (dag). The above dag can be collapsed further.

†The precise term is uniform word problem for finitely presented algebras.

The dag representation of expressions arises naturally in the translation of sequences of assignments [AU, Cu]. A dag is also a much more economical representation -- there are dags with n nodes whose equivalent trees have 2^n nodes.

Consider the dag in Figure 1. The two leaves representing b necessarily have the same value, as do the two leaves representing a. One way of recording these facts is to assign a number, called a value number, to each node. Nodes with the same value will have the same value number. An axiom like $a = b \div a$ can be recorded by forcing the nodes representing a and $b \div a$ to have the same value number.

Since nodes u and x have the same operator, and their respective sons have the same value number, it follows that nodes u and x must have the same value. If nodes u and x initially had different value numbers, we must now change the value number of u to that of x (or vice versa). But there may be a set of nodes with the same value number as u and a set of nodes with the same value number as x, so we need to merge sets of equivalent nodes. The UNION-FIND algorithm of McIlroy, Morris and Tritter, shown by Tarjan [Tarj2] to have a practically linear running time, can be used for the purpose.

The final issue we must confront is the technique used to locate nodes like u and x in Figure 1 which have the same operator and whose corresponding sons are equivalent. Suppose leaves labelled b, a, c, in Figure 1 have value numbers 1, 2, 3, respectively. We can form a string for each node using the operator for the node and the value numbers of the sons of the node. Thus the strings for u and x will be $\div 12$ and the string for z will be $\times 23$. The problem now becomes one of locating identical strings. Cocke and Schwartz [CS] use hashing of strings which takes linear expected time, but quadratic worst case time. If the strings to be checked for equality are all available at the same time, then the lexicographic string sorting algorithms of [AHU] are guaranteed to take linear time, but hashing may be more attractive in practice. Methods which do not explicitly set up the strings to be sorted, but rely on controlled edge traversals are also available. The data structures needed are similar to those in Sethi [Se2] where an algorithm in the context of processor scheduling is given.

Our objective in this paper is to show how the solution of the uniform word problem reduces to these two issues: the maintenance of equivalence relations, and the detection of nodes like u and x which have the same operator and equivalent sons.

With operators like + and × there is a constant bound on the number of operands. For this case, the uniform word problem can be solved in $O(n \log n)$ time, where n is the number of nodes in a dag representation of the expressions to be tested. An algorithm is given in section 4. A linear algorithm is given to detect identical subexpressions in section 5.

Design of algorithms to handle processing of equalities on-line involves some different design choices which are discussed in section 6. Section 7 places the uniform word problem in perspective by relating it to results on other problems.

2. The Problem

An expression will be viewed as a tree, where each leaf is labelled with an input name chosen from a finite set S, and each nonleaf node is labelled with an operator chosen from a finite set Θ. As usual, the number of operands of each operator ψ in Θ is given by an integer $r \geq 1$ called the rank of ψ, and all nodes labelled with ψ have exactly r sons. The order in which the sons of a node appear is significant.

The semantics of expressions are also standard. Expressions denote elements of a set V of values. We define the value of an expression in terms of an interpretation $\underline{i}$ that maps each name to an element of V and operator ψ of rank r to a function from V^r to V.

Definition 1: The value under $\underline{i}$ of an expression e, denoted by $\underline{m}(e)$, is given by

1. $\underline{m}(a) = \underline{i}(a)$ if a is in S,
2. $\underline{m}(\psi e_1 \ldots e_r) = (\underline{i}\psi)(\underline{m}e_1, \ldots, \underline{m}e_r)$ if $e_1, \ldots, e_r$ are expressions.

An equality is a string of the form $e = f$, where e and f are expressions. The equality is satisfied under $\underline{i}$ provided $\underline{m}(e) = \underline{m}(f)$ under $\underline{i}$. Given a set of equalities A, interpretation $\underline{i}$ is said to satisfy A if every equality in A is satisfied under $\underline{i}$.

With these definitions we can formulate the uniform word problem† mentioned in the introduction.

UNIFORM WORD PROBLEM (for a single equality): Let $A = \{e_{i1} = e_{i2} \mid 1 \leq i \leq m\}$ and the equality $g = h$ be given. Determine whether $g = h$ is satisfied for all $\underline{i}$ satisfying A? □

Since the other problems mentioned in the introduction (equality, consistency, inequality) reduce to instances of the uniform word problem, we have chosen to defer a discussion of them to Appendix A.

There exist several ways of describing the uw problem, depending on one's point of view and favorite field. It can be phrased as: the problem of assigning identical value numbers to the nodes of a dag that are equivalent under a set of equalities; the problem of deriving all valid implications from a set of axioms [Kol]; the problem of calculating the smallest congruence generated by a set of axioms; or the decision

†We use the phrase "uniform word problem" as an abbreviation of the phrase "uniform word problem for finitely presented algebras". "Finitely presented" comes about because A is a finite set of equalities between variable-free expressions. The adjective "uniform" stresses that A is a parameter to the problem. The further abbreviated phrase "uw problem" will sometimes be used for "uniform word problem".

problem for the quantifier-free theory of equality with uninterpreted function symbols [A, NO1]. The formulation we will actually use facilitates proving the correctness and analyzing the complexity of algorithms for the problem.

Since we will represent expressions by dags, then nodes represent subexpressions. With each node x of a dag is associated a unique expression (tree) τx. The value $\underline{m}(x)$ of a node under $\underline{i}$ is $\underline{m}(\tau x)$. Let A be a given set of equalities. If node x_g has expression g, and if node x_h has expression h, then testing whether $g = h$ follows from A, reduces to testing if $\underline{m}(x_g) = \underline{m}(x_h)$ under all $\underline{i}$ satisfying A.

A is empty in the classic common subexpression problem, and we are interested not so much in determining if $g = h$, as in finding all redundant computations. The related statement in terms of dags is that we want to partition the nodes of a dag representing the relevant expressions so that two nodes u and x are in the same class if and only if $\underline{m}(u) = \underline{m}(x)$ under all $\underline{i}$ satisfying A. Such a partition defines an equivalence relation on nodes.

All equalities between expression will be translated to relations between nodes. A given set of equalities between expressions $A = \{e_{i1} = e_{i2} \mid 1 \leq i \leq m\}$ will be represented as a symmetric binary relation on the nodes of a dag. We agree to call this relation A also. Interpretation $\underline{i}$ is said to satisfy A if $\underline{m}(u) = \underline{m}(x)$ whenever uAx.

We can now restate the uw problem in the setting of dags.

UNIFORM WORD PROBLEM (full version): Given a dag D and a binary relation A on the nodes of D, find a relation R such that uRx if and only if $\underline{m}(u) = \underline{m}(x)$ under all $\underline{i}$ satisfying A. □

At this time, we will not pursue the distinction between the full version of the uw problem which determines a partition on the nodes of a dag, and the single equality version which checks two given nodes for equality. A similar distinction applies to finite automata, where the best known algorithms for the two versions have different time complexities: Hopcroft and Karp [HK] give a practically linear algorithm for determining equivalence of two finite automata, and Hopcroft [H] gives an $O(n \log n)$ algorithms for partitioning the states of a finite automaton into equivalence classes.

3. Basic Lemmas

Given $a = b$ and $b = c$ it immediately follows that $a = c$ since equivalence is transitive. Equally obvious is that we can always substitute equals for equals within an expression. These two observations form the basis of two transformations $\Rightarrow_t$ (transitive) and $\Rightarrow_c$ (collapsing or congruence). We will show in this section that $\Rightarrow_t$ and $\Rightarrow_c$ are exactly what is needed to solve the uw problem; efficiently computing the closure of these transformations then becomes our goal.

Consider the dag in Figure 1. The two leaves representing b necessarily have the same value, as do the two leaves representing a. Once leaves representing the same name have been related, we can infer that nodes u and x must have the same value under all interpretations. These observations lead to the following collapsing transformation $\Rightarrow_c$.

Definition 2: Let R be a binary relation on the nodes of dag D. R transforms to $R \cup \{(u,x)(x,u)\}$ under $\Rightarrow_c$ if and only if uRx is false, and

1. u and x are leaves representing the same name a, or
2. u and x representing the same operator ψ, have sons $w_1, w_2, \ldots, w_r$ and $y_1, y_2, \ldots, y_r$, respectively, and for all j, $1 \leq j \leq r$, $w_j R y_j$, or $w_j = y_j$. □

We also need a transitive transformation.

Definition 3: Let R be a binary relation on the nodes of a dag D. R transforms to $R \cup \{(u,x), (x,u)\}$ under $\Rightarrow_t$ if and only if uRx is false, and for some node z, uRz and zRx are both true. □

Since we will apply both $\Rightarrow_c$ and $\Rightarrow_t$ we use $\Rightarrow$ to denote $\Rightarrow_c \cup \Rightarrow_t$ i.e. $P \Rightarrow R$ if and only if $P \Rightarrow_c R$ or $P \Rightarrow_t R$. By definition, an application of $\Rightarrow$ adds a pair of nodes to a relation, so starting with any relation P, $\Rightarrow$ can only be applied a finite number of times in succession. We say P is irreducible under $\Rightarrow$ if and only if for all R, $P \Rightarrow R$ is false. We write $P \Rightarrow {}^{\sim}R$ when $P \Rightarrow^* R$ and R is irreducible under $\Rightarrow$.†

A major advantage of not fixing the order of application of $\Rightarrow_c$ and $\Rightarrow_t$ is that the correctness of any algorithm that applies $\Rightarrow_c$ and $\Rightarrow_t$ in any order, until no longer possible, follows from Theorem 1. The price to be paid for this advantage ia a modest one: we have to verify that the order in which the transformation $\Rightarrow$ adds pairs does not matter. More precisely, from the properties of $\Rightarrow_c$ and $\Rightarrow_t$, we can verify that if $P \Rightarrow^* R$ and $P \Rightarrow^* S$ by adding different pairs to P, then there exists T such that $R \Rightarrow^* T$ and $S \Rightarrow^* T$. Thus if $P \Rightarrow {}^{\sim}T_1$ and $P \Rightarrow {}^{\sim}T_2$, we can

† $P \Rightarrow^0 R$ if and only if R is P. For $i > 0$, $P \Rightarrow^i R$ if and only if $P \Rightarrow P'$ and $P' \Rightarrow^{i-1} R$. We write $P \Rightarrow^* R$ if $P \Rightarrow^i R$ for some $i \geq 0$.

show that T_1 must be the same as T_2. A system such as the one we are working with in which each object transforms to a unique irreducible element is said to have the finite Church Rosser property (See Sethi [Se1] for details).

Since irreducible elements are unique under $\Rightarrow$, we will write $\tilde{P}$ for the irreducible relation that P transforms to under $\Rightarrow$.

The rest of this section shows (Theorem 1) that starting with a given relation A, the solution to the full version of the uniform word problem is given by the relation $\tilde{A}$.

LEMMA 1: For all nodes u and x, $u\tilde{A}x$ implies that $\underline{m}(u) = \underline{m}(x)$ under all $\underline{i}$ satisfying A.

Proof: Let A_0 be A, and let $A_0, A_1, \ldots, A_k$ be some sequence of symmetric relations such that for $i > 0$, $A_{i-1} \Rightarrow A_i$, and $A_k = \tilde{A}$. We prove the lemma by induction on the least $\underline{i}$ such that $uA_i x$.

basis, $i = 0$: Then uA_0x, so by definition, $\underline{m}(u) = \underline{m}(x)$ under all $\underline{i}$ satisfying A.

inductive step, $i > 0$: If $A_{i-1} \Rightarrow_t A_i$, then there exists z such that $uA_{i-1}z$ and $xA_{i-1}z$. From the inductive hypothesis, $\underline{m}(u) = \underline{m}(z) = \underline{m}(x)$ under all $\underline{i}$ satisfying A. If $A_{i-1} \Rightarrow_c A_i$, then u and x either represent the same input name, or they represent the same operator, and (from the inductive hypothesis) son w_j of u has the same value as son y_j of x, for all j. The lemma follows. □

The interesting part is showing that the converse of the above lemma is true. In other words, if u and x have the same value under all interpretations satisfying A, then $u\tilde{A}x$. Some insight can be gained by considering the case where no two nodes are assumed equal so A is the empty relation. Then we can use a "free" interpretation that assigns the expression of node u as the meaning of u, so that u and x have the same value exactly when they have identical expressions. Since the transformation $\Rightarrow$ collapses identical subexpressions, the last statement translates to "...exactly when $u\tilde{A}x$". A similar idea works with any A. We will proceed by defining a particular interpretation $\underline{i}_A$ satisfying A, under which nodes u and x will have the same value exactly when u and x are in the same equivalence class under $\tilde{A}$.

Definition 4: Given a relation A let $\tilde{A}$ partition the nodes of D into i equivalence classes. Assign an integer from 1 through i to each equivalence class: let $\eta(x)$ give the number of the equivalence class for node x. Define $\underline{i}_A$ as follows:

1. If some leaf x represents name a, then let $\underline{i}_A(a) = \eta(x)$.
2. If some node x represents operator ψ, and has sons $y_1, y_2, \ldots, y_r$, let the function $\underline{i}_A\psi$ map $(\eta y_1, \eta y_2, \ldots, \eta y_r)$ to $\eta(x)$. □

Before we use the above definition, we must verify that $\underline{i}_A$ is a function.

LEMMA 2: $\underline{i}_A$ is an interpretation satisfying A.

Proof: By definition, all leaves representing the same name are in the same equivalence class, so for any name a, $\underline{i}_A(a)$ is unique.

Suppose that there are two nodes u and x representing the same operator ψ, with sons $w_1, w_2, \ldots, w_r$ and $y_1, y_2, \ldots, y_r$, respectively, such that $\eta w_j = \eta y_j$. Since $\tilde{A}$ is irreducible under $\Rightarrow_c$, it follows that $\eta u = \eta x$. Thus $\underline{i}_A\psi$ is a function.

Since A is a subset of $\tilde{A}$, if uAx, we have $u\tilde{A}x$, so $\eta u = \eta x$. Therefore $\underline{i}_A$ satisfies A. □

THEOREM 1 (Completeness of $\Rightarrow$): For all nodes u and x in D, $u\tilde{A}x$ if and only if $\underline{m}(u) = \underline{m}(x)$ under all $\underline{i}$ satisfying A.

Proof: One direction is Lemma 1. Conversely, suppose $\underline{m}(u) = \underline{m}(x)$ under all $\underline{i}$ satisfying A. Then $\underline{m}(u) = \underline{m}(x)$ under $\underline{i}_A$. By the definition of $\underline{i}_A$, it is clear that $\underline{m}(z) = \eta(z)$ under $\underline{i}_A$ for all z. Thus $\eta(u) = \eta(x)$ and $u\tilde{A}x$. □

Relation $\tilde{A}$ which is formed from A by taking the closure under $\Rightarrow_c$ and $\Rightarrow_t$ is the minimal congruence relation on expressions which satisfies A [Ko1, NO1]. In an independent development Shostak [Sh] gives a result similar to Theorem 1, using a least binary relation closed under conditions similar to $\Rightarrow_c$ and $\Rightarrow_t$. The interpretation $\underline{i}_A$ of Definition 4 corresponds to the Herbrand model given by Shostak. Our use of transformations makes it easy to connect Theorem 1 with algorithms.

4. An O(n log n) Algorithm

From Theorem 1, the uniform word problem can be solved by starting with a given set of equalities A and applying transformations $\Rightarrow_c$ and $\Rightarrow_t$ in any order, until no longer possible, to yield $\tilde{A}$. For then two nodes are in the same equivalence class of $\tilde{A}$ if and only if they have the same value subject to the equalities of A.

If we can find a sequence of ever "coarser" equivalence relations that starts with A and converges to $\tilde{A}$, then we can give an easy upper bound on the length of the sequence. Given a symmetric relation A, let A_0 be the equivalence relation formed by taking the reflexive, transitive closure of A. Consider a sequence of equivalence relations $A_0, A_1, \ldots, A_k$ on the

nodes of D such that $A_j \Rightarrow^* A_{j+1}$ and A_k is irreducible. Then A_k must be $\tilde{A}$. Clearly, A_{j+1} is formed by merging equivalence classes of A_j. If for each j, A_{j+1} has at least one less equivalence class than A_j, then we are assured that $k \leq n$, where n is the number of nodes of D.

For operators like + and × there is a constant bound (two) on the number of operands. For simplicity of exposition we will give the algorithm and its analysis assuming all operators have rank two.

Algorithm Atilde in Figure 2 starts by partitioning the nodes into equivalence classes based on A_0, the reflexive, transitive closure of the given equalities A. All nodes in a particular equivalence class are given a common value number, represented by the array VN indexed by nodes. The nodes are then processed in some topologically sorted order. As a new node u is encountered, a string σ consisting of the operator of u followed by the current value numbers of the sons of u, is constructed. For the purposes of this discussion, the strings are hashed into a table ACL (for Available Computations List). If σ is in the table, then there must be some other node x with the same string σ, so the equivalence classes for u and x have to be merged by procedure Merge. When value numbers are changed to put u and x in the same class, the strings for the fathers may have changed, so procedure Merge examines the fathers.

Let D be a dag with n vertices and e edges. Let A be a set of node pairs, and let $|A|$ give the number of elements of A.

LEMMA 3: Algorithm Atilde correctly computes $\tilde{A}$.

Proof: The value numbers in the array VN define an equivalence relation on nodes. Let R_i be the equivalence relation defined by VN just before the $i+1$st call to Merge.

It is clearly the case that $R_{i-1} \Rightarrow^* R_i$, so that uR_ix only if $u\tilde{A}x$.

Conversely, let $u\tilde{A}x$. Let A_0 be the reflexive closure of A and let $A_0, A_1, \ldots, A_k$ be some sequence of relations such that $A_{i-1} \Rightarrow A_i$ and $A_k = \tilde{A}$. We prove by induction on i that: for all i there is some r such that uA_ix implies uR_rx.

Suppose uA_ix.

basis, $i = 0$: Here $r = 0$ suffices.

inductive step, $i > 0$: If $A_{i-1} \Rightarrow_t A_i$ then there exists z such that $uA_{i-1}z$ and $zA_{i-1}x$. From the induction hypothesis, there is a r such that uR_rz and zR_rx, so that uR_rx. If $A_{i-1} \Rightarrow_c A_i$ then u and x have the same label and their corresponding sons are related by A_{i-1}. From the induction hypoyhesis, corresponding sons are related by R_r, for some least integer, r, i.e., they have the same value number just before the $r+1$st call to Merge.

Consider the rth call to Merge. From the choice of r, a pair of corresponding sons of u and x get placed in the same equivalence class by being assigned the same value number. If both u and x have been marked by Atilde, then there is an appropriate string for each of u and x on ACL. When all nodes on FATHERS(j) are considered in Merge, u and x will be placed in the same equivalence class. Alternately, if u and x have not both been marked by Atilde, then when the last of u and x to be marked is subsequently processed by Atilde, it will be placed in the same equivalence class as the other node.

The above induction establishes that if $u\tilde{A}x$ then uR_rx for some r, and at termination u and x have the same value number. □

THEOREM 2: Given a dag D and a set of equalities A, let D have n nodes. Assume each nonleaf node has two sons. Algorithm Atilde determines $\tilde{A}$ in $O(|A| + n \log n)$ time.

Proof: From Lemma 3, Algorithm Atilde determines $\tilde{A}$, so we just need to verify the time bound. Step 1 requires $O(n + |A|)$ time since each equality must be looked at, and the components of an undirected graph can be found in linear time [Tarj1]. Aside from the time taken to check entries on ACL and the time spent in Merge, the time taken in the while loop is proportional to the number of nodes.

In order to ensure that an entry can be added or retrieved from ACL in constant time, organize the ACL as a trie [Kn]. Each node in the trie will be an array of length n, with each element of the array being a pointer to the appropriate son of the node. Since there is a constant bound 2 on the number of sons of a node, the length of each string is bounded by 2. Given strings of length 2 the space required by the trie is $O(n^2)$. We avoid initialization costs using a pointer stack as in [AHU, exercise 2.12]. Thus the overall time taken in procedure Atilde is linear.

Consider the calls to procedure Merge. Calls of the form Merge (k,k) take constant time and can be charged to the point at which the call occurs. If $|\text{FATHERS}(k)| < |\text{FATHERS}(j)|$, then in constant time the call Merge (k,j) is converted to Merge (j,k). Since the value numbers define an equivalence class on nodes, there may be at most n calls in which value numbers are changed. So all we have to verify is that the total time spent over these n calls is $O(n \log n)$.

When two classes are merged, we are careful to merge the class with fewer fathers into the other class. Thus, every time a father x is

reprocessed because of son y, the FATHERS list for the class of y doubles in size. Since there are a linear number of edges in the dag, and the number of entries on a FATHERS list is limited by the number of edges, node x is reprocessed at most $O(\log n)$ times because of y. Since there are at worst two sons of x, node x is reprocessed at most $O(\log n)$ times. It follows that the total time charged to Merge is $O(n \log n)$.

Summing the time taken by Atilde and Merge, we get $O(|A| + n \log n)$. □

```
/*Input:  A dag D and a set A of node pairs
     that are given to have the same value. The
     nodes of D are assumed to have been given
     numeric identifiers in the range 1 to n, so
     that they can be bucketed efficiently. Since
     lexical analyzers normally assign internal
     names to variables, this assumption is not
     felt to be unreasonable.

Output: Equivalence Ã where xÃy if and only if
     VN(x) = VN(y), i.e. x and y have the same
     value under all interpretations satisfying A.

Structures: VN(1:n) maintains the value numbers of
     nodes. Each element of arrays FATHERS (1:n) and
     NODES(1:n) is a linked list of nodes. ACL, the
     available computations list, contains pairs
     ⟨σ, m⟩, where σ is a string, and m is a
     value number. The two operations performed on
     ACL are the insertion of a new entry on ACL,
     and the retrieval of an entry for a string σ.
     G(1:n) is the adjacency structure for an un-
     directed graph. NX(D) enumerates the nodes
     of D in topological order, starting with the
     leaves. It returns null when there are no
     more nodes.  */

procedure Atilde (D,A)
/*initialize A0 */

1.   for each pair (x,y)εA do add edge (x,y) to
        G end
     Locate the connected components of G using
        depth-first search.
     Set the VN of each node in the first
        component to 1, in the second component to
        2, etc.

2.   x:=NX(D);
     while  x ≠ null do
                Mark x.
                Let ℓ, r be the sons of x,
                   left to right.
                Add x to the list.FATHERS(VN(ℓ))
                   and FATHERS (VN(r))/* if x
                   is a leaf, do nothing*/
                if the ACL has an entry for
                   σ = OP(x)VN(ℓ)VN(r)
                   then Merge (VN(x),m), where
                      m is the value number
                      entered for σ
                   else insert ⟨σ, VN(x)⟩ on ACL
                      and add x to the list
                      NODES(VN(x)).
                x:=NX(D)
                end
     end Atilde;
```

```
procedure Merge (k,j):

     if k = j then return
     elseif |FATHERS(k)| < |FATHERS(j)| then
        Merge(j,k)
     else for all y on NODES(j) do VN(y):=k end;
        for all x on FATHERS(j) do
            Let x have sons ℓ, r left to right.
            if the ACL has an entry for
               σ=OP(x)VN(ℓ)VN(r)
                 then Merge (VN(x),m), where m is
                    the value number entered for σ.
            end
        Append list NODES(j) to NODES(k).
        Append list FATHERS(j) to FATHERS(k).
     end Merge;
```

Figure 2: Algorithm for the uniform word problem

In practice, ACL might well be implemented as a hash table, not as a trie. But then only the expected time to search the ACL is constant, not the worst case time.

To extend the above algorithm to the case of dags D with other than binary operators, we use an observation due to Tarjan [Tarj3]. Replace each node x with label ψ and r sons by a chain of binary nodes and dummy operator symbols, as suggested in Figure 3. The resulting dag D' can be processed by Algorithm Atilde. Since every r-ary node of D gives rise to r-1 binary nodes in D', then D' has no more vertices than D has edges. This yields an $O(e \log e + |A|)$ time overall.

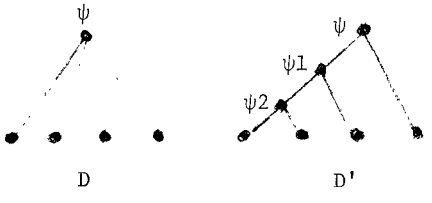

Figure 3: Transforming r-ary to binary nodes.

Suppose that the operators Θ are assumed to be commutative, i.e., the ordering on the sons at each node is immaterial. We can readily extend Algorithm Atilde to cover this case.

COROLLARY 1: When operators are commutative, Ã can be computed from A without affecting the time bound.

Proof: Instead of constructing string $\sigma = OP(x)VN(\ell)VN(r)$ in procedures Atilde and Merge, sort VN(ℓ), VN(r) to get string v_1v_2 in decreasing order. Use string $OP(x)v_1v_2$ instead of σ when checking strings against ACL.

The time bound does not change. □

5. A Linear Algorithm for the Common Subexpression Problem

For particular cases of the uniform word prob problem, more efficient algorithms can be written. If the set A of pairs is empty then xAy if and only if the trees rooted at x and y are identical. Such nodes are called strongly equivalent. The problem of computing $\tilde{A}$ has been called "common subexpression detection". Cocke and Schwartz [CS] use hash table searching to solve this problem in expected linear time, and worst case time $O(n^2)$. Here we give an algorithm, Partition, with linear performance in worst case.

Let the height of a node x in D be the length of the longest path from x to a leaf. Any two strongly equivalent nodes of D must be at the same height. Thus Algorithm Partition needs to discriminate only nodes of a given height. Each node is assigned a value number which never changes. To assign value numbers to nodes at height i, the algorithm needs to lexicographically sort only the height i nodes, and needs to use only as many buckets as these nodes have sons. The algorithm is given in Figure 4.

Algorithm Partition generalizes the algorithm from Aho et. al. [AHU, p. 84] for determining whether two labelled, unordered trees are isomorphic. Algorithm Partition solves the "subdag isomorphism problem" for labelled, ordered dags (and for trees as a special case).

THEOREM 3: Let A be an empty relation on the nodes of dag D, with n nodes and e edges. Algorithm Partition correctly determines $\tilde{A}$ in time $O(n + e)$.

Proof: (correctness). Any two strongly equivalent nodes must be at the same hieght. On the first iteration, the leaves are correctly identified. Assuming Partition has identified strongly equivalent nodes at height i, it is easy to see that the strongly equivalent nodes at height $i + 1$ are correctly identified.

(analysis). Let n_i be the number of nodes of height i. Let e_i be the total number of edges from nodes of height i to their sons. Clearly $\Sigma n_i = n$ and $\Sigma e_i = e$.

Consider the loop body. Step 2 takes time $O(n_i + e_i)$ on iteration i. The size of the LIST built is $|\text{LIST}| \leq n_i + e_i$. Step 3 adds to QUEUE strings with total length $n_i + e_i$ in time $O(n_i + e_i)$. Thus Step 4 sorts strings of total length $n_i + e_i$ using $|\text{LIST}|$ buckets. This takes $O(n_i + e_i + |\text{LIST}|) = O(n_i + e_i)$ time. Step 5 can clearly be done in $O(n_i + e_i)$, and Step 6 is bounded by $O(|\text{LIST}|)$. The loop body takes $O(n_i + e_i)$ overall. Since Step 1 takes $O(n)$, the Algorithm requires $O(n + \Sigma_i(n_i + e_i)) = O(n + e)$ time total. □

```
/*Input: A dag D with n nodes. The leaves of
     D have labels from a set S of names, and
     the nonleaves have labels from a set Θ.
     For convenience assume each element ψ of
     Θ ∪ S has been assigned an integer NR(ψ)
     between 1 and n.

Output: An array VN(1:n) where x is strongly
     equivalent to y if and only if VN(x) = VN(y).

Structures: BUCKET(1:n) and QUEUE are needed for
     lexicographic sorting. BBIT(1:n) is an array
     of bits. LIST is a linked list of bucket
     indices. */

procedure Partition(D)

1.   Find the height of all nodes in D. Let
       h = maximum height of any node.

     i:=0
     COUNT:=1
     LIST:=null
     Set all BUCKETS to null
     Set all BBITS to false
     QUEUE:=null

while i ≤ h do

   2. Scan the nodes at height i, collecting in
        LIST the value numbers of sons of these
        nodes. Set BBIT(i) each time value
        number i is encountered and add i to
        LIST only the first time. Add NR(ψ)
        to LIST for each distinct element ψ
        of Θ ∪ S encountered.

   3. For each node x at height i with label
        ψ and sons y_1, ..., y_r, add string
        NR(ψ)VN(y_1) ... VN(y_r) to QUEUE. (Note
        r = 0 for leaves.) Call this the string
        of x.

   4. Lexicographically sort the strings on
        QUEUE, using only the BUCKETS with
        indices on LIST.

   5. Scan the QUEUE, assigning distinct suc-
        cessive value numbers to those nodes with
        distinct strings on QUEUE. Increment
        COUNT accordingly.

   6. for each i on LIST do
        unset BBIT(i)
        BUCKET(i):=null
        end
      LIST:=null
      i:=i+1

   end
return (VN)
end Partition
```

Figure 4: Algorithm for Common Subexpression Detection

Algorithm Partition can be used as a subroutine to provide an algorithm for the uniform word problem, that does not require hashing or tries as in the proof of Theorem 2. Nelson and Oppen [NO1] have independently discovered an algorithm with the same time bound.

COROLLARY 2: Given a dag D with n nodes and e edges, and a set of equalities A, the uniform word problem can be solved in $O(n \cdot e + |A|)$ time.

Proof: See Nelson and Oppen [NO1]. □

In the special case of the uniform word problem in which there is given a single equality $e = f$ between expressions, linear time suffices. This problem arises in Downey and Sethi [DS] where a restricted class of programs with array assignments is studied.

COROLLARY 3: Let A contain a single pair of nodes $A = \{(u,v)\}$. Then $\tilde{A}$ can be determined in time $O(n+e)$.

Proof: Let D_u and D_v be the subdags of D rooted at u and v respectively. If neither of u and v is an ancestor of the other, then redirect all edges coming into u or v to a new node x and run Partition on the resulting modified dag.

Suppose that v is an ancestor of u in D. Then D_u is a subdag of D_v. Call two nodes x and y equivalent if $x \tilde{A} y$. A node equivalent to v will be called a v-node. For each node x in D define the v-height of x to be the eight of node x above its closest v-node descendant. Nodes with no descendant v-nodes have v-height ∞. Nodes u and v have v-height zero.

Let x be a node in D of height i and v-height j. It is easy to see that if x is equivalent to any node of height less than i, then x is equivalent to some node y of the same v-height j in D_v.

Given D, we can run Partition on D and collapse strongly equivalent nodes. So assume that D is given with distinct nodes having distinct expressions. Run Partition on subdag D_v, assigning value numbers. Assign the same value number to u and v. Next modify Partition to proceed to consider nodes in order of v-height above v. Steps 2 and 3 are modified to scan nodes of v-height i in D_v and queue up the appropriate strings at this v-height. After all nodes with v-height less than ∞ have been processed, the remaining nodes are processed by Partition in order of increasing height.

To maintain the v-height of each node above v an array VHT(1:n) is maintained. The v-height of a node is calculated using the v-height of its sons, and is set to zero whenever the count reaches the v-height of v. □

6. An On-Line Algorithm

The above algorithms for the uniform word problem all assume that the given equalities (and inequalities) between expressions are known before any deductions need to be drawn, and that expressions are presented in the form of dags. Thus the algorithms are off-line. These assumptions are realistic in contexts involving code optimization; however, for interactive applications such as theorem proving and symbolic execution, equalities, inequalities and requests to perform deductions may arrive in any order. Algorithms are needed which are on-line and lend themselves to a dynamically changing environment of "known" equalities and inequalities. This is especially true in symbolic execution systems [Ki] where alternating program paths are explored: in one path an equality is assumed true and in another path it is assumed false.

For example, we would like to be able to process the following transaction stream on-line:

example:
1. $f(a) = c$
2. $f(b) = d$
3. $a = b$
4. deduce: $c = d$?
5. $g(a,b) \neq g(b,d)$
6. $h(c) \neq h(d)$
7. deduce: $b \neq d$?

The order in which the equalities are encountered is important: if $a = b$ were encountered before $f(a) = c$ and $f(b) = d$ then the equality $c = d$ would be quite easy to prove.

An algorithm for the above example is on-line if each equality, inequality or deduction request is completely processed before the next is read. Thus before input 5 is read, the algorithm must answer whether or not $c = d$ is deducible from the information in 1, 2 and 3. Similarly before input 7 is read, the algorithm must respond that inequality 6 is inconsistent with the information in 1 to 5.

The basic problem is that of creating and updating a data base for equalities so that a simple decision algorithm can be used to determine if two expressions are indeed equal. We will use the notion of equivalence classes to keep track of all expressions known to be equal. An equivalence class is constructed for each expression which has been encountered while processing equalities. A class is represented by a value number. Moreover, the subexpressions of each expression are represented in terms of their value numbers.

For example, when the equality $f(a) = c$ is processed an equivalence class is created for a (say $\underline{0}$), for $f(a)$ (say $\underline{1}$ whose member is $f(\underline{0})$), and for c (say $\underline{2}$). An Available Computation List (ACL) is maintained with the definitions $\langle a,\underline{0}\rangle\langle f(\underline{0}),\underline{1}\rangle\langle c,\underline{2}\rangle$, which is accessed by hashing on the lefthand element. The equality of $f(a) = c$ is processed by merging the two equivalence classes $\underline{1}$ and $\underline{2}$, and all subsequent references to $f(a)$ or c are by use of the lowest numbered equivalence class which was merged -- i.e., $\underline{1}$ in our example. This would result in an ACL altered to $\langle a,\underline{0}\rangle\langle f(\underline{0}),\underline{1}\rangle\langle c,\underline{1}\rangle$.

The process of adding an equality to the data base consists of:

(1) For each half of the equality determine the equivalence class in which it is contained (create one if it is not contained in any equivalence class). This is done by "parsing" each expression bottom-up, using the ACL to assign value numbers to subexpressions. This is done by routine Parse in figure 5.
(2) Merge the two equivalence classes (by routine Propagate in figure 5.
(3) Update all references to the merged equivalence classes to point to the new equivalence class.
(4) Merge all equivalence classes whose equivalence is a direct consequence of 2 (by virtue of the collapsing transformation $\Rightarrow_c$). This may result in the merge routine Propagate being called recursively.

As a clarification of (4) consider the case when $a = b$, and $f(a)$ and $f(b)$ appear in separate equivalence classes, labelled p and q with $p < q$. Then (2) implies that $f(a)$ and $f(b)$ are to be uniquely represented as $f(n)(n$ is the name of the equivalence class containing a and b), and thus the two classes containing $f(a)$ and $f(b)$ are merged into one class p. All ACL references to q are altered to p.

The process of determining the equivalence class (value number) of an expression may be thought of as a form of simple precedence parsing against the "reductions" available in the ACL.

The algorithm for adding an equality to the data base is given in Figure 5. The data base consists of an indexed table ACL, one entry for each node processed to date. (We will persist in referring to nodes instead of expressions and subexpressions). An entry ACL[j] is of the form $\langle \sigma, m \rangle$ where σ is a string $\psi v_1 v_2 \ldots v_r$ consisting of an operator symbol and son value numbers; m is the value number of the associated node. ACL is accessed by hashing on the strings σ.

All references to a member of an equivalence class are in terms of its value number. Procedure Propagate insures that the value number of any class equals the smallest index of any node in the class.

Steps 1 and 2 correspond to the merging of equivalence classes due to transitivity: all references to the merged value numbers min, max are replaced by the minimum value number. Step 3 enforces the propagation of equality by the collapsing transformation: when two ACL entries $\langle \sigma, i \rangle$ and $\langle \sigma, k \rangle$ with identical strings are found, the classes i and k are merged by recursively calling Propagate. There are no more than $|ACL|$ calls of Propagate for each call to Update.

MODS is a list of ACL indexes of entries $\langle \sigma, m \rangle$ which mention max or min in σ; thus it is a list of father nodes affected by the merge of max and min. MODS is maintained in ascending sort to simplify the search for a string match in Step 3. Step 3 need only be applied to entries on MODS. Note that duplicate ACL entries caused by merging of value numbers are deleted (Since ACL is hashed, we do not actually remove these duplicates, but only mark them as deleted).

The process of determining the equivalence of two expressions is quite simple from a computational standpoint. Specifically, in parsing an expression there are exactly as many probes of the ACL to be made as there are input names and operator symbols in the expressions.

The process of updating the ACL can be speeded up significantly by the maintenance of linked lists NODES[1:n] and FATHERS[1:n]. NODES[i], which links all members of equivalence class i, obviates, in part, the need for Step 2. FATHERS[i], which links all equivalence classes having the son i, obviates the need for Step 3 and the preparatory Steps 1 and 2.

Inequalities can also be handled. This is accomplished by maintaining a table NEQL of pairs of equivalence classes (value numbers) which are known to be unequal. The algorithm for proving equalities needs only a slight modification to be able to cope with inequality queries such as $b \neq d$? in the above example. In such a case, the inequality does not appear explicitly in the data base. Instead, we derive it by contradiction. We assume that $b = d$ and (temporarily) add this relationship to our data base. If $b \neq d$ is true, then a contradiction will occur. This contradiction is detected at the occurrence of an attempted merge of two equivalence classes which are known to be unequal (those of $g(a,b)$ and $g(b,d)$ in the above example).

Each entry in NEQL is a pair of value numbers. Therefore, whenever a merge of two equivalence classes occurs, this table must also be updated. NEQL is updated just before Step 1 of Propagate as follows:

```
for each pair (x,y) in NEQL do replace each
    occurrence of max by min in (x,y);
      if x = y then "contradiction";
end
```

Note that the inequality algorithm modifies the data base ACL. Should a contradiction be detected, we would like to undo the updating that has occurred. This is not a problem if the algorithm operates in a recursive environment where dynamic storage allocation and garbage collection are available (e.g., LISP).

7. Related Results

Paterson and Wegman [PW] consider unification, which can be described informally as follows: given two expressions g and h containing variable symbols, substitute subexpressions for the variables in such a way as to make g and h identical. In the uniform word problem, we start with a given set of equalities A and ask if g and h are equivalent. The unification problem is to start with expressions g and h and ask if there exists a (suitably constrained) set of equalities A subject to which g and h are equivalent. The two problems are quite different. While an $O(n)$ algorithm exists for unification, $O(n \log n)$ is the best comparable result for the uw problem. And while assuming operator commutativity does not affect the uw problem complexity, it can be shown that unification with commutative operators is NP-hard.

```
procedure Update (e,f)
     /* e,f are expressions; e = f is the axiom to
        be added to the data base*/

     ℓ:=Parse(e) /* obtain the value numbers
     r:=Parse(f) /* of e,f from the ACL*/
     Propagate (ℓ, r, null) /* propagate effect of
        new equivalence through the data base*/
end Update;

procedure Propagate (min, max, MODS)

     if min > max then swap min and max
        /* min ≤ max*/
     /* min, max are the value numbers of the
        equivalence classes being merged. MODS is
        a list of ACL indices, in ascending sort,
        of entries which mention max or min*/

     1. for non-deleted j , min+1 ≤ j ≤ |ACL| do
             Let ACL[j] = ⟨σ,m⟩
             if min occurs in σ then insert j
                in MODS
        end

     2. for non-deleted j , max ≤ j ≤ |ACL| do
             /* replace all instances of max by
                min*/
             Let ACL[j] = ⟨σ,m⟩
             if m = max then replace m by min
             if max occurs in σ
                  then
                       Replace all occurrences of
                         max in σ by min
                       Rehash (σ)
                       Insert j in MODS
                       /* rehashing needed since
                          string σ has changed*/
        end

     3. while MODS ≠ null do
             /* step through the list of affected
                nodes looking for matching
                strings and propagate merge*/

             Let h be the smallest index on MODS
             Delete h from MODS
             Let ACL[h] = ⟨σ,i⟩
             for each j on MODS do
                  Let ACL[j] = ⟨τ,k⟩
                  if σ = τ
                       then /*matching string*/
                            Delete j from MODS
                            Mark ACL[j] deleted
                               from ACL
                            /* Classes i,k must
                               be merged*/
                            if i ≠ k then
                            Propagate (i, k, MODS)
             end
        end
end Propagate;
```

Figure 5: Algorithm to Process an Equality

The axioms given in the uniform word problem are equalities between expressions not involving any variables. But a law like the commutative law $x+y=y+x$ is really a scheme for inferring many instances of axioms of the form $e+f=f+e$ where e,f are variable-free expressions.

Let $X=\{x, y, \ldots\}$ be a set of variables. Equalities between expressions over $\Theta \cup S \cup X$ are called axiom schemata. We review the effect on the complexity of the uniform word problem which results from admitting schemata.

As we have seen, introducing only the commutative axiom schema for operators does not materially affect the decision procedure for equality. If A is a set of axioms over $S \cup \{\cdot\}$, then $A \cup \{x\cdot(y\cdot z) = (x\cdot y)\cdot z\}$ generates a finitely presented semigroup. The uniform word problem for semigroups is well known to be undecidable [Tars]. Indeed there exists a fixed set of axioms A for which the word problem is undecidable.

What happens when both associativity and commutativity of operator $\cdot$ are admitted? If A is a finite set of axioms, $A \cup \{x\cdot(y\cdot z) = (x\cdot y)\cdot z,\ x\cdot y = y\cdot x\}$ generates a finitely presented commutative semigroup. The uniform word problem for such commutative semigroups is decidable; however, a recent result shows that this problem is complete in exponential space [CLM]. It follows that for infinitely many A, deciding equality of expressions requires exponential time.

Kozen [Kol] has shown that inferring schemata from a set of variable-free axioms is a hard problem. Knuth and Bendix [KB] and Lankford [L] give algorithms which work to decide word problems for some sets of schemata.

Appendix A: Inequalities and the Uniform Word Problem

Let $A=\{e_{i1}=e_{i2} \mid 1 \le i \le m\}$ and $B=\{f_{j1} \neq f_{j2} \mid 1 \le j \le n\}$ be sets of equalities and inequalities, respectively. We shall also use A and B for the conjunctions $A = \bigwedge_i (e_{i1}=e_{i2})$ and $B = \bigwedge_j (f_{j1} \neq f_{j2})$.

Formulas are built up from equalities using the usual logical connectives $\vee$, $\wedge$, $\neg$ and $\supset$. Formulas of the type $\neg\, g=h$ will be abbreviated $g \neq h$. The meaning function m can readily be extended to formulas. A formula F is satisfied under i provided m(F) is true. A formula is consistent if it is satisfied by some i. A formula is valid if it is satisfied by all i.

Consider the following four problems. Given sets A and B as above, and expressions g and h:

(a) EQUALITY PROBLEM: Determine whether $g=h$ is satisfied for all i satisfying $A \wedge B$. This is the same as determining whether $A \wedge B \supset (g=h)$ is valid.

(b) CONSISTENCY PROBLEM: Determine whether there exists i satisfying $A \wedge B$. This is the same as determining whether $\neg\, (A \wedge B)$ is invalid.

(c) INEQUALITY PROBLEM: Determine whether $g \neq h$ is satisfied for all i satisfying $A \wedge B$. This is the same as determining whether $A \wedge B \supset (g \neq h)$ is valid.

(d) UNIFORM WORD PROBLEM: Determine whether $g=h$ is satisfied for all i satisfying just the equalities A. This is the same as determining whether $A \supset (g=h)$ is valid.

All four problems reduce to instances of the uniform word problem, as may be seen from the following lemmas.

LEMMA A1: Given $A = \Lambda_i (e_{i1} = e_{i2})$, and $B = \Lambda_j (f_{j1} \neq f_{j2})$, and expressions g and h:

(a) $A \wedge B \supset (g = h)$ is valid if and only if $A \supset V_j (f_{j1} = f_{j2}) \vee (g = h)$ is valid.

(b) $\neg (A \wedge B)$ is invalid if and only if $A \supset V_j (f_{j1} = f_{j2})$ is invalid.

(c) $A \wedge B \supset (g \neq h)$ is valid if and only if $A \wedge (g = h) \supset V_j (f_{j1} = f_{j2})$ is valid.

Proof: Part (a) follows from the logical equivalence of $A \wedge B \supset (g = h)$ and $A \supset \neg B \vee (g = h)$. The other parts are similar. □

LEMMA A2: Let A be a conjunction of equalities. $A \supset V_{j=1}^{n} (g_j = h_j)$ is valid if and only if at least one of $A \supset (g_j = h_j)$ is valid, for $1 \leq j \leq n$.

Proof: Assume that $A \supset V_{j=1}^{n} (g_j = h_j)$ is valid, and suppose that all of $A \supset (g_j = h_j)$ are invalid. Then there exist $\underline{i}_1, \ldots, \underline{i}_n$ over value domains $V_1, \ldots, V_n$, such that $\underline{i}_j$ satisfies $A \wedge (g_j \neq h_j)$. Consider $\underline{i}$ over $V_1 \times \ldots \times V_n$, defined by $\underline{m}(e) = \underline{m}_1(e) \times \ldots \times \underline{m}_n(e)$, where $\underline{m}_j(e)$ is the value of e under $\underline{i}_j$. For this $\underline{i}$, the formula $A \wedge (g_1 \neq h_1) \wedge \ldots \wedge (g_n \neq h_n)$ is satisfied. But then $A \supset V_j (g_j = h_j)$ is not valid -- a contradiction.

The other direction of the lemma is immediate. □

The above lemmas show that all the problems (a), (b), (c) reduce to deciding the validity of formulas of the form $A \supset (g = h)$, where A is a conjunction of equalities. Thus a solution to the uniform word problem solves all the problems.

Acknowledgement

We wish to thank Al Aho, John Bruno, Don Johnson, Jeff Ullman, Mike Garey and Dave Johnson for helpful discussions.

References

[A] W. Ackermann, Solvable Cases of the Decision Problem, North-Holland, Amsterdam (1954).

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

[AU] A. V. Aho and J. D. Ullman, Optimization of straight line programs, SIAM J. Computing 1, 1 (March 1972) 1-19.

[B] M. S. Breuer, Generation of optimal code for expressions via factorization, CACM 12, 6 (June 1969) 333-340.

[CLM] E. Cardoza, R. Lipton and A. R. Meyer, Exponential space complete problems for Petri nets and commutative semigroups, Proc. 8th Ann. ACM Symp. on Theory of Computing, Hershey, PA (May 1976) 50-54.

[CS] John Cocke and J. T. Schwartz, Programming Languages and Their Compilers Preliminary Notes, Second Revised Version, Courant Institute of Mathematical Sciences, New York, NY (April 1970).

[Cu] Karel Culik, Combinatorial problems in the theory of complexity of algorithmic nets without cycles for simple computers, Aplikace Matematiky 16 (1971) 188-202.

[DS] P. J. Downey and Ravi Sethi, Assignment commands and array structures, Proc. 17th Ann. Symposium on Foundations of Computer Science, (October 1976) 57-66.

[Go] D. I. Good, R. L. London and W. W. Bledsoe, An interactive program verification system, IEEE Trans. on Software Engineering SE-1 (March 1975) 59-67.

[H] J. E. Hopcroft, An n log n algorithm for minimizing states in a finite automaton, in Z. Kohavi and A. Paz (ed.) Theory of Machines and Computations, Academic Press, New York, NY (1971) 189-196.

[HK] J. E. Hopcroft and R. M. Karp, An algorithm for testing equivalence of finite automata, TR-71-114, Dept. of Computer Science, Cornell Univ. (1971); see description in Aho et. al. (1974) 143-145.

[Ki] J. C. King, Symbolic execution and program testing, C. ACM 19 (July 1976) 385-394.

[Kn] D. E. Knuth, The Art of Computer Programming: Volume 3, Sorting and Searching, Addison Wesley, Reading, MA (1973).

[KB] D. E. Knuth and P. B. Bendix, Simple word problems in universal algebras. In J. Leech, Ed., Computational Problems in Abstract Algebra, Pergamon Press, (1970).

[Ko1] D. Kozen, Complexity of finitely presented algebras, Proc. 9th Ann. ACM Symp. on Theory of Computing, Boulder, Co. (May 1977) 164-177.

[Ko2] D. Kozen, Lower bounds for natural proof systems, Proc. 18th Ann. Symp. on Foundations of Computer Science, (Oct. 1977), 254-266.

[L] D. S. Lankford, Canonical algebraic simplification in computational logic, Department of Mathematics report, Southwestern University, Georgetown, TX (1975).

[NO1] G. Nelson and D. Oppen, Fast decision algorithms based on union and find, Proc. 18th Ann. Symp. on Foundations of Computer Science, (Oct. 1977), 114-119.

[NO2] G. Nelson and D. Oppen, A simplifier for program manipulation, this proceedings.

[PW] M. S. Paterson and M. N. Wegman, Linear unification, Proc. 8th Ann. ACM Symp. on Theory of Computing, Hershey, PA (May 1976) 181-186.

[Sa1] H. Samet, A normal form for compiler testing Proc. Symp. on Artificial Intelligence and Programming Languages, Rochester, NY (August 1977).

[Sa2] H. Samet, Proving the correctness of heuristically optimized code, Communications of the ACM, to appear.

[Se1] Ravi Sethi, Testing for the Church Rosser property, JACM 21, 4 (October 1974) 671-679; errata JACM 22, 3(July 1975) 424.

[Se2] Ravi Sethi, Scheduling graphs on two processors, SIAM J. Computing 5, 1 (March 1976) 73-82.

[Sh] R. E. Shostak, An algorithm for reasoning about equality, Proc. of the Symp. on AI and Programming Languages, SIGPLAN Notices 12 (Aug. 1977), 155-162.

[Tarj1] R. E. Tarjan, Depth first search and linear graph algorithms, SIAM J. Computing 1, 2 (June 1972) 146-160.

[Tarj2] R. E. Tarjan, On the efficiency of a good but not linear set merging algorithm, JACM 22, 2 (April 1975) 215-225.

[Tarj3] R. E. Tarjan, private communication.

[Tars] A. Tarski, A. Mostowski and R. M. Robinson, Undecidable theories, North-Holland Publishing Co., Amsterdam (1953).