PURGING IN AN EQUALITY DATA BASE

Hanan SAMET
Department of Computer Science, University of Maryland, College Park, MD, U.S.A.

Leo MARCUS
Information Sciences Institute, University of Southern California, Marina del Rey, CA, U.S.A.

Received 27 June 1979; revised version received 7 January 1980

Equality, purging, on-line algorithms

1. Introduction

Equality data bases are important in facilitating proofs of equality between terns. They are useful in the implementation of code optimizers [2], program verifiers [7,9], theorem provers [1], and symbolic execution systems [5]. In these applications the typical operations are determining whether or not two terms are known to be equal and updating the data base to include a new equality. Some solutions include [3,4, 8,10]. In this paper we discuss the problem of removing information ('purging') from an equality data base and describe an algorithm for carrying this out. A natural example where purging is useful is symbolic machine execution where register values are changing.

As an example of the type of operations that we wish to handle, suppose we are given the set of equalities following from $a \equiv b$, $g(a) \equiv h(c)$, and $f(b) \equiv h(d)$, and we wish to remove all information about $f(b)$. One possible interpretation will result in eliminating the equation $f(a) \equiv h(d)$, since $f(b) \equiv f(a)$ by $a \equiv b$. Another interpretation will be given later.

* This research was supported in part by the Rome Air Development Center under contract F30602-78-C-0008 and in part by the Defense Advanced Research Projects Agency under contract DAHC15-72-C-0308. The views expressed are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

2. Preliminaries

The language contains function symbols and constant symbols considered as zero-place function symbols, but no variable symbols. Thus the data base contains equivalences between constant terms, but cannot store 'laws', i.e., universal equivalences. Terms and subterms are defined in the usual way. We assume a standard enumeration of all terms $t_0, t_1, ...,$ where for all i, t_i appears after all its proper subterms. Equivalence (between terms) is denoted $\equiv$, identity is denoted $=$.

The equivalences will be coded by a 'congruence function'.

Let **N** denote the set of natural numbers, and let T be a finite set of terms.

Definition 1. A congruence function on T is a function $E : T \to \mathbf{N}$ such that $E(s_i) = E(s_i')$ for all i implies $E(f(s_1, ..., s_n)) = E(f(s_1', ..., s_n'))$ if the two terms on the right-hand side of the implication are in $\mathrm{Dom}(E) = T$. In the following E (or E with sub- or superscripts) will always denote a congruence function.

Definition 2. $s \equiv_E t$ means $E(s) = E(t)$.

Claim 1. $\equiv_E$ is an equivalence relation on Dom(E) satisfying substitutivity, i.e., a congruence relation.

Definition 3. If $s_1, ..., s_n$ are disjoint (occurrences of) subterms of s and $s_1', ..., s_n'$ are any terms, then the simultaneous substitution of s_i' for s_i is well defined and denoted $s(s_i'/s_i)$. Thus we allow that s_i is the same *term* as s_j for some i, j, but the occurrences are disjoint.

Claim 2. If $s_1, ..., s_n$ are disjoint (occurrences of) subterms of s, $E(s_i) = E(s_i')$ and $s' = s(s_i'/s_i)$, then $E(s) = E(s')$.

Definition 4. s is E-trivially equivalent to t if $s = f(r_1, ..., r_n)$, $t = f(r_1', ..., r_n')$ and $E(r_i) = E(r_i')$. Thus, if $E(a) = E(b)$, then f(a, a) is E-trivially equivalent to f(a, b), f(b, a), f(b, b), and f(a, a).

Let cl(S, E, T) = {s ∈ T: s is E-trivially equivalent to an element of S}.

Clearly, E-trivial equivalence is an equivalence relation refining $\equiv_E$, $S \subseteq cl(S, E, T)$, and if s is an atom, then s ∈ S iff s ∈ cl(S, E, T).

If S is a set of terms, SUB(S) is the closure of S under subterm and SUP(S) is the closure of S (in T) under superterms.

If $E : T \to N$ is a congruence function and $T' \supseteq T$ is a finite set closed under subterms, the (deductive, as we shall see in Corollary 1 below) closure of E in T', CL(E, T'), is the function $E' : T' \to N$ defined as follows:
Step 0: if s ∈ T, then $E'(s) = E(s)$.
Step i: let s be the ith term in the standard enumeration for which $E'(s)$ is not yet defined. If $s = f(r_1, ..., r_n)$, then $E'(r_i)$ is already defined. If there are r_i' such that $E'(r_i')$ is already defined, and $E'(f(r_1', ..., r_n'))$ is already defined, and $E'(r_i') = E'(r_i)$ for all i, then define $E'(f(r_1, ..., r_n)) = E'(f(r_1', ..., r_n'))$. Otherwise (including the case where s is an atom) $E'(s) = \max(\{E'(t) : E'(t)$ is already defined$\} \cup Ran(E)) + 1$.

Definition 5. The index of s with respect to S in T, Ind(s, S, T) = if s ∈ S, then 0, else the i such that s is the ith term of T not in S, by the standard enumeration.

Claim 3. If E is a congruence function and $T' \supseteq T$ is a finite set closed under subterms, then $E' = CL(E, T')$ is a congruence function.

Lemma 1. If $E(s) = E(t)$, then there are subterms $s_1^*, ..., s_n^*$ of s and $t_1^*, ..., t_n^*$ of t such that $t = s(t_1^*/s_1^*, ..., t_n^*/s_n^*)$ and $E(s_i^*) = E(t_i^*)$ for all i, but not E-trivially.

Proof. If s and t are not E-trivially equivalent, then we are done. If they are, then $s = f(s_1, ..., s_n)$, $t = f(t_1, ..., t_n)$ and $E(s_i) = E(t_i)$. Now break down these equations until non-trivial equivalences are reached (as they must finally, since equivalence between atoms is non-trivial).

Definition 6. Let E_1, E_2 be two congruence functions defined on the same domain. $E_1 \equiv E_2$ if $E_1(s) = E_1(t) \Leftrightarrow E_2(s) = E_2(t)$.

Of course, $E_1 \equiv E_2$ means that except for naming of equivalence classes, E_1 and E_2 contain the same information.

Definition 7. Let $E : T \to N$, $S \subseteq T$, s, t ∈ T. t is an S-step from s if there are s_1 a subterm of s and t_1 a subterm of t, such that $E(s_1) = E(t_1)$, $s_1, t_1 \in S$, $t = s(t_1/s_1)$. That is, t is obtained from s by substituting a term in S for an occurrence of an E-equivalent subterm of s in S. An S-chain from s to t is a sequence $s_0 = s, s_1, ..., s_n = t$, where each s_{i+1} is an S-step from s_i.

Definition 8. If $S \subseteq Dom(E)$, then $E \mid S$ is the restriction of E to S.

Lemma 2. Let $S \subseteq T = Dom(E)$, $E_1 = CL(E \mid S, T)$. Let s, t ∈ T. $E_1(s) = E_1(t)$ iff

(1) s and t are E_1-trivially equivalent or

(2) there are s', t' ∈ S, $E(s') = E(t')$, such that there are S-chains from s to s' and t to t'.

Proof. ⇐: Assume (2). $E_1(s) = E_1(s')$ and $E_1(t) = E_1(t')$ since S-stepping preserves E_1. Also $E_1(s') = E_1(t')$ since s', t' ∈ S. Thus $E_1(s) = E_1(t)$.

⇒: By induction on the index. Assume $E_1(s) = E_1(t)$, $E_1(t)$ was defined before $E_1(s)$, and the claim is true for all equations containing only terms defined before $E_1(s)$. If s and t are not E_1-trivially equivalent, then the only way for them to be equivalent is for $s = f(r_1, ..., r_n)$ and there is $s^* = f(r_1^*, ..., r_n^*)$ such that $E_1(r_i^*) = E_1(r_i)$, $E_1(s^*) = E_1(t)$, where all the $E_1(r_i^*)$ and $E_1(s^*)$ were defined earlier. Now for each $E_1(r_i^*) = E_1(r_i)$ equation, by Lemma 1, we can find subterms of r_i^* and r_i which are E_1-equivalent but not trivially. For ease in notation assume that already $E_1(r_i^*) = E_1(r_i)$ non-trivially for all i. Now by induction hypothesis for $E_1(s^*) = E_1(t)$ there is a finite S-chain from s^* to some $s' \in S$ such that the rest of the conclusions hold. In addition, by using the induction hypothesis for each $E_1(r_i^*) = E_1(r_i)$, there are finite S-chains from r_i to $r_i' \in S$ and from r_i^* to $r_i^{*\prime} \in S$, such that $E(r_i') = E(r_i^{*\prime})$ (and of course $E_1(r_i') = E_1(r_i^{*\prime})$).

Now we shall exhibit an S-chain from s to s′:

$$\begin{aligned} s = f(r_1, r_2, ..., r_n) &\to \cdots \to f(r_1', r_2, ..., r_n) \\ &\to f(r_1^{*\prime}, r_2, ..., r_n) \to \cdots \to f(r_1^*, r_2, ..., r_n) \\ &\to \cdots \to f(r_1^*, r_2', ..., r_n) \to f(r_1^*, r_2^{*\prime}, ..., r_n) \\ &\to \cdots \to f(r_1^*, ..., r_n^*) = s^* \to \cdots \to s'. \end{aligned}$$

In particular,

Corollary 1. Under the conditions of Lemma 2, $E_1(s) = E_1(t)$ iff this follows by equational logic from $E \mid S$.

3. Purging

Now we come to the central concept of the paper. Given an equality data base, that is, a set of terms with equivalences among them (perhaps coded by a congruence function, as in Section 2), we view these equivalences as true in the 'current' state. A state change can consist of two aspects, positive and negative. The positive aspect says that we now know what the new values of some terms are. The negative aspect says we no longer know what the values of some terms are, thus we must destroy ('purge') all old information dependent on these terms. It is this latter aspect we concentrate on here [1]. There are two different kinds of purging we consider:

(1) 'by name': given a term s, throw out all equivalences containing s (and close the resulting relation back up to a congruence relation). Of course, the trivial equivalences containing s will be restored, even after s is purged;

(2) 'by value': given a term $s = f(s_1, ..., s_n)$, we plead ignorance as to the effect of f on the *values* represented by $s_1, ..., s_n$. Thus we throw out all equivalences containing $f(s_1', ..., s_n')$ for all $s_i' \equiv s_i$. Here again, trivial equivalences containing s will be restored, but there will be fewer of these than in the name-purging case, since more equivalences were thrown out.

[1] Another possibility, brought to our attention by Chee Yap, is to focus on the equivalences themselves, rather than on the terms. Thus, after a state change we may know that certain equivalences are now true or that certain others are now false. We do not deal at all with this kind of state change.

We claim that both name-purging and value-purging are concepts worthy of study, in that both have valid uses in the appropriate contexts.

As an example, assume $a \equiv b$, $g(f(a)) \equiv c$. Then after name-purging a we have only $g(f(b)) \equiv c$ (this was true before, it does not contain a, so it remains true). The same effect results from value-purging a. Name-purging f(a) or f(b) results in no change in the original data base; however, value-purging f(a) or f(b) results in $a \equiv b$ (and so of course, $f(a) \equiv f(b)$ and $g(f(a)) \equiv g(f(b))$, but not $g(f(a)) \equiv c$).

We shall show that name-purging a set of terms is equivalent to value-purging a subset of those terms. Now for the formal definitions.

Definitions 9. If T is closed under subterms and E is a congruence function on T and $S \subseteq T$, then NPURGE(S, E) is the function $E' : T \to \mathbf{N}$ defined by $E' = CL(E \mid (T - SUP(S)), T)$. Thus, we throw out of T all those terms having subterms in S, and close the resulting restricted function.

Definition 10. VPURGE(S, E) = $CL(E \mid T - (SUP(cl(S, E, T))), T)$. Thus, we first close S under E, then under superterms, throw these out of T and close the resulting restricted function under deduction.

We now proceed to show that for every $E : T \to \mathbf{N}$, $S \subseteq T$, there is $S' \subseteq S$ such that NPURGE(S, E) = VPURGE(S′, E).

Theorem 1. For every $E : T \to \mathbf{N}$, $S \subseteq T$, there is $S' \subseteq S$ such that NPURGE(S, E) $\equiv$ VPURGE(S′, E).

Proof. Let $S \subseteq T$, $S' = \{s \in S$: for all $t \in T$ E-trivially equivalent to s there is an element of S which is a subterm of t$\}$.

Let $S_1 = T - SUP(S)$, $S_2 = T - SUP(cl(S', E, T))$, $E_i = CL(E \mid S_i, T)$ for i = 1, 2. Then we must prove $E_1 \equiv E_2$.

First we need the following lemmas:

Lemma 3. $S_1 \subseteq S_2$.

Proof. It is sufficient to prove SUP(cl(S′, E, T)) ⊆ SUP(S). Let s ∈ SUP(cl(S′, E, T)). Then s has a subterm s_1 such that s_1 ∈ cl(S′, E, T). Thus s_1 is trivially E-equivalent to an element of S′. Thus s_1 has a subterm in S. But then so does s, and s ∈ SUP(S).

Lemma 4. (S_2 is not necessarily ⊆ S_1, but) if s ∈ S_2, then there is an S_1-chain from s to an element of S_1.

Proof. If s ∈ S_1, we are through. So assume s ∉ S_1. Thus s has a subterm s′ ∈ S. Let s′ be a minimal such subterm. Let s′ = f(s'_1, ..., s'_n). (Note that s′ cannot be a constant symbol, i.e., n > 0, since otherwise it would follow s′ ∈ S′ which contradicts s ∈ S_2.) By the minimality of s′, s'_i ∈ S_1. Since s ∈ S_2, s′ is not E-trivially equivalent to anything in S′. Thus s′ ∉ S′. Thus there is t = f(t_1, ..., t_n) E-trivially equivalent to s′, t has no subterm in S. Thus t ∈ S_1. Substituting t for s′ in s is equivalent to substituting t_i for s'_i for all i. Each of these is an S_1-step (E(s'_i) = E(t_i) by the above). Thus the substitution s^* = s(t/s′) can be accomplished by an S_1-chain. If s^* ∈ S_1 we are through. If not continue as before; this process terminates since we never have to deal with subterms of previously considered subterms, and those terms always grow; that is, s ∉ S_1, but t ∈ S_1. When the process terminates, the result is in S_1.

Now we return to the proof of the theorem. We shall prove that E_1(s) = E_1(t) iff E_2(s) = E_2(t) for all s, t.

⇒: By Lemma 3, S_2 ⊇ S_1, and so certainly E_1(s) = E_1(t) ⇒ E_2(s) = E_2(t).

⇐: If E_2(s) = E_2(t), then either

(1) s and t are E_2-trivially equivalent, or

(2) there is an S_2-chain from s, t to s′, t′ ∈ S_2, E(s′) = E(t′) by Lemma 2.

The proof is by induction on the structure of s, t. If (1) holds, use the induction hypothesis on the subterms of s, t which make them E_2 trivially equivalent.

In the case of (2), assume the S_2-chain from s goes s → s'_1 → ··· → s′, where s'_1 is obtained from s by exchanging s_1 ∈ S_2 with s_1^* ∈ S_2, and of course E_2(s_1^*) = E_2(s_1), and so E(s_1^*) = E(s_1). By Lemma 4, there are S_1-chains s_1 → ··· → s_2 ∈ S_1, s_1^* → ··· → s_2^* ∈ S_1. It follows that E(s_2) = E(s_2^*), and since both are in S_1, E_1(s_2) = E_1(s_2^*).

Now it is easy to construct an S_1-chain from s to s'_1, as in the proof of Lemma 2. Continuing this process, we can construct an S_1-chain from s, through s′, to an element of S_1, and likewise for t. Thus, by Lemma 2, E_1(s) = E_1(t).

Of course, NPURGING a set of terms is not necessarily equivalent to NPURGING them one at a time, but:

Claim 4. For all terms s and sets of terms S, VPURGE(S ∪ {s}, E) ≡ VPURGE({s}, VPURGE(S, E)).

In the remainder of the paper, we give the algorithm for implementing VPURGE({s}, E).

4. Algorithm

In brief, the data base is represented as a finite set of equivalence classes, that is, a function E from a finite set of terms to equivalence class names, say E : T → N, where T is some finite set of terms and N is the natural numbers. Of course, it is possible to finitely store the equivalence information of infinitely many terms, for example by {f(a) ≡ a}, but we are really interested in only finitely many of them at one time. Then E(s) = E(t) iff s ≡ t. VPURGE({s}, E) is accomplished by assigning a new (common) equivalence class number to every term trivially equivalent to s (thus if s is atomic, no other atoms will be changed). Subsequently, new class numbers are assigned to terms containing subterms for which new class numbers have already been assigned, in such a way that substitutivity is preserved.

Our algorithm for implementing purging relies on a variation of the representation used in [10] for on-line proofs of equalities and inequalities of terms. In that method an equality data base was constructed consisting of pairs of entries which were analogous to productions of an equality grammar, where the equivalence classes corresponded to non-terminal symbols. Determining if two terms are known to be equivalent was reduced to parsing the two terms and examining whether they were in the same equivalence class. In the following we present the data structure, the revised algorithm for adding an equality to the data base, and the necessary steps required to implement purging.

The algorithm, called update, for adding an equality to the data base is given in Fig. 1 using a combination of LISP and ALGOL. Vertical bars are used to clarify the block structure. The data base is a table, ACL, containing one entry for each different term (i.e., a constant symbol or a function symbol and its arguments). Each entry is uniquely numbered and has two fields, STRING and VN, having the interpretation VN ⇒ STRING. STRING is either a constant symbol or a list consisting of a function symbol followed by

```
procedure update(lh,rh);
/*lh=rh is the equivalence to be added to the data base*/
begin
|
| MERGES:=<<parse(lh),parse(rh)>>;
| /*MERGES is a list of lists which is initialized to the two
|    equivalence classes which are to be merged*/
|
| while MERGES ≠ NULL do
|    /*each iteration processes a pair of equivalence classes in
|      MERGES and propagates the equivalence if the elements of
|      the pair are not already in the same equivalence class*/
|    begin
|     |
|     | 1.  m:=MERGES[1,1]; /*insure that the classes to be merged are*/
|     |     n:=MERGES[1,2]; /*not already equivalent*/
|     |     while MERGED[m] ≠ NULL do m:=MERGED[m];
|     |     while MERGED[n] ≠ NULL do n:=MERGED[n];
|     |
|     | if m ≠ n then
|     |   begin
|     |    |
|     |    | 2.  for each j in NODES[n] do ACL[j].VN:=m;
|     |    |     /*update the equivalence class field*/
|     |    |
|     |    | 3.  append NODES[n] to NODES[m];
|     |    |     NODES[n]:=NULL;
|     |    |     MERGED[n]:=m;
|     |    |
|     |    | 4.  while FATHERS[n] ≠ NULL do
|     |    |       /*propagate the equivalence of m and n through all
|     |    |         formulas in which n is a component*/
|     |    |       begin
|     |    |        |
|     |    |        | j:=FATHERS[n,1];
|     |    |        | /*get the first element in the list FATHERS[n]*/
|     |    |
|     |    |        | if not deleted(ACL[j]) then
|     |    |        |   begin
|     |    |        |    |
|     |    |        |    | delete ACL[j] from the hash table;
|     |    |        |    | substitute m for n in ACL[j].STRING;
|     |    |
|     |    |        |    | found:=hash(ACL[j].STRING);
|     |    |        |    | /*hash returns the index into ACL of an undeleted
|     |    |        |    |   entry having the same ACL[j].STRING value; if
|     |    |        |    |   no such entry exists, then NULL is returned*/
|     |    |        |    |
|     |    |        |    | if found = NULL then
|     |    |        |    |   begin
|     |    |        |    |   | enterhashtable(ACL[j]);
|     |    |        |    |   | add ACL[j] to FATHERS[m];
|     |    |        |    |   end
|     |    |        |    |
|     |    |        |    | else
|     |    |        |    |   begin
|     |    |        |    |   | mark ACL[j] as deleted;
|     |    |        |    |   | if ACL[j].VN ≠ ACL[found].VN then
|     |    |        |    |   |   add <ACL[found].VN,ACL[j].VN> to MERGES;
|     |    |        |    |   |   /*there is no need to add ACL[j] to FATHERS[m]
|     |    |        |    |   |     since at least one formula of the form
|     |    |        |    |   |     ACL[j].STRING must already be on that list
|     |    |        |    |   |     by virtue of found being non-NULL*/
|     |    |        |    |   end;
|     |    |        |   end;
|     |    |        | FATHERS[n]:=tail(FATHERS[n]);
|     |    |        | /*delete FATHERS[n,1] from FATHERS[n]*/
|     |    |       end;
|     |   end;
|     | MERGES:=tail(MERGES); /*delete MERGES[1] from MERGES*/
|    end;
end;
```

Fig. 1. Algorithm to add an equality pair.

pointers to the equivalence classes containing its arguments. The indices into the ACL table also serve as the names of the equivalence classes. For each equivalence class, one ACL entry is designated to identify all members of the equivalence class. VN contains the index of the ACL entry corresponding to the equivalence class to which the term represented by the STRING field belongs.

The speed of update is greatly enhanced by the maintenance of two arrays of linked lists NODES[1 : s] and FATHERS[1 : s], an array MERGED[1 : s], and a list MERGES. In particular, they enable the avoidance of lengthy searches in Steps 2, 3 and 4. NODES [i] links all ACL entries that are members of the same equivalence class, i. FATHERS [i] links all ACL entries whose STRING field contains a reference to i. MERGED [i] indicates if equivalence class i has been merged into another equivalence class, and if so, then it contains its value; otherwise, it contains NULL. MERGES is a list of pairs of equivalence classes which are to be merged. With respect to notation, ⟨⟨A, B⟩, ⟨C, D⟩⟩ denotes a list of pairs containing the pairs ⟨A, B⟩ and ⟨C, D⟩.

Step 1 insures that merges that have taken place since the placement of the pair of equivalence classes in MERGES are taken into account. This is also aided by Step 3 which records in MERGED the name of the equivalence class which has been subsumed. Steps 2 and 3 implement set union on equivalence classes, while Step 4 insures that set union will be performed for all equivalences that are direct consequences of Steps 2 and 3. In other words, terms having the same function symbols and equivalent arguments are equivalent and thus their equivalence classes should be merged. Note that Step 4 is only applied to entries in FATHERS [n] since only these entries can possibly generate new equivalences. The key to the algorithm is that the occurrence of a success collision upon rehashing in Step 4 (i.e., the entry was already in the table), indicates that a pair of equivalence classes has been found that should be merged. If the two entries are not already in the same equivalence class, then their equivalence classes are added to the end of MERGES. Once Step 4 is completed, Steps 1–4 are reapplied to any pair of elements of MERGES.

The equality updating algorithm terminates since parsing is a process that is limited by the length of the input string and by the number of productions. Step 1 is a list traversal of a subset of the equivalence classes and the time it takes is bounded by the number of productions. Steps 2 and 3 correspond to a merge of two equivalence classes, and the time they take is bounded by the number of productions. Step 4 makes use of FATHERS[n] to determine whether or not a subsequent merge of two equivalence classes is to occur when the current merge causes two equivalence classes to have an element in common. In the affirmative case, each such pair of equivalence classes is added to MERGES. In order to perform the subsequent merge operations, Steps 1–4 are reapplied. However, when these steps are reapplied, there remains one less equivalence class and thus by the well-ordering principle termination is guaranteed. When MERGES is exhausted, the updating process is finished. Note that if an equivalence class is found to contain a duplicate occurrence of an element after a merge, then, by Step 4, the duplicate occurrence is not reinserted in the hash table [2]. This insures that the equality grammar will always have the property that no two productions have the same right-hand side.

The process of determining the equivalence of two terms is quite simple from a computational standpoint. Specifically, when parsing a term there are exactly as many reductions to be made as there are atoms and function symbols in the term. Thus when hashing is used, the running time of the equivalence determination procedure is directly proportional to the size of the terms whose equivalence is being ascertained (i.e., the number of atoms and operators in the terms).

The running time of the equality updating algorithm depends only on the efficiency of the hash table search mechanism. It has a worse case behavior of d^2, where d is the size of the data base (i.e., the number of terms). This bound is attained when an equivalence class, which is currently being merged, appears as an element in every other entry in the data base – i.e., |FATHERS[i]| = d for equivalence class i. Recall that each merge results in a decrease of at least one in the size of the equality data base. Thus after at most d steps, the algorithm terminates.

[2] Note that we mark deleted ACL entries as deleted rather than physically deleting them. This is only done in the interest of saving time. If the ACL entries were to be physically deleted and their space reused, then NODES and FATHERS entries would have to be purged of references to deleted nodes. This can be done when one has truly run out of space by using garbage collection techniques [6].

```
1. Let p be the index of the ACL entry corresponding to t;
2. Remove p from NODES[ACL[p].VN];
3. Allocate a new equivalence class, say q;
4. Insert p in NODES[q];
5. ACL[p].VN := q;
6. FATHERS[q] := NULL;
```

Fig. 2. Algorithm for purging an equality.

The algorithm for purging the equalities associated with a certain term, say t, is now quite straightforward. Let p be the index of the ACL entry containing term t. All that is required is to remove p from its equivalence class (i.e., delete it from NODES[ACL[p].VN]), allocate a new equivalence class, say q, and insert p in NODES[q]. All FATHERS[i], where i is an element of ACL[p].STRING, remain the same since t remains in ACL[p]. Fig. 2 shows the algorithm in greater detail.

The key to the purging algorithm is that once the equalities associated with a particular term, say t, have been purged, all new terms which will be encountered in which t is a subterm will appear in different equivalence classes than previously. This is because t is now in a different equivalence class and thus the ACL entries in which it appeared earlier as a subterm are no longer accessible.

5. Example

As an example, consider the following three equalities:

$$f(a) \equiv c, \qquad f(b) \equiv d, \qquad a \equiv b.$$

Once the equalities have been processed, the equality

Table 1

Term	Index	ACL STRING	VN
a	1	a	a1
f(a), f(b)	2	f(a1)	a2
c	3	c	a2
b	4	b	a1
d	5	d	a2

Table 2

Term	Index	ACL STRING	VN
a	1	a	a1
f(a), f(b)	2	f(a1)	a3
c	3	c	a2
b	4	b	a1
d	5	d	a2

data base has the form as in Table 1, where ai corresponds to equivalence class i.

If f(a) is to be purged, then we get the equality data base in Table 2.

Note that f(a) is still equal to f(b) and likewise for c ≡ d and a ≡ b; however, the equivalence of f(a) with c and d has been purged as a result of the lack of knowledge about the value of f(a) and likewise for the equivalence of f(b) with c and d. This is not surprising since despite lack of knowledge with respect to the actual value of f applied to a, the fact that a ≡ b means that f(a) is still equal to f(b).

As another example, suppose we are originally given the equality pair f(a) ≡ a. This implies that f(f(a)) ≡ a, f(f(f(a))) ≡ a, etc. If f(f(a)) were to be purged, then the equality of f(a) with a is also affected because f applied to f(a) is the same as f applied to a and thus lack of knowledge about the value of f(f(a)) is equivalent to lack of knowledge about f(a) and thus we no longer have f(a) ≡ a.

Finally, note well the following phenomenon, which was mentioned earlier. Suppose we have the equalities f(a) ≡ g(b) and g(b) ≡ h(c). This implies that f(a) ≡ h(c), a relationship that is still true after g(b) is purged. This is because with no variables in the data base, there is no way to remember laws, axioms, or histories; i.e., there is no way for f(a) to remember that it equals h(c) 'because of' g(b).

Acknowledgment

The suggestion to add purging as an operation to equality data bases was made by Steve Crocker, with whom we had many helpful discussions. We also wish to thank Randy Trigg for finding an error in a previous version of this paper.

References

[1] C. Chang and R.C. Lee, Symbolic Logic and Mechanized Theorem Proving (Academic Press, New York, 1973).

[2] J. Cocke and J.T. Schwartz, Programming languages and their compilers, Courant Institute, New York University (1970).

[3] P.J. Downey, H. Samet and R. Sethi, Off-line and on-line algorithms for deducing equalities, Proc. Fifth Annual ACM Symposium on Principles of Programming Languages (1978) 158–170.

[4] P.J. Downey and R. Sethi, Variations on the common subexpression problem, Technical Report, Bell Laboratories, Murray Hill, NJ (1977).

[5] J.C. King, Symbolic execution and program testing, Comm. ACM (July 1976) 385–394.

[6] D.E. Knuth, The Art of Computer Programming, Vol. 1, Fundamental Algorithms (Addison-Wesley, Reading, MA, 2nd ed., 1973).

[7] R.L. London, The current state of proving programs correct, Proc. ACM 25th Annual Conference (1972) 39–46.

[8] D.C. Oppen, Reasoning about recursively defined data structures, Proc. Fifth Annual ACM Symposium on Principles of Programming Languages (1978) 151–157.

[9] H. Samet, Proving the correctness of heuristically optimized code, Comm. ACM (July 1978) 570–582.

[10] H. Samet, Efficient on-line proofs of equalities and inequalities of formulas, IEEE Trans. Comput. (March 1980).