5

Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic

R. S. Boyer and J S. Moore
Institute for Computing Science and Computer Applications,
University of Texas at Austin, USA

Abstract

We discuss the problem of incorporating into a heuristic theorem prover a decision procedure for a fragment of the logic. An obvious goal when incorporating such a procedure is to reduce the search space explored by the heuristic component of the system, as would be achieved by eliminating from the system's data base some explicitly stated axioms. For example, if a decision procedure for linear inequalities is added, one would hope to eliminate the explicit consideration of the transitivity axioms. However, the decision procedure must then be used in all the ways the eliminated axioms might have been. The difficulty of achieving this degree of integration is more dependent upon the complexity of the heuristic component than upon that of the decision procedure. The view of the decision procedure as a 'black box' is frequently destroyed by the need to pass large amounts of search strategic information back and forth between the two components. Finally, the efficiency of the decision procedure may be virtually irrelevant; the efficiency of the final system may depend most heavily on how easy it is to communicate between the two components. This paper is a case study of how we integrated a linear arithmetic procedure into a heuristic theorem prover. By *linear arithmetic* here we mean the decidable subset of number theory dealing with universally quantified formulas composed of the logical connectives, the identity relation, the Peano 'less than' relation, the Peano addition and subtraction functions, Peano constants, and variables taking on natural values. We describe our system as it originally stood, and then describe chronologically the evolution of our linear arithmetic procedure and its interface to the heuristic theorem prover. We also provide a detailed description of our final linear arithmetic procedure and the use we make of it. This description graphically illustrates the difference between a stand-alone decision procedure and one that is of use to a more powerful theorem prover.

1. INTRODUCTION

Decision procedures, alone or in co-operation with other decision procedures, are fast and predictable but often too limited to be of general use. On the other hand, today's heuristic theorem provers are capable of producing proofs of fairly deep theorems, but are generally so slow and unpredictable that few users have the patience and knowledge to use them effectively. It is generally agreed that when practical theorem provers are finally available they will contain both heuristic components and many decision procedures.

This paper is a case study of how we integrated into a heuristic theorem prover a linear arithmetic procedure for the natural numbers based on a decision procedure for the rationals. By *linear arithmetic* here we mean the decidable subset of number theory dealing with universally quantified formulas composed of the logical connectives, the identity relation, the Peano 'less than' relation, the Peano addition and subtraction functions, Peano constants, and variables taking on natural values. We built in linear arithmetic primarily to eliminate from the heuristic theorem prover's search space the huge number of often irrelevant deductions arising from such theorems as the transitivity of the 'less than' relation.

This paper can be divided up into three distinct phases. The first, represented by Sections 2–4, argues that it is necessary to combine decision procedures and heuristic theorem provers. During the first phase we also give some necessary background material on our heuristic theorem prover and what we mean by 'linear arithmetic procedures'. The second phase, Sections 5–7, describes chronologically our attempts to incorporate linear arithmetic into our theorem prover. In this phase we cite examples from program verification applications that show the inadequacy of our early integration strategies and that illustrate and motivate our final scheme. The third phase of the paper, Section 8, gives a precise and detailed description of the current linear arithmetic procedure and how it is used by the rest of the theorem prover. The final scheme is so elaborate that reading it in isolation would prompt many readers to ask such questions as 'why is it necessary to know which literals contributed to the deduction?' or 'why didn't the authors use this simpler scheme?'. Despite the tedium of this description we regard Section 8 as the high point of the paper because it makes clear the distinction between a stand-alone decision procedure and one that is useful to a larger system. The last two sections of the paper give some statistics supporting our contention that the efficiency of the stand-alone decision procedure is often irrelevant and a summary of our conclusions.

We believe this report will be useful to those designing decision procedures intended for eventual integration into larger systems. We

identify many requirements for such procedures that are not obvious when the procedures are considered in isolation.

For example, much work on linear arithmetic procedures (e.g. that of Nelson and Oppen, 1979, and Shostak, 1979) focuses on universally quantified formulas either with no function symbols (other than sum and difference) or with only uninterpreted function symbols. But interpreted function symbols play a key role in many theorem-proving applications. In particular, they are crucial in what is perhaps the most active application of mechanical theorem provers today: the verification of properties of computer programs. The mathematical specification of new programs frequently involves 'new' mathematical functions (e.g. 'the number of non-0 elements among the first N elements of A'). Furthermore, these functions very frequently have important numeric relations to one another (e.g. if $N<M$ then the number of non-0 elements among the first N elements of A is less than or equal to the number among the first M elements). Unless provision is made for one's arithmetic procedure to take into consideration the numeric properties of interpreted symbols, the heuristic theorem prover must deal explicitly with such explosive theorems as the transitivity of 'less than' and the primary advantage in having an arithmetic decision procedure is lost.

The work reported here deals with interpreted function symbols: our linear arithmetic procedure contains heuristics for instantiating and using axioms or lemmas about arbitrary function symbols. For example, by appealing to the lemma that the minimum element, $\text{MIN}(A)$, of a sequence is less than or equal to the maximum element, $\text{MAX}(A)$, our linear arithmetic procedure proves:

$$L \leq \text{MIN}(A) \wedge 0 < K \rightarrow L < \text{MAX}(A) + K.$$

The use of universally quantified axioms or theorems in the linear procedure is very similar to the admission of universally quantified hypotheses in the formulas being proved. Thus, our work is similar in spirit to the work of Bledsoe and Hines (1980) in which arbitrary quantification is permitted. However, we make no completeness claims about our heuristics.

2. BACKGROUND

Our theorem prover deals with a quantifier-free, first-order logic. In addition to modus ponens, instantiation, and substitution of equals for equals, the logic provides for the axiomatic introduction of new 'types' of inductively constructed objects (e.g. the natural numbers, sequences, graphs), the user definition of new mathematical functions (e.g. prime, permutation, path), and proof by induction on well-founded relations. The logic is described precisely in Chap. III of Boyer and Moore (1979).

Our theorem prover as it stood before we incorporated any linear arithmetic is described in Chaps V–XV of Boyer and Moore (1979). The theorem prover consists of an *ad hoc* collection of heuristic proof techniques. The two most important ones are simplification and the invention of 'appropriate' induction arguments. The system also contains heuristics for eliminating 'undesirable' expressions, the use of equality, generalization, and the elimination of irrelevance.

Because our linear arithmetic procedure interacts with our 'simplifier' and term 'rewriter', it is necessary to explain these procedures in more detail.

To prove a formula our system first applies the simplifier to it. The *simplifier* is a procedure that takes a formula as input and returns a set of supposedly simpler formulas as output. Under the assumption that the input formula is false, it is equivalent to the conjunction of the output formulas. Since we are trying to prove the formula, it is permitted to assume its negation. If the simplifier returns the empty set of formulas we have succeeded in proving the input formula. If the simplifier returns a singleton set containing the input formula, it has failed to reduce the problem and we try some other proof technique, e.g. induction. Otherwise we try to prove, recursively, each output formula.

A formula is represented as a *clause* consisting of an implicitly disjoined set of literals. Literals are in fact *terms* of our logic. The term p, when used as a literal, can be thought of as the formula $p \neq F$, where F is a distinguished constant in our logic.

The simplifier works by successively 'rewriting' the literals of the goal clause while assuming the complements of the remaining literals. The object is to rewrite at least one literal to T (or any other non-F value).

The *rewriter* is a procedure that takes a term, a substitution and a 'context' and returns a term. Among other things the context specifies a set of assumptions. The term returned by the rewriter is equal (in a certain sense determined by the context) to the result of instantiating the input term with the input substitution, under the assumptions in the context. The context contains a variety of other information which we will explain when necessary.

The rewriter applies conditional rewrite rules derived from axioms, recursive definitions, and previously proved theorems tagged 'rewrite rule' by the user. Roughly speaking, a previously proved lemma of the form:

$$h_1 \wedge h_2 \wedge \cdots \wedge h_n \rightarrow lhs = rhs$$

causes the rewriter to replace all instances of *lhs* by the corresponding instance of *rhs* provided each of the instantiated h_i can be established. To establish the hypotheses the rewriter attempts recursively to rewrite them to non-F—a form of backwards chaining. The system contains fairly

sophisticated search strategic heuristics for controlling the expansion of definitions, stopping unproductive backwards chaining, using permutative rewrites, etc.

Among the theorems proved by the theorem prover described in Boyer and Moore (1979) are: the totality, soundness, and completeness of a decision procedure for propositional calculus; the correctness of a 'toy' expression compiler; the correctness of the fastest known string-searching algorithm; and the existence and uniqueness of prime factorizations. Other proofs discovered before the linear algorithm was implemented include the correctness of a recursive descent parser (Gloess, 1980), the correctness of an arithmetic simplifier now in routine use in the system (Boyer and Moore, 1981a), and the correctness of several FORTRAN programs (Boyer and Moore, 1981b).

3. LINEAR ARITHMETIC

The theorem prover described above can easily prove by mathematical induction such simple theorems as:

$$X \leq Y \wedge Y \leq Z \rightarrow X \leq Z \tag{1}$$

$$X - 1 \leq X \tag{2}$$

$$0 \leq Y \rightarrow X \leq X + Y, \tag{3}$$

but because of search strategic heuristics the system cannot always employ such lemmas intelligently after they have been proved. For example, while the system would easily recognize:

$$A \leq B \wedge B \leq C \rightarrow A \leq C$$

as an instance of the transitivity of $\leq$, (1), it would not so easily prove:

$$A - 1 \leq A + B$$

where B is non-negative. But $A - 1 \leq A + B$ can be derived from (2) and (3) using (1). The theorem prover described in Boyer and Moore (1979) tries to derive $A - 1 \leq A + B$ from (1)–(3) by rewriting. In particular, it observes that the result follows from (1) if one instantiates X with $A - 1$ and Z with $A + B$, chooses an appropriate instantiation of the intermediate variable Y, and backward chains to relieve the two hypotheses. If the instance chosen for Y is A, then the two hypotheses are immediate from (2) and (3) and the assumption that B is non-negative. But our system is unable to guess that an appropriate choice for Y is A. After failing to find a proof by simplification, the system in Boyer and Moore (1979) proves the inequality by induction on A.

Now suppose that in a subsequent proof the rewrite routine is faced with the task of relieving the hypothesis $A - 1 \leq A + B$. Since it cannot

derive this inequality from (1)–(3) by rewriting alone, and since we do not try induction to relieve hypotheses, the inequality hypothesis can be established only if $A-1 \leq A+B$ (or some mild variation of it) is available explicitly as a previously proved rewrite rule. There are two undesirable aspects to this situation. First, the search space for rewrites about $\leq$ gets very large because it contains many derived facts involving transitivity and addition. Second, the user is obliged to recognize when the system is failing to find a proof because of its lack of knowledge of such composite 'linear' facts and to state such *ad hoc* lemmas explicitly.

But the integers are probably the most important objects in the mathematics of computer programming. Facts about the integers must be second nature to any practical theorem prover for program analysis. Therefore, after having convinced ourselves of the power of our underlying heuristics, we decided to build-in a linear arithmetic procedure.

The naturals, $\mathcal{N}$, or Peano numbers are the most primitive inductively constructed domain in our theory. Many other domains are constructed on top of the naturals (e.g. the 'atomic symbols' or literal atoms, the integers, the rationals). Thus, we decided to build-in a procedure for deciding some linear inequalities over $\mathcal{N}$, i.e. universally quantified formulas involving the equality ($=$) and Peano 'less than' ($<$) relations, the natural constants $(0, 1, 2, \ldots)$, and the Peano addition ($\oplus$) and subtraction ($\dot{-}$) functions. The syntax of our logic is actually the prefix syntax of LISP. We adopt infix here for the purposes of exposition. For readers familiar with Boyer and Moore (1979) or our theorem prover, $x \in \mathcal{N}$ here denotes (NUMBERP x), $x < y$ denotes the term (LESSP $x\ y$), $x \leqslant y$ denotes (NOT (LESSP $y\ x$)), $x \oplus y$ denotes (PLUS $x\ y$), and $x \dot{-} y$ denotes (DIFFERENCE $x\ y$).

While our logic provides many different types of objects it does not have a typed syntax. Thus, $T \oplus 3$ is a well-formed term. Our definitions of $<$, $\leqslant$, $\oplus$, and $\dot{-}$ 'coerce' non-natural arguments to 0. Thus, $T \oplus 3 = -1 \oplus 3 = 3$ and $T < 4$ is true but $-3 < 0$ is false. In addition, the Peano subtraction function returns 0 if the minuend is smaller than the subtrahend, i.e. $x \dot{-} y = 0$ if $x < y$, so $5 \dot{-} 8 = 0$.

In this paper we use the more familiar signs, $<$, $+$, $-$, and $*$ to denote less than, sum, difference, and multiplication, respectively, over the integers or rationals (according to context).

Linear integer arithmetic, and thus linear Peano arithmetic, is decidable. However, integer decision procedures (e.g. Cooper, 1972) are quite complicated compared to the many well-known decision procedures for linear inequalities over the rationals (King, 1969; Hodes, 1971; Bledsoe, 1975; Shostak, 1977, 1978). Therefore, following the tradition in program verification, we adopted a rational-based procedure, exploiting the observation that if a conjunction of inequalities is unsatisfiable over

the rationals it is unsatisfiable over the integers. Such a procedure is sound but incomplete. For example, $2*X = 3$ is satisfiable over the rationals but not over the integers. Thus, a rational procedure would not prove $2*N \neq 3$, where N is an integer. Of course, it is hoped such theorems do not arise frequently in program verification (Nelson and Oppen, 1979). Should they, we might prove them by more powerful methods (e.g. induction).

For efficiency reasons, the rational method we eventually adopted was based on that described in the literature by Hodes (1971). The algorithm is just a formalization of the high school idea of 'cross-multiplying and adding' equalities to eliminate variables.

In its simplest form the algorithm is used to detect unsatisfiability in a conjunction of linear inequalities over the rationals. The first step is to convert each inequality into a 'normalized polynomial inequality' (or simply 'polynomial') by collecting like terms, cancelling when possible, and making all coefficients integers. Thus, the expression:

$$2 + X - (Y + Z) \leq (A - X) - 1$$

is 'linearized' to

$$3 - 1 * Z - 1 * Y + 2 * X - 1 * A \leq 0.$$

Then, working one's way down through some ordering on the multiplicands, one eliminates one multiplicand at a time from the set of polynomial inequalities by cross-multiplying coefficients and adding inequations so as to cancel out the selected multiplicand in all possible ways. Eventually one obtains a set of ground inequalities whose validity may be determined by evaluation and which is satisfiable over the rationals iff the initial set is.

To apply such a procedure to problems over the integers it is convenient to adopt $\leq$ as the main connective and to transform $X < Y$ into $X + 1 \leq Y$. By making explicit the information that distinct integers are separated by at least 1, fewer valid integer inequalities 'fall between the cracks' in the rationals. An equality, such as $X = Y$, is handled as though it were the conjunction $X \leq Y \wedge Y \leq X$; a negative equality, such as $X \neq Y$, is handled as though it were the disjunction $X < Y \vee Y < X$ (i.e. the main conjunction must be split into two cases).

To apply the procedure to the naturals as they are axiomatized in our logic, we must take precautions to ensure that all quantities involved in arithmetic expressions are of type $\mathcal{N}$ and that the minuend of each $\dot{-}$ expression is no smaller than the subtrahend. For example, we can linearize $X \neq Y$ to $1 - Y + X \leq 0 \vee 1 + Y - X \leq 0$ only if we know both $X \in \mathcal{N}$ and $Y \in \mathcal{N}$. Failure to consider the non-numeric case would permit the linear arithmetic procedure to prove the non-theorem $X \leqslant Y \wedge Y \leqslant$

$X \rightarrow X = Y$.† Similarly, it is permitted to linearize $X \preccurlyeq Y \dot{-} Z$ to $Z - Y + X \leq 0$ only when $Z \preccurlyeq Y$. We call such additional conditions as $X \in \mathcal{N}$ and $Z \preccurlyeq Y$ 'linearization hypotheses.' Roughly speaking, we assume the linearization hypotheses necessary to obtain the normalized polynomials for the main conjecture. If the linear arithmetic procedure is able to prove the main conjecture under those hypotheses we set ourselves the task of proving the main conjecture under the negation of each linearization hypothesis.

Readers troubled by our desire to handle the naturals instead of the integers or by the lack of typing in our language should not be discouraged from reading on. These aspects of the problem only contribute in minor ways to the difficulty of using linear procedures.

Readers troubled by our selection of such a simple and old-fashioned decision procedure are invited to reflect upon the fact that an instantaneous oracle for deciding linear arithmetic problems like those above would increase the speed of our theorem prover on typical program verification problems by less than 3%. Furthermore, as a cursory reading of Section 8 reveals, our final linear arithmetic procedure is an implementation of the above procedure in much the same sense that the software for the Space Shuttle is an implementation of Newton's laws.

4. ADEQUACY OF LINEAR ARITHMETIC PROCEDURES

Before we describe how we have combined the above decision procedure with our more powerful heuristic techniques we address the question 'is linear arithmetic alone sufficient?'. Of course, one must ask 'sufficient for what?'. Since our major concern is the mechanization of the mathematics underlying computer program analysis, we focus our attention on proofs of program correctness. In this context linear arithmetic procedures, and particularly those that are decision procedures only on the rationals, are far from adequate. In this section we discuss three simple program verification exercises that involve either valid integer inequalities that are invalid over the rationals or that are non-linear.

A problem of the first type arose in the first program we tried to verify after adding a linear procedure. The specification of the program was to implement a simple table look-up scheme. The implementation of the table was an array of positive even length, D, with keys and their associated values stored in alternate locations. The access program searched the array linearly, pushing the current index, I, up from 1 in increments of 2, stopping when $I > D$. One of the verification conditions established that when the winning key is found at I and the associated value is fetched from $I + 1$ no array-bound violation occurs. That is, if the current index, I, is a positive odd integer and $I \leq D$, then $I + 1 \leq D$. By

† A counterexample to the conjecture is obtained by letting X be T and Y be F.

letting D be $2 * L$ and I be $2 * K + 1$, where L and K are arbitrary positive integers, we can cast the problem as a linear arithmetic problem:

$$0 < K \wedge 0 < L \wedge 2 * K + 1 \leq 2 * L \rightarrow 2 * K + 2 \leq 2 * L.$$

This is a theorem over the naturals. However, it is not a theorem over the rationals and so does not yield to a rational-based linear arithmetic procedure. Our modified system proved it by induction.

More frequently we see arithmetic problems that do not fall into the linear domain at all. The classic verification example, Euclid's gcd algorithm, illustrates this. Consider the *verification condition* (*v.c.*) that states that if $X < Y$ then the largest number that divides X and Y is also the largest number that divides X and $Y - X$. Some attempts to verify Euclid's algorithm assume this v.c. We wish to prove it. The key step in the proof is that if $X < Y$ and Z divides X then Z divides Y iff Z divides $Y - X$. The definition of 'I divides J' is that $(J \bmod I) = 0$, where mod is defined recursively. This problem falls outside linear arithmetic.

A much more mundane example arises in the attempt to implement a two-dimensional array access module on top of a linear storage scheme. The element at position I,J in the two-dimensional array is mapped to location $I + D * J$ in the linear array. One of the v.c.s establishes that every pair of distinct points in the two-dimensional array maps to a pair of distinct points in the one-dimensional array. Roughly speaking one wishes to establish that $I_1 + D * J_1 = I_2 + D * J_2$ iff $I_1 = I_2$ and $J_1 = J_2$. Because of the multiplication by D, this problem falls outside of linear arithmetic. One might assume this obvious fact. However, attempts to prove it—by induction (on J_1 and J_2 simultaneously) or by appeals to inductively proved facts about mod and quotient—reveal that as stated it is not a theorem. One must hypothesize that I_1 and I_2 are legal indices in the two-dimensional array, i.e. that $I_1 < D$ and $I_2 < D$.

Our point is not to say that linear arithmetic is useless. We have invested several years in building it into our system and have seen it help out in the verification of very many programs. Our point is that it is not unusual to see programs—mundane, everyday programs—that require the proof of arithmetic theorems beyond those of linear arithmetic. If a verification system cannot establish such results then one is forced to assume, rather than prove, many verification conditions.

5. SIMPLE INTEGRATION STRATEGIES

In this and the next section we sketch the evolution of our linear arithmetic procedure and its use by the theorem prover described in Boyer and Moore (1979). We then illustrate the procedure in use. Our intention in these sections is to motivate some of the elaborate bells and whistles described in Section 8.

Our goal was simple: build in enough information about the naturals so that is no longer necessary for the rewriter to consider explicitly those rewrite rules expressing the truths of linear arithmetic. For example, if some old proof required an explicit appeal to the transitivity of $<$ and some inspired 'guess' instantiating the intermediate variable, then after building in linear arithmetic we should be able to find that proof without an explicit appeal to transitivity or a heuristic guess. Ideally, achieving this goal should speed up the theorem prover because certain facts are built-in and because the search space of lemmas is reduced by the deletion of many derived truths of linear arithmetic. Furthermore, achieving the goal frees the user from having to bring linear facts to the system's attention by proving them as rewrite rules.

Our first attempt at incorporating a linear arithmetic procedure was to add it as a 'black-box' applied to every formula produced by the simplifier. The procedure took as its input a clause to prove. If the conjunction of the negations of the literals in the clause is unsatisfiable, the clause is valid. The linear procedure extracted the inequalities in the clause, negated them, introduced linearization hypotheses, formed the set of normalized polynomials and tested the set for unsatisfiability as described. If the set was not found unsatisfiable, the theorem prover tried the next proof technique in its repertoire. If the set was found unsatisfiable, the linear procedure produced as its output a set of clauses, each obtained by adding the negation of a linearization hypothesis to the input clause. The theorem prover then recursively set out to prove each of those 'pathological' cases.

When we applied the modified theorem prover to the 404 definitions and theorems in Appendix A of Boyer and Moore (1979) the linear arithmetic procedure contributed to almost no proof except those of rewrite rules expressing linear facts. That is, when the theorem prover was working on interesting theorems, the simplifier did not produce many conjectures that yielded to linear arithmetic.

Here is an example. One of the uninteresting rewrite rules proved by the linear procedure was named GT.SUB1 and stated that $X \dot{-} 1 \preccurlyeq X \oplus Y$. The only reason this rewrite rule was in the list was because it was needed in the proof of an interesting verification condition later in the list, named FSTRPOS.VC7. The linear procedure in the modified theorem prover established GT.SUB1 immediately but the linear procedure did not participate in the proof of FSTRPOS.VC7. In fact, the proof of FSTRPOS.VC7 still required the explicit use of GT.SUB1 as a lemma and if that lemma was absent then the proof attempt failed—even though the lemma was built-in.

The problem was that while GT.SUB1 was built into the modified theorem prover it was not built-in in the right place. The proof of FSTRPOS.VC7 used GT.SUB1 to relieve a hypothesis of another lemma, not to

prove a simplified part of the main theorem. In our experience linear arithmetic reasoning is most often required during term rewriting and is not terribly useful if its only role is to establish simplified v.c.s.

How can we move the linear arithmetic procedure into the rewriter? Recall that the rewriter operates in a context of assumptions. Suppose we wish to establish an inequality—say a hypothesis of a rewrite rule we wish to apply. We may do so as follows: negate the inequality, conjoin it to the inequalities among our assumptions, linearize the inequalities to obtain a set of polynomials, and then apply the cross-multiply and add procedure to detect unsatisfiability. If the set is found unsatisfiable, the inequality is valid under our assumptions. (Since assumptions of the form $x \neq y$ generate disjunctions of polynomials we will for the time being simply discard any such assumptions.)

For efficiency, we implement this test incrementally. We store the assumptions in a pre-processed internal form in which all polynomials have been maximally 'propagated' in the sense that every admissible cross-multiply and add has been performed. A cross-multiply and add is admissible under our propagation rules only if it eliminates the 'heaviest' multiplicand in both inequalities. For example, because $F(G(X))$ has greater weight than $G(X)$, a cross-multiply and add involving the polynomial

$$8 + F(G(X)) - G(X) \leq 0$$

is permissible only if it eliminates $F(G(X))$. Thus, if no inequality in the incremental data base has $F(G(X))$ occurring negatively as the heaviest multiplicand we do not propagate the above polynomial even if there are other polynomials about $G(X)$. Such propagation will occur as soon as $F(G(X))$ is eliminated and $G(X)$ is exposed as the heaviest multiplicand. This reduces the amount of work the procedure does if irrelevant polynomials are present.

The 'heavier' relation is a total ordering on terms. We say t_1 is *heavier* than t_2 iff either the number of variables in t_1 is greater than that in t_2, or the number of variables in the two are equal but the 'size' of t_1 is greater than that of t_2, or the number of variables in and the sizes of the two are equal and t_1 comes later than t_2 in the lexicographic ordering of terms. By *size* we mean the number of open parentheses in the unabbreviated presentation of the term.

When we introduce a new polynomial into a data base and perform all admissible cross-multiplies and adds, we say we have 'pushed' the polynomial into the data base. The result is a new data base representing the conjunction of the old assumptions and the new inequality.

If pushing a polynomial destructively modifies the initial data base one needs a 'pop' or 'undo' operation; otherwise the attempt to establish a hypothesis by assuming its negation would permanently alter our

assumptions. If pushing an inequality does not destructively modify the data base, the initial data base may be recovered by the usual variable binding mechanisms. This aspect of the problem detracts from the efficiency and simplicity of linear algorithms that rely upon destructively modified cyclic structures (e.g. Shostak, 1978) since the 'pop' algorithm is usually messy. The simple procedure we chose allows the data base to be an ordered alist (associating polynomials with the heaviest multiplicand in them) and permits the implementation of a non-destructive push operation that constructs the new data base from the old using little new structure.

Using such a scheme we programmed the rewriter to use the linear arithmetic procedure when trying to establish inequalities. We found that while the new system was an improvement over the earlier one, our goal—of eliminating the need for explicit lemmas expressing linear facts—was far from achieved.

The problem arises from the presence of interpreted functions. Here is a simple, artificially constructed example. Suppose one needs to prove:

$$L \leqslant \text{MIN}(A) \wedge 0 < K \rightarrow L < \text{MAX}(A) \oplus K \tag{4}$$

where MIN(A) and MAX(A) are defined as the minimum and maximum elements of A. This theorem is not a consequence of linear arithmetic; in particular, since MIN and MAX are treated as uninterpreted function symbols (4) is treated as though it were:

$$L \leqslant \text{MIN} \wedge 0 < K \rightarrow L < \text{MAX} \oplus K.$$

However, if one adds to (4) the additional hypothesis that

$$\text{MIN}(A) \leqslant \text{MAX}(A) \tag{5}$$

the resulting linear arithmetic problem is equivalent to

$$L \leqslant \text{MIN} \wedge 0 < K \wedge \text{MIN} \leqslant \text{MAX} \rightarrow L < \text{MAX} \oplus K,$$

which is a theorem.

To use a linear arithmetic procedure to prove formulas like (4) it is necessary to identify 'interesting' additional hypotheses like (5) to connect multiplicands in the linearization of the goal.

Many readers may object that we should not be trying to use linear arithmetic to prove formulas like (4). But what is the alternative? If we do not use our built-in linear arithmetic procedure we are forced to derive (4) from (5) and explicit linear facts such as the transitivity of $\leqslant$. But if linear arithmetic procedures are to be useful to larger systems they should free the larger system from having to consider the truths of linear arithmetic (such as transitivity). If we do not extend our handling of linear arithmetic to take into account lemmas about 'non-linear' function symbols then the only way we will prove many arithmetic facts is to

ignore the work on linear procedures altogether and return to the heuristic instantiation and chaining methods rejected earlier.

How shall we take into account facts about defined functions? We decided that if after a polynomial has been pushed into the data base no contradiction was found we would look at the multiplicands in the data base and try to link them via additional inequalities obtained by instantiating previously proved lemmas. We call this 'augmenting' the data base. For example, if we have previously proved that:

$$\text{MIN}(S) \leqslant \text{MAX}(S)$$

and construct a linear data base containing the multiplicand MIN(A) or MAX(A) we might push the polynomial obtained from MIN(A) $\leqslant$ MAX(A).

Of course, as Herbrand knew, the problem of which instances of which lemmas to consider is the heart of the theorem-proving problem. We therefore implemented heuristics to control the instantiation of previously proved inequalities and their addition to the polynomial data base. For example, a lemma such as $X < F(X)$ is a 'pump' that may cause one to push, successively, $N < F(N)$, $F(N) < F(F(N))$, etc. Just as with backwards chaining, one has to decide when to stop trying to add new multiplicands to the data base. Our heuristic is to use the same criteria we use to limit backwards chaining, namely, add no multiplicand that is 'worse than' every multiplicand in the data base. For example, we might go around the above loop five times if $F^5(N)$ was initially a multiplicand in the data base.

The problem is further complicated by the need to consider inequality lemmas with hypotheses. For example, let MEMB(X, S) be the predicate that X occurs in the sequence, S, LEN(S) be the length of S, and DEL(X, S) be the result of deleting all occurrences of X from S. Then the following lemma links the theory of lists to arithmetic.

$$\text{MEMB}(X, S) \rightarrow \text{LEN}(\text{DEL}(X, S)) < \text{LEN}(S). \tag{6}$$

Suppose we are asked to prove

$$\text{MEMB}(Z, A) \wedge W \oplus \text{LEN}(A) \leqslant K \rightarrow W \oplus \text{LEN}(\text{DEL}(Z, A)) < K \oplus V.$$

The theorem is a consequence of linear arithmetic if we first add the additional information that:

$$\text{LEN}(\text{DEL}(Z, A)) < \text{LEN}(A).$$

To obtain this inequality we must first instantiate (6), replacing X by Z and S by A [so as to obtain a new inequality about the multiplicand LEN(DEL(Z, A))], and then relieve the hypothesis MEMB(Z, A). Note that to relieve the hypothesis we may have to engage in non-arithmetic reasoning. Therefore, we relieve the hypotheses of 'linear rules' like (6)

by the same methods we relieve the hypotheses of conditional rewrite rules: we recursively rewrite them under our current assumption.

As a consequence, the rewrite mechanism and the linear arithmetic procedure are mutually recursive. The rewrite mechanism calls the linear arithmetic procedure to establish certain inequalities and the linear arithmetic procedure calls the rewrite routine to establish the hypotheses of lemmas providing additional information about the multiplicands in the problem.

6. FURTHER REFINEMENTS

Thus far it has not been crucial to this discussion that we adopted the simple propagation procedure based on Hodes's algorithm. Indeed, at this stage in our actual experimentation we had coded several different linear arithmetic decision procedures and used them as 'black boxes'. However, the attempt to implement the use of 'linear rules' required opening up the black box. In addition, other problems, not yet discussed in detail, required significant modifications to the procedure itself.

One obvious problem is that the heuristic component of the theorem prover must be able to determine what the multiplicands in the current data base are. Either the linear arithmetic procedure should construct the set of multiplicands and make that available outside, or the heuristic component should know the structure of the internal data base. We chose the latter because it was most efficient. However, this choice blurs the line between the heuristic component and the linear procedure.

A second problem arises from the restrictions on the order in which inequalities are processed by the propagation procedure. Consider our procedure. It eliminates the heaviest multiplicands first. Thus, it is a waste of time for the heuristic component to obtain an inequality about $G(X)$ in response to a polynomial such as $8 + F(G(X)) - G(X) \leq 0$.

A related problem is the organization of previously proved inequalities so that the system can rapidly determine relevant facts about the key multiplicands in the data base. Suppose we want to pre-process and store a lemma whose conclusion is an inequality. We store such a lemma so that it may be accessed according to the function symbols of the terms that might, when the lemma is instantiated, become the heaviest multiplicands in the linearized form of the concluding inequality. For example, if a lemma concludes with $F(X) \oplus G(Y) \preccurlyeq H(G(Y)) \oplus X$, we store it under the function symbols F and H. We further require that each such 'key multiplicand' contain enough of the variables in the lemma so that if the key multiplicand is instantiated and the hypotheses are relieved (possibly requiring the instantiation of additional variables) the concluding inequality is fully instantiated.

Note that the notion that the linear procedure is a black box has been

destroyed. Once a particular linear procedure has been selected by the implementor, an extremely large amount of work must be done to interface to it efficiently. In our case, the time taken to program the linear procedure was insignificant (one man-day) compared to the time taken to interface to it (several man-months, not counting the several man-months devoted to the empirical evaluation of each successive implementation). It is certainly not possible to substitute one linear procedure for another. But the worst is yet to come. Much to our dismay we were eventually forced to modify both the linearization subroutine and the propagation subroutine to complete the integration. Thus, the notion that we could choose a linear procedure 'off the shelf' is also destroyed.

It has been found useful by those who write verification systems for the theorem prover to report which lemmas were used in a proof. Such information is necessary if the verification system is to permit the user to redefine or re-axiomatize concepts without having to rederive the proofs of logically independent results. How can the heuristic theorem prover determine whether a given linear rule was used? A 'shotgun' approach can be used. That is, when linear arithmetic participates in a proof it can report that it used every linear rule from which a polynomial was generated and pushed. But the shotgun approach tends to make proofs depend upon many irrelevant lemmas. The approach we finally took was to modify the linearization and propagation subroutines so that every polynomial in the data base carries with it a record of the linear rules from which it was derived. This information is propagated in the obvious way as new inequalities are formed from old ones. When a contradiction is found it is possible to announce exactly which linear rules were used.

More seriously, the search for 'interesting' lemmas and the work involved in relieving their hypotheses make it more expensive to set up the data base for a clause initially. Our first approach was as follows. To rewrite a literal in a clause we pushed into an initially empty data base the polynomials derived from the negations of the remaining inequalities in the clause initially. The data base was then closed under the operation of pushing polynomials derived from heuristically chosen instances of linear rules after relieving their hypotheses. An arbitrary amount of work might be done in setting up the data base for the rewriting of a single literal. Furthermore, the work done to set up the data base for one literal is often very similar to that done for the adjacent literal. Thus, if there is a set of 'expensive' literals in the clause the work they trigger is duplicated each time an 'inexpensive' literal is set up. We found this prohibitively expensive and abandoned the idea of setting up a different data base for each literal.

While trying to prove p it is permitted to assume the negation of p. Thus, we set up just one data base containing the negations of all the

inequalities in the clause and closed under linear rules. That data base was used during the rewriting of each literal.

Of course, while working on the literal p one must be very careful to avoid using the assumption that p is false to rewrite p to F (This phenomenon happens often to students learning proof by contradiction. They assume the negation of what they wish to prove, engage in a long sequence of steps, and then announce that the 'theorem is false'.) 'Accidentally' replacing a literal by F is sound but risky: the rewritten literal is dropped from the disjunction being proved and one is forced to prove a stronger goal which may in fact be invalid and hence unprovable. When this occurs we say the simplifier has 'bitten its own tail'. If the literal being rewritten is assumed false in the context of the rewriter then special precautions must be taken.

Our first attempt to keep the simplifier from biting its own tail was to prevent any attempt to push into the data base the literal we are trying to simplify. This method failed to be effective because the inequality being pushed might be different but linearly equivalent to the one being simplified.

The presence of conditional rewrite rules also complicates the situation. For example, suppose the system knows the lemma:

$$X < Y \rightarrow (X \oplus 1 < Y) \leftrightarrow (X \oplus 1 \neq Y). \tag{7}$$

This permits $(X \oplus 1 < Y)$ to be rewritten to $(X \oplus 1 \neq Y)$ under the condition that $X < Y$. Suppose the current literal is $\neg(A \oplus 1 < B)$. We assume its complement, $A \oplus 1 < B$, and begin simplifying $\neg(A \oplus 1 < B)$. The rewriter observes that it can use (7) to simplify $(A \oplus 1 < B)$ to $(A \oplus 1 \neq B)$ if it can establish $A < B$. By appealing to linear arithmetic it derives $A < B$ from $A \oplus 1 < B$. Therefore, it simplifies $\neg(A \oplus 1 < B)$ to $A \oplus 1 = B$, biting its own tail. We abandoned the hope that we could easily avoid tail biting when we saw such examples. (Less pathological examples can be constructed if one considers lemmas about user-defined function symbols.)

The solution we finally adopted required the further elaboration of the linear algorithm itself. We programmed the linearization subroutine to attach to each polynomial the set of literals from which it was derived. In most cases this is a singleton set containing the inequality literal itself, but in some cases (as when the linearization depends upon another literal to relieve linearization hypotheses) it contains multiple literals. We programmed the propagation subroutine to merge these sets as new polynomials are formed. Thus, we know which literals are involved in the derivation of each polynomial in the data base. Finally, we programmed the propagation subroutine to avoid using any polynomial whose derivation involves the literal we are currently trying to rewrite or any literal previously rewritten to F. (The latter restriction prevents another form of

tail biting. Suppose p and p' are two equivalent but non-identical inequality literals. Consider simplifying the clause $\{p\ p'\}$. The polynomial data base contains two equivalent polynomials one descending from $\neg p$ and the other from $\neg p'$. While rewriting p, we use the polynomial descending from $\neg p'$ to derive $p = F$. If we permit ourselves, while rewriting p', to use the old polynomial descending from $\neg p$ we will bite our tail.)

Another problem we faced is dealing with the linearization hypotheses. These hypotheses are generated when inequalities are put into polynomial form and must be relieved (i.e. either by proving them from other hypotheses or by splitting on them and proving the entire conjecture assuming each of them false).

For example, if the linear arithmetic procedure is applied to:

$$W < I \dot{-} K \wedge J < K \rightarrow W \oplus J < I$$

then, under the hypothesis $K \leqslant I$ the formula is found valid. But an additional case, obtained by assuming that $I < K$ and $I \dot{-} K = 0$, must be proved:

$$I < K \wedge W < 0 \wedge J < K \rightarrow W \oplus J < I.$$

Our first attempt to handle this problem took the shotgun approach again. That is, if the linear arithmetic procedure participated in the simplification of a clause we added to the simplified clause(s) the linearization hypotheses for each literal in the input clause. Furthermore, we produced as new goals additional versions of the input clause in which each of these added hypotheses is negated. Once again, the advantage to the shotgun approach is that while we had to modify the subroutine for putting literals into normal form we did not have to modify the propagation subroutine itself.

But the difficulty with the shotgun approach was that it caused many irrelevant splits. For example, suppose the theorem to be proved is:

$$X < Y \wedge J \leqslant I \dot{-} K \rightarrow X < Y \oplus 1.$$

The contradiction found is actually derived from the first hypothesis and the conclusion. The second hypothesis is irrelevant. But the $I \dot{-} K$ expression in it gives rise to the additional case:

$$X \leqslant Y \wedge I < K \wedge J \leqslant 0 \rightarrow X \leqslant Y \oplus 1.$$

This case will be proved exactly as before, but it need not have arisen in the first place and, in general, reproducing the proof may be quite expensive because of the need for lemmas. Furthermore, the number of irrelevant splits grows exponentially with the number of irrelevant difference or predecessor expressions in the clause. Unfortunately, irrelevant hypotheses are common in mechanically generated formulas.

For example, in our system's first proof of the termination of the Takeuchi function (Moore, 1979) the proof of one lemma involved 412 cases, many of which were irrelevant.

One solution to this problems is that adopted for the tail biting problem. If the linear procedure keeps track of which literals are involved in the derivation of each polynomial, it is possible to report which literals are involved in the eventual contradiction found. Then one can split on the hypotheses necessary to obtain the polynomial for those literals.

However, one can do better. Recall that an assumption such as $I \dot{-} J = K$ gives rise to two inequalities, $I \dot{-} J \preccurlyeq K$ and $K \preccurlyeq I \dot{-} J$. If one is asked to assume $I \dot{-} J \preccurlyeq K$ it is permitted to assume $-K - J + I \leq 0$ even if $I \preccurlyeq J$. That is, the polynomial for the first inequality can be obtained without any additional hypothesis. However, if one is asked to assume $K \preccurlyeq I \dot{-} J$, the polynomial, $K + J - I \leq 0$, may be obtained only under the additional hypothesis that $J \preccurlyeq I$. It is possible that only one of the two polynomials will participate in a contradiction. Thus, in our implementation of the linearization subroutine we attach to each polynomial its linearization hypotheses, we propagate that information in the obvious way, and split on the hypotheses necessary for those polynomials used in the eventual contradiction found. This eliminates many irrelevant case splits when dealing with the naturals. For example, in the proof of the above-mentioned lemma about Takeuchi's function, 311 of the 412 cases were eliminated.

There is one remaining aspect of our scheme. If is often the case that the linear arithmetic procedure derives two polynomials $-y + x \leq 0$ and $y - x \leq 0$. That is, under the hypothesis that x and y are both naturals, they are equal. While the knowledge of this equality is available to the linear arithmetic procedure it is not known to the rewriter. Therefore, after we have set up the polynomial data base for a clause but before we begin rewriting the literals of the clause, we search the data base for 'mated' pairs of polynomials as above and under certain circumstances add equality hypotheses to the clause.

This concludes the casual description of how we integrated a linear arithmetic procedure into our heuristic theorem prover. The objective of these two sections has been to substantiate our assertion that integrating such a procedure into a larger system is quite difficult and frequently requires discarding the notion that the procedure is a black box. In addition, we have attempted to motivate the rather elaborate data structures and procedures described in Section 8.

7. TWO EXAMPLES

In this section we present two examples illustrating the co-operation between our heuristic theorem prover and its linear arithmetic procedure.

The first example comes from our system's proof of the correctness of the Boyer–Moore fast string-searching algorithm (Boyer and Moore, 1977). The algorithm searches for the first occurrence of a given pattern in a given text. Both the pattern and text are strings of characters over a finite alphabet. The algorithm uses an array that associates with each character in the alphabet the distance between the last occurrence of that character in the pattern and the end of the pattern. For those characters in the alphabet that do not occur in the pattern the array contains the length of the pattern. A previously verified subroutine initializes this array. In particular, after the call of this subroutine the Cth element of the array is known to be DELTA1(PAT, LP, C), where DELTA1 is defined recursively to be the distance specified above, PAT is the input pattern and LP is the length of PAT. DELTA1 is another example of an interpreted function symbol needed to specify a program. An interesting inequality involving DELTA1 is DELTA1.LESSEQP.PATLEN, which states that DELTA1(PAT, LP, C) $\preccurlyeq LP$. This result can be proved by induction on LP.

Repeatedly during the execution of the string-searching algorithm the current index into the text, I, is incremented by the Cth element of the above array. One must prove that this addition does not cause an arithmetic overflow. The input assertion of the program assures us that the sum of the length of PAT, LP, and that of the text, LT, is less than or equal to the maximum representable positive integer, MAXINT. Furthermore, at the time of the increment we know that I is less than or equal to LT. We must prove:

$$LP \oplus LT \preccurlyeq \text{MAXINT} \wedge I \preccurlyeq LT$$

$$\rightarrow$$

$$I \oplus \text{DELTA1}(PAT, LP, C) \preccurlyeq \text{MAXINT}.$$

This is not a consequence of linear arithmetic alone. However, after pushing the above inequalities into the data base, DELTA1(PAT, LP, C) is a heaviest multiplicand in an inequality in the data base. By appealing to DELTA1.LESSEQP.PATLEN we obtain the additional information that DELTA1(PAT, LP, C) $\preccurlyeq LP$, from which the above conjecture follows by linear arithmetic.

A second example comes from our program's proof that a certain tree-normalization algorithm terminates. The proof involves showing that the measure MS of the nested ordered pair $\langle\langle a, b\rangle, c\rangle$ is strictly greater than MS of $\langle a, \langle b, c\rangle\rangle$, where MS is defined by the user as follows:

MS(atm) = 1, if atm is not a pair, and

$$\text{MS}(\langle x, y\rangle) = \text{MS}(x) \times \text{MS}(x) \oplus \text{MS}(y)$$

where $\times$ is the Peano multiplication function.

Note that by induction one can prove $0 < \text{MS}(X)$.
Consider our goal:

$$\text{MS}(\langle a, \langle b, c\rangle\rangle) < \text{MS}(\langle\langle a, b\rangle, c\rangle). \quad (8)$$

After simplifying by expanding the definition of MS several times and applying such previously proved arithmetic rewrite rules as the associativity and commutativity of addition and the distributivity of multiplication over addition the goal becomes:

$$\text{MS}(c) \oplus \text{MS}(a)^2 \oplus \text{MS}(b)^2 < \text{MS}(c) \oplus \text{MS}(b)^2 \oplus 2 \times \text{MS}(a)^2 \times \text{MS}(b) \oplus \text{MS}(a)^4, \quad (9)$$

where $\text{MS}(x)^2$ is an abbreviation for $\text{MS}(x) \times \text{MS}(x)$ and $\text{MS}(x)^{n+1}$, for $n > 1$, is an abbreviation for $\text{MS}(x) \times \text{MS}(x)^n$.

Upon trying to simplify the above inequality we push into an empty data base the polynomial obtained from its negation:

$$\text{MS}(a)^4 + 2 * \text{MS}(a)^2 \times \text{MS}(b) - \text{MS}(a)^2 \leq 0$$

from which we hope to derive a contradiction. No linear contradiction is found. Therefore we note that $\text{MS}(a)^4$ is the heaviest multiplicand and search for linear rules about $\times$. We find the linear rule

$$0 < I \rightarrow J \leqslant I \times J$$

and instantiate it by replacing I with $\text{MS}(a)$ and J with $\text{MS}(a)^3$ to produce:

$$0 < \text{MS}(a) \rightarrow \text{MS}(a)^3 \leqslant \text{MS}(a)^4.$$

The hypothesis is rewritten to T by appealing to $0 < \text{MS}(X)$. We then heuristically decide whether we wish to push the polynomial produced from the concluding inequality. Even though it contains $\text{MS}(a)^3$, which is a new multiplicand, we decide it is no worse than the existing $\text{MS}(a)^4$ [indeed, it is a subterm of $\text{MS}(a)^4$]. By pushing the concluding polynomial into the data base and cancelling it against the negated goal we obtain:

$$2 * \text{MS}(a)^2 \times \text{MS}(b) + \text{MS}(a)^3 - \text{MS}(a)^2 \leq 0.$$

No contradiction is found so we again look for linear rules about the heaviest multiplicands. $\text{MS}(a)^2 \times \text{MS}(b)$ is the heaviest multiplicand in the new polynomial [since it is the same size as $\text{MS}(a)^3$ but b is lexicographically larger than a]. We appeal to the same lemma about $\times$, relieve the hypothesis in exactly the same way as before, once again approve the new conclusion as being no worse than existing polynomials and push:

$$-\text{MS}(a)^2 \times \text{MS}(b) + \text{MS}(a) \times \text{MS}(b) \leq 0.$$

Propagation produces the new polynomial:

$$\text{MS}(a)^3 + 2 * \text{MS}(a) \times \text{MS}(b) - \text{MS}(a)^2 \leq 0.$$

Again no linear contradiction is found. This time the heaviest term is $\text{MS}(a)^3$. Appealing again to our lemma about multiplication we obtain and push the polynomial:

$$-\text{MS}(a)^3 + \text{MS}(a)^2 \leq 0.$$

Cancelling again produces:

$$2 * \text{MS}(a) \times \text{MS}(b) \leq 0.$$

No contradiction is found. This time the largest multiplicand is $\text{MS}(a) \times \text{MS}(b)$. Appealing again to our $\times$ lemma we obtain and push:

$$-\text{MS}(a) \times \text{MS}(b) + \text{MS}(b) \leq 0$$

which produces

$$2 * \text{MS}(b) \leq 0.$$

Again no linear contradiction is found. But this time $\text{MS}(b)$ is the heaviest multiplicand. We search for lemmas about MS and obtain:

$$0 < \text{MS}(b)$$

which, when linearized is

$$1 - \text{MS}(b) \leq 0.$$

Pushing this polynomial produces $2 \leq 0$, which is a contradiction. Thus the goal has been proved.

It takes our system a total of 22.3 seconds to prove the goal, equation (8). We can break the proof down into two phases: producing from (8) the fourth-degree polynomial (9), and appealing to linear arithmetic reasoning to prove (9). The first phase, which consists of expanding the definition of MS and applying previous proved rewrite rules, takes a total of 16.5 seconds. However, five of those seconds are consumed by attempts to prove the theorem by linear arithmetic before the normalized polynomial (9) is produced. The second phase—in which the linear arithmetic interface performs the iterated sequence of pushes and lemma instantiations leading to the final contradiction—consumes 5.8 seconds. The times measured are DEC KL-10 c.p.u. seconds consumed while running compiled INTERLISP (not counting garbage collection times and the time taken to output the proof). During the four year period this research was conducted we converted our system from INTERLISP to the MACLISP family. The MACLISP version of the system runs about twice as fast as the INTERLISP version. However, all experimental statistics in this paper are based on the INTERLISP version.

8. THE CURRENT IMPLEMENTATION

In this section we describe precisely the current (if not the final) version of the linear arithmetic procedure.

8.1. More background on the rewriter

Since the linear arithmetic procedure is mutually recursive with the new rewriter, the description of the two are intertwined. We here explain in greater detail aspects of the new rewriter that are mentioned as we describe the linear arithmetic procedure.

The rewriter takes a term, a substitution, and a context and returns a term, a set of linearization hypotheses, and the list of all rewrite and linear rules used to derive the result. The context specifies, among other things, some assumptions and the sense of equality to be maintained by the rewriter. The primary specification satisfied by the rewriter is that under the assumptions in the context plus the returned linearization hypotheses, the output term is equal (in the specified sense) to the instantiation of the input term with the input substitution. Two senses of equality are supported: propositional equivalence and identity. Two terms, p and q, are *propositionally equivalent* iff either $p = F$ and $q = F$ or $p \neq F$ and $q \neq F$. Thus, 3 and T are propositionally equivalent. When rewriting the literals of a clause, the hypotheses of lemmas, and the tests of IFs it is sufficient to maintain propositional equivalence only. At all other times we maintain identity.

Fundamental to the rewriter and to linearization is the notion of 'type sets' and 'type alists'.

A *type alist* is an association list pairing terms with 'type sets'. If r is a 'shell recognizer' then we denote by $\boldsymbol{r}$ the set of all x such that $(r\,x) = T$; we call $\boldsymbol{r}$ a *type*. For example, TRUEP is the set $\{T\}$, FALSEP is the set $\{F\}$, NUMBERP is the set of all natural numbers, and LISTP is the set of all ordered pairs constructed with CONS. In addition, we define one additional type containing all the non-shell objects. A *type set* is a set of types. If the term t is associated with the type set $\{\boldsymbol{r}_1 \cdots \boldsymbol{r}_n\}$ on the type alist then we are assuming that t is an element of one of the $\boldsymbol{r}_i$'s.

Type alists are used in the rewriter to record the assumptions governing the term being rewritten. This mechanism is discussed at length in Chap. V of Boyer and Moore (1979). When the simplifier applies the rewriter to a literal it supplies as part of the context a type alist encoding the assumptions that all the other literals of the clause are false.

In addition to a type alist, the context contains a polynomial data base, set up by the simplifier by pushing the negations of all the inequalities of the clause. The dependence on literals of each polynomial in this data base is carefully tracked so that the rewriter can ignore polynomials descending from the current literal and literals previously rewritten to F. Because of the ubiquity of type sets the rewriter does not record which type set assumptions were used by a rewrite. This makes it difficult to track the dependencies of polynomials derived from linear rules when

setting up the initial data base and we will describe several kludges to mitigate these difficulties.

The polynomial data base is used by the rewriter when it encounters an inequality: if the negation of the inequality, when pushed into the data base, produces a contradictory polynomial, we rewrite the inequality to T. But we must note the hypotheses governing the contradictory polynomial and report them with our final answer if the reduction of this inequality to T is part of the derivation of that answer. It may not be: the inequality just established might be the first of two hypotheses of a rewrite rule. If we fail to establish the second one the work done on the first is irrelevant.

To keep track of the hypotheses generated by a rewrite, we use a push down stack called the *hyps stack* consisting of *frames* each of which contains a set of hypotheses. It is assumed that when the rewriter is called a frame has been pushed onto the hyps stack. The rewriter is implemented so that it adds to the top frame of the stack all of the hypotheses assumed during the derivation of the answer delivered. For example, suppose we are rewriting some term t and try to apply a rewrite rule with hypotheses. Then we push an empty frame on the hyps stack and rewrite the hypotheses of the rule, accumulating the linearization hypotheses in the new frame. If all the hypotheses of the rule are relieved and our heuristics permit us to apply the rule, we pop the new hyps frame off the stack and union its contents into the frame below (which is accumulating the linearization hypotheses actually used to rewrite t). If, on the other hand, some hypothesis of the rule is not relieved, we pop the new frame off the stack and throw its contents away.

A similar stack, called the *lemma stack* is used to keep track of the axiom, definition, and lemma names used by a given rewrite.

The context in which a rewrite takes place thus contains:

TA: a type alist encoding the assumption that each literal of the current clause is false (except the literal we are in the process of rewriting) and, during recursive calls to the rewriter from within the rewriter, the assumptions of the truth or falsity (as appropriate) of the tests of IF's governing the occurrence of the term being rewritten.

DB: a polynomial data base encoding the assumption that every arithmetic literal in the clause is false.

LITS-TO-BE-IGNORED-BY-LINEAR: a list containing the current literal and all previous literals of the goal clause rewritten to F. Polynomials in DB that descend from any literal on this list are ignored by the propagation routine.

LITS-THAT-MAY-BE-ASSUMED-FALSE: the clause being simplified, during the initial construction of the linear arithmetic data base.

HEURISTIC-TA: a type alist used for heuristic purposes when setting up the initial polynomial data base.

OBJECTIVE: a flag that tells the rewriter whether it should try to rewrite the input term to *T*, to *F*, or to anything it can. If the rewriter is 'trying' to get to *T* it does not attempt to apply rewrite rules that would replace the term by *F*. The flag is used to direct the rewriter's efforts when trying to establish the hypotheses of rewrite and linear rules.

ID/IFF: a flag specifying whether identity or propositional equivalence is to be maintained.

hyps stack: a stack of frames containing linearization hypotheses. When the rewriter returns, the assumed hypotheses will have been unioned into the top-most frame.

lemma stack: a stack of frames containing lemma names (and literals from LITS-THAT-MAY-BE-ASSUMED-FALSE used to relieve hypotheses). When the rewriter returns, the lemmas and literals used will have been unioned into the top-most frame.

history: a record of the ancestry of the current clause and what proof techniques were involved in producing each clause in the ancestry.

Other aspects to the context, not relevant to the current discussion, include such search strategic information as the stack of lemmas through which we are currently backwards chaining and a flag indicating whether the term being rewritten is textually within the clause or is part of a lemma or definition.

In addition, of course, the context implicitly contains a set of rewrite rules and linear rules derived from axioms, definitions, and previously proved theorems. This set of rules is here called the *library*.

This elaborate notion of 'context' is used implicitly not only by the rewriter but also by the linearization and augmentation procedures.

8.2. Polynomials

A *polynomial* is a five-tuple, $\langle c, alist, hyps, lits, supports\rangle$. The first field, c, is an integer constant. The second, *alist*, is a list of pairs $\langle t_i, k_i\rangle$, where each t_i is a term, called a *multiplicand*, and each k_i is an integer, called the *coefficient* of the corresponding multiplicand. The pairs in *alist* are ordered according to the multiplicands, with the heaviest first, and no two distinct pairs have identical multiplicands. The multiplicand in the first pair of the *alist* (the heaviest) is called the *key multiplicand* of the polynomial and the *sign* of the polynomial is the sign of the key multiplicand's coefficient. The third field of a polynomial, *hyps*, contains a set of terms. The fourth and fifth fields, *lits* and *supports*, contain sets of LISP objects. (Note: the last three fields all contain LISP lists treated as sets. However, we use EQUAL to compare elements in the hyps field but EQ to compare elements in the other two fields.)

The *formula represented* by a polynomial with constant c, *alist* $(\langle t_1, k_1\rangle, \ldots, \langle t_n, k_n\rangle)$, and *hyps* $h_1, \ldots, h_m$ is

$$h_1 \wedge \cdots \wedge h_m \rightarrow c + k_1 * t_1 + \cdots + k_n * t_n \leq 0.$$

The *lits* field of a polynomial contains the literals linearized to produce the polynomial or its ancestors. The *hyps* of a polynomial contain the linearization hypotheses. The supports field contains a variety of things: the names of the axioms, definitions, and lemmas used in the derivation of the polynomial, the literals used to relieve linearization hypotheses or the hypotheses of rewrite and linear rules contributing to the derivation of the polynomial, and special marks explained in Section 8.7.

We say a polynomial is *impossible* iff its constant is greater than 0 and no coefficient is negative. We say a polynomial is *vacuous* