Efficient On-Line Proofs of Equalities and Inequalities of Formulas

HANAN SAMET

Abstract—**An algorithm is presented for proving equivalence and inequivalence of instances of formulas involving constant terms. It is based on the construction of an equality data base in the form of a grammar. The algorithm differes from other approaches to the problem by being an on-line algorithm. Equality between two formulas can be proved in time proportional to the number of constant and function symbols appearing within them. An algorithm is also given for updating the equality data base. It has a worse case running time which is proportional to the square of the number of different formulas previously encountered.**

Index Terms—**Equality, code optimization, hashing, on-line algorithms, program verification, symbolic execution, theorem proving.**

I. Introduction

There exist practical and efficient methods for proving equality and inequality between instances of formulas involving constant terms. Such methods are crucial in the implementation of code optimizers [4], program verifiers [8], [11], theorem provers [3], and symbolic execution systems [7]. All of these applications require a capability of maintaining a data base to enable the performance of operations such as: determining whether or not two items are known to be equal or unequal, updating the data base to include a new equality or inequality, and ascertaining whether or not a pair of equalities is consistent with the current set of equalities and inequalities.

For example, we would like to be able to prove relationships such as

Example 1: Given: $f(a) = c$, $f(b) = d$, $a = b$
Derive: $c = d$

Example 2: Given: $g(a, b) \neq g(c, d)$, $a = c$
Derive: $b \neq d$

The order in which the equalities are encountered is important in the sense that the algorithms must lend themselves to a dynamically changing environment (i.e., on-line algorithms [2][1]). This is especially true in symbolic execution systems where alternate program paths are explored, i.e., in one path a relationship is assumed to be true, while in another path it is assumed to be false. In Example 1 we see that if $a = b$ were encountered prior to $f(a) = c$, and $f(b) = d$, then the equality of c and d would be quite easy to prove. Our goal is to be able to prove such relationships regardless of the order in which the data base is constructed.

In order to aid our exposition, we define some terms using the following grammar:

$$\begin{aligned}
\langle\text{formula}\rangle &\Rightarrow \langle\text{atomic formula}\rangle \mid \langle\text{compound formula}\rangle \\
\langle\text{atomic formula}\rangle &\Rightarrow \langle\text{constant symbol}\rangle \\
\langle\text{compound formula}\rangle &\Rightarrow \langle\text{function symbol}\rangle(\langle\text{argument list}\rangle) \\
\langle\text{argument list}\rangle &\Rightarrow \langle\text{formula}\rangle \mid \langle\text{argument list}\rangle, \langle\text{formula}\rangle
\end{aligned}$$

where ⟨constant symbol⟩ and ⟨function symbol⟩ are drawn from some suitably defined alphabet. For the purposes of our examples we will let

$$\begin{aligned}
\langle\text{constant symbol}\rangle &\Rightarrow a \mid b \mid c \mid d \\
\langle\text{function symbol}\rangle &\Rightarrow f \mid g \mid h
\end{aligned}$$

Formulas are used to build equalities and inequalities, i.e.,

$$\begin{aligned}
\langle\text{equality}\rangle &\Rightarrow \langle\text{formula}\rangle = \langle\text{formula}\rangle \\
\langle\text{inequality}\rangle &\Rightarrow \langle\text{formula}\rangle \neq \langle\text{formula}\rangle
\end{aligned}$$

II. Solution

The problem we address, also termed the uniform word problem, is known to be decidable [1]. Thus our goal is to demonstrate a solution amenable to an on-line situation. One potential solution is to use equivalence classes such that whenever an equality is added to the data base, a set union is performed on the corresponding classes so that the resulting class reflects all possible members based on this equality. Such an approach is unacceptable because all possible equalities can not be generated [e.g., $f(a) = a$ requires the equivalence class to contain $a, f(a), f(f(a)), \cdots$].

The previous solution can be termed a generative approach to the problem. An alternative solution is one employing reduction methods. An equality grammar, G',

Manuscript received January 23, 1978. This work was sponsored in part by a General Research Board Faculty Award of the University of Maryland. Preliminary results were presented in part at the Principles of Programming Languages Conference in Tucson, Arizona, January 23–25, 1978.

The author is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

[1] Each equality, inequality, or deduction request is fully processed before attempting the next one.

0018-9340/80/0100-0028$00.75

$\langle S_1 \rangle \Rightarrow \langle \text{atom}_1 \rangle$
$\langle S_2 \rangle \Rightarrow \langle \text{atom}_2 \rangle$
$\vdots$
$\langle S_k \rangle \Rightarrow \langle \text{atom}_k \rangle$
$\langle S_m \rangle \Rightarrow (\langle \text{fname}_m \rangle \langle S_{m1} \rangle \langle S_{m2} \rangle \cdots \langle S_{mp} \rangle)$
$\langle S_n \rangle \Rightarrow (\langle \text{fname}_n \rangle \langle S_{n1} \rangle \langle S_{n2} \rangle \cdots \langle S_{nq} \rangle)$
$\vdots$
$\langle S_r \rangle \Rightarrow (\langle \text{fname}_r \rangle \langle S_{r1} \rangle \langle S_{r2} \rangle \cdots \langle S_{rn} \rangle)$

Fig. 1. Sample set of productions for G.

can be constructed where for each equality there is a pair of symmetric productions. For example, for formula1 = formula2, we have productions:

$$\text{formula1} \Rightarrow \text{formula2}$$
$$\text{formula2} \Rightarrow \text{formula1}$$

If $L(G')$ is taken to mean the language generated (recognized) by the grammar G', then the equality problem can be recast as given a pair of sentences of $L(G')$, determine if one sentence may be generated from the other. However, this decision problem is undecidable because G' is a type 0 grammar.

Our solution is based on the use of an equality grammar G with the following properties. Each formula is represented by a production whose left-hand side is a nonterminal symbol representing the formula and whose right-hand side is either a constant symbol, or a list containing a function symbol (i.e., $\langle \text{fname}_i \rangle$) and a sequence of nonterminal symbols corresponding to its arguments. The terminal symbols are the constant symbols (i.e., $\langle \text{atom}_i \rangle$) and the function symbols (i.e., $\langle \text{fname}_i \rangle$). Fig. 1 is an example of a G where S_i and S_{ij} correspond to nonterminal symbols. Note that we use prefix notation (with parentheses to denote grouping) and thus the order in which the arguments appear is important.

As a more concrete example, consider Fig. 2 where the productions corresponding to $f(a)$, c, $f(b)$, and d of Example 1 are given.

The basic problem to which we address ourselves is that of finding an efficient method for creating and updating a data base for equalities so that a simple decision algorithm can be used to determine if formula1 = formula2 given that formula1 and formula2 are in $L(G)$. We make use of an analogy between nonterminal symbols and equivalence classes to keep track of all sentences of $L(G)$ known to be equivalent. In essence, we create a sequence of grammars G_1, G_2, G_3, $\cdots$ such that the set of formulas seen up to point i is exactly $L(G_i)$. The nonterminal symbols of G_i represent the known equivalence classes up to point i. Moreover, for those productions of G_i that represent compound formulas [e.g., $g(a,b)$], the argument lists are written in terms of the nonterminal symbols of G_i which represent them. This has a couple of important ramifications. First, when a and b are in the same equivalence class, then so are $f(a)$ and $f(b)$. For example, with G as

$$A0 \Rightarrow a$$
$$A0 \Rightarrow b$$
$$A1 \Rightarrow f(A0),$$

$A0 \Rightarrow a$
$A1 \Rightarrow (f\ A0)$
$A2 \Rightarrow c$
$A3 \Rightarrow b$
$A4 \Rightarrow (f\ A3)$
$A5 \Rightarrow d$.

Fig. 2. Productions for the formulas in Example 1.

a and b being equivalent is shown by having both represented by $A0$. Furthermore, $A1$ represents both $f(a)$ and $f(b)$. Second, our grammar method is recursive thereby enabling the representation of equalities such as $f(a) = a$, i.e., a pair of productions of the form:

$$A0 \Rightarrow a$$
$$A0 \Rightarrow f(A0).$$

Recall that this was a deficiency of the generative approach discussed earlier.

For example, if $f(a) = c$, then recalling Fig. 2 we see that when this equality is processed, the current grammar G_i is modified to become G_{i+1}. In particular, productions of the form:

$$A0 \Rightarrow a$$
$$A1 \Rightarrow f(A0)$$
$$A2 \Rightarrow c$$

are created for formulas a, $f(a)$, and c, respectively. The equivalence of $f(a)$ and c is noted by making $f(A0)$ and c the right-hand sides of the same nonterminal symbol, say A_i, and replacing all occurrences of $A1$ and $A2$ by A_i. Note that instead of creating a new nonterminal symbol we reuse one of the two nonterminal symbols which are equivalent, e.g., $A1$ in this example. As another example, consider the equality $f(a) = a$. In this case, the existing grammar, G_{i+1}, is modified by making a and $f(A0)$ the right-hand sides of the same nonterminal symbol, say $A0$, and replacing all occurrences of $A1$ by $A0$.

More formally, the process of adding an equality to our data base consists of the following steps.

Step 1: For each half of the equality, determine the appropriate nonterminal symbol. This will result in the creation of productions for formulas that do not appear as right-hand sides of any production.

Step 2: Replace all occurrences of one of the nonterminal symbols corresponding to the two halves of the equality by the other. This is done for both right- and left-hand sides of all productions.

Step 3: Reapply Step 2 for all equalities that are a direct consequence of Step 2.

As a clarification of Step 3 consider the grammar of Fig. 2, say G_i. The formulas $f(a)$ and $f(b)$ appear as right-hand sides of nonterminal symbols $A1$ and $A4$, respectively. The addition of the equality $a = b$ causes the existing grammar G_i to be modified and results in G_{i+1}. In particular, Step 2 implies that $f(a)$ and $f(b)$ are both represented by the same right-hand side, i.e., $f(A0)$. Step 3 causes the reapplication of Step 2 for nonterminal symbols $A1$ and $A4$. This process is

continued until there do not remain any different productions with identical right-hand sides. Our algorithm will be seen to attempt to perform this step as efficiently as possible by a judicious selection of data structures.

Step 1 is analogous to parsing a sentence of $L(G)$. The main problem associated with parsing is that there may be a reduction to be made (in the form of a handle [6]); yet there is no nonterminal symbol to which this handle is to be reduced. In our case the problem is alleviated by the fact that G is always simple precedence [9][2], and thus we always know when a reduction is desired. Therefore, if no reduction is possible, then a new production is added with the handle as its right hand side and a new non-terminal symbol as the left hand side. Thus, to determine if two formulas are known to be equivalent, we simply parse them and see if they parse to the same nonterminal symbol.

Our algorithm, called update, for adding an equality to the data base is given in Fig. 3 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 production. Each entry is uniquely numbered and has two fields, STRING and VN, having the interpretation VN ⇒ STRING. At this point it is useful to refer to the set of right-hand sides of the productions having the same nonterminal symbol as their left-hand side, say S, as the equivalence class of S. This notation will be used in the remainder of the discussion. In the case of a compound formula, STRING is a list consisting of the function symbol followed by pointers to the equivalence classes containing its arguments. We let the indices into the ACL table serve as the names of the equvalence classes. In order for all members of the same equivalence class to be uniquely identified and referenced, for each equivalence class (i.e., nonterminal symbol) we designate one ACL entry to identify all members of that class (hereinafter termed the head of the equivalence class). This is facilitated by the VN field which contains the index of the ACL entry corresponding to the equivalence class to which the contents of the STRING field belongs. The table is accessed by hashing on the value of the STRING field.

The speed of update is greatly enhanced by the maintenance of two arrays of linked lists NODES[1:s] and FATHERS[1:s], and a list MERGES. In particular, they enable the avoidance of lengthy searches in Steps 1, 2, and 3. NODES[i] links all members of equivalence class i where i is the index of the ACL entry corresponding to the head of the equivalence class. FATHERS[i] links all equivalence class members whose STRING field contains a reference to i (i has the same meaning as in NODES[i]). MERGES is a list of pairs of equivalence classes which are to be made equal (i.e., merged). With respect to notation, $\langle\langle A,B\rangle,\langle C,D\rangle\rangle$ denotes a list of pairs containing the pairs $\langle A,B\rangle$ and $\langle C,D\rangle$.

Steps 1 and 2 of update implement set union on equivalence classes, while Step 3 assures that set union will be

```
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
      m := ACL[MERGES[1,1]].VN; /*insure that heads of equivalence*/
      n := ACL[MERGES[1,2]].VN; /*classes are being merged*/
      if m ≠ N then
        begin
          1. for each j in NODES[n] do ACL[j].VN := m;
             /*update the head of the equivalence class. This step
               is also performed for deleted nodes since there may
               exist elements in MERGES which refer to deleted nodes*/
          2. append NODES[n] to NODES[m] and set NODES[n] to NULL;
          3. 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
                     production having the same right hand side; 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 if ACL[j].VN = ACL[found].VN then
                     mark ACL[j] as deleted
                   else 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;
               /*delete FATHERS[n,1] from FATHERS[n]*/
               FATHERS[n] := tail(FATHERS[n]);
             end;
        end;
      MERGES := tail(MERGES); /*delete MERGES[1] from MERGES*/
    end;
end;
```

Fig. 3. Algorithm to add an equality pair.

performed for all equivalences that are direct consequents of Steps 1 and 2. In other words, formulas having the same function symbols and equivalent argument lists are equivalent and thus the heads of their equivalence classes should be merged. Note that Step 3 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 3

[2] The list-like representation in Fig. 1 enables a simple proof of the simple precedence property, i.e., by Step 3 no two different productions have the same right-hand sides and there are no conflicts among the simple precedence relations.

(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[3], then the heads of their respective equivalence classes are added to the end of MERGES. Once Step 3 is completed, Steps 1–3 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. Steps 1 and 2 correspond to a merge of two equivalence classes, and the time they take is bounded by the number of productions. Step 3 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 1–3 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 3, the duplicate occurrence is not reinserted in the hash table.[4] This insures that the equality grammar will always have the property that no two productions have the same right-hand side. However, since ACL consists of pointers to other ACL entries in the form of indices, deleted entries cannot be removed from the table. Instead, such ACL entries are marked as deleted.

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

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 formulas). 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.

III. Example

As an example of the equality updating algorithm, consider:

[3] The equality $f(a) = f(b)$ followed by the equality $a = b$ is an example where two entries are already in the same equivalence class, i.e., when Step 3 is applied after adding $a = b$, $f(a)$ and $f(b)$ are found to already be in the same equivalence class.

[4] However, the duplicate occurrence is not deleted from ACL since it may still be referred to by an element of MERGES that has yet to be processed.

Given: $h(b) = f(a)$
$h(c) = f(b)$
$a = b$
$c = d$
Derive: $h(a) = h(d)$.

In the following derivation, each numbered line represents the result of either parsing a sentence or updating the data base to include a new quality. In the former case, only the modifications to the data base, ACL, are shown; while in the latter case, the entire updated data base is shown. Also, in the former case, the name of the equivalence class containing the sentence being parsed is returned. Note that deleted ACL entries have a value of NIL in their VN field.

Formula		ACL INDEX	ACL STRING	ACL VN	Result
1. $h(b)$	yields	$A0$:	b	$A0$	returns $A1$
		$A1$:	$h(A0)$	$A1$	
2. $f(a)$	yields	$A2$:	a	$A2$	returns $A3$
		$A3$:	$f(A2)$	$A3$	
3. $h(b) = f(a)$	yields	$A0$:	b	$A0$	
		$A1$:	$h(A0)$	$A1$	
		$A2$:	a	$A2$	
		$A3$:	$f(A2)$	$A1$	
4. $h(c)$	yields	$A4$:	c	$A4$	returns $A5$
		$A5$:	$h(A4)$	$A5$	
5. $f(b)$	yields	$A6$:	$f(A0)$	$A6$	returns $A6$
6. $h(c) = f(b)$	yields	$A0$:	b	$A0$	
		$A1$:	$h(A0)$	$A1$	
		$A2$:	a	$A2$	
		$A3$:	$f(A2)$	$A1$	
		$A4$:	c	$A4$	
		$A5$:	$h(A4)$	$A5$	
		$A6$:	$f(A0)$	$A5$	
7. a	yields		no change		returns $A2$
8. b	yields		no change		returns $A0$
9. $a = b$	yields	$A0$:	b	$A0$	
		$A1$:	$h(A0)$	$A1$	
		$A2$:	a	$A0$	
		$A3$:	$f(A0)$	$A1$	
		$A4$:	c	$A4$	
		$A5$:	$h(A4)$	$A5$	
		$A6$:	$f(A0)$	$A5$	
	followed by a merge of $A1$ and $A5$	$A0$:	b	$A0$	
		$A1$:	$h(A0)$	$A1$	
		$A2$:	a	$A0$	
		$A3$:	$f(A0)$	$A1$	
		$A4$:	c	$A4$	
		$A5$:	$h(A4)$	$A1$	
		$A6$:	$f(A0)$	NIL	
10. c	yields		no change		returns $A4$
11. d	yields	$A7$:	d	$A7$	returns $A7$
12. $c = d$	yields	$A0$:	b	$A0$	
		$A1$:	$h(A0)$	$A1$	
		$A2$:	a	$A0$	
		$A3$:	$f(A0)$	$A1$	
		$A4$:	c	$A4$	
		$A5$:	$h(A4)$	$A1$	
		$A6$:	$f(A0)$	NIL	
		$A7$:	d	$A4$	

At this point we wish to determine if $h(a) = h(d)$.
$h(a)$ gets parsed successively as

1. $h(A0)$
2. $A1$

and $h(d)$ gets parsed successively as

1. $h(A4)$
2. $A1$

Therefore, $h(a) = h(d)$.

```
/*update NEQL and check for a contradiction*/
for j := 1 step 1 until maxneq do
  begin
    if NEQL[j].RIGHT = n then NEQL[j].RIGHT := m
    else if NEQL[j].LEFT = n then NEQL[j].LEFT := m;
    if NEQL[j].LEFT = NEQL[j].RIGHT then "contradiction";
  end;
```

Fig. 4. Additional step for testing inequalities.

IV. Inequivalence

Inequalities can also be handled. This is accomplished by maintaining a list of pairs of equivalence classes which are known to be unequal. The algorithm for proving equalities needs only a slight modification to be able to cope with inequalities such as $b \neq d$ in Example 2. In such a case, the inequality does not appear explicitly in the data base. Instead, it is derived by contradiction. Assume that $b = d$, and add this relationship to the data base. If $b \neq d$ is true, then a contradiction will occur. This contradiction is detected at the occurrence of a merge of two equivalence classes which are known to be unequal.

We keep track of equivalence classes which are known to be unequal by use of a table known as NEQL. Each entry in this table is accessed by two fields named LEFT and RIGHT which correspond to indices into ACL for the two inequivalent equivalence classes. Therefore, whenever a merge of two equivalence classes occurs, this table must also be updated. NEQL is updated just before Step 1 of update in Fig. 3 (i.e., the first task after testing $m \neq n$). Fig. 4 shows the updating step. maxneq indicates the number of entries in NEQL.

As an example of the utility of an inequivalence capability, consider Fig. 5—a program fragment in which Example 2 has been embedded. Use of a symbolic execution system in the context of program verification (e.g., [7]) results in an attempt to exercise some or all possible program paths. In case of Fig. 5, one such path may include $g(a, b) \neq g(c, d)$, $a = c$, and $b = d$ being true. However, such a path is impossible by virtue of the inconsistency of $g(a, b) \neq g(c, d)$ with $a = c$ and $b = d$.[5] Thus the capability to handle inequivalences enables a symbolic execution system to avoid exercising impossible program paths.

Note that the inequivalence algorithm is an in-place algorithm, i.e., it overwrites the data base. Should a contradiction be detected, it is desirable 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).

V. Conclusion

An algorithm has been presented for the efficient derivation of proofs for equality and inequality of instances of formulas involving constant terms. The algorithm differed from other approaches such as [5] by proceeding in an on-line manner. This means that the overall time bounds for the algorithm as presented here may be inferior; however, the representation lends itself more readily to an on-line environment, i.e., whenever a new equality is added, new inferences can be made without having to rebuild the entire equality data base. For other related work, see [10].

[5] It is assumed that no computations involving side-effects on a, b, c, and d have taken place between the computation of $g(a, b) \neq g(c, d)$ and the test $b = d$.

```
if g(a, b) ≠ g(c, d) then
  begin
    :
    if a = c then
      begin
        :
        if b = d then f(a) /*this case is impossible*/
        else g(d);
          :
      end
        :
  end
```

Fig. 5. Example of the utility of an inequivalence capability.

Commutativity can also be handled. In such a case, each time a formula corresponding to a commutative binary operation is encountered, then when parsing the formula for the first time, an additinal entry is made into the equality data base with the arguments interchanged. Associativity is a more complicated problem.

Acknowledgment

Special thanks go to P. J. Downey, D. F. Martin, and R. Sethi for helpful discussions.

References

[1] W. Ackermann, *Solvable Cases of the Decision Problem*. Amsterdam: North-Holland, 1954.
[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974, p. 109.
[3] C. Chang and R. C. Lee, *Symbolic Logic and Mechanical Theorem Proving*. New York: Academic Press, 1973.
[4] J. Cocke and J. T. Schwartz, *Programming Languages and their Compilers*. New York: NYU Courant Institute, Apr. 1970.
[5] P. J. Downey and R. Sethi, "Variations on the common subexpression problem," Tech. Rep., Bell Lab., Murray Hill, NJ, 1977.
[6] D. Gries, *Compiler Construction for Digital Computers*. New York: Wiley, 1971.
[7] J. C. King, "Symbolic execution and program testing," *Communications Assoc. Comput. Mach.*, pp. 385–394, July 1976.
[8] R. L. London, "The current state of proving programs correct," in *Proc. ACM 25th Annu. Conf.*, 1972, pp. 39–46.
[9] D. F. Martin, "Boolean matrix methods for the detection of simple precedence grammars," *Commun. Assoc. Comput. Mach.*, pp. 685–687, Oct. 1968.
[10] D. C. Oppen, "Reasoning about recursively defined data structures," in *Proc. Fifth Annu. ACM Symp. Principles of Programming Languages*, 1978, pp. 151–157.
[11] H. Samet, "Proving the correctness of heuristically optimized code," *Commun. Assoc. Comput. Mach.*, pp. 570–582, July 1978.

Hanan Samet (S'70–M'75) received the B.S. degree in engineering from the University of California, Los Angeles, the M.S. degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.

Since 1975 he has been an Assistant Professor of Computer Science at the University of Maryland, College Park. His research interests are data structures, programming languages, code optimization, data base management systems, and artificial intelligence.

Dr. Samet is a member of the Association for Computing Machinery, SIGPLAN, Phi Beta Kappa, and Tau Beta Pi.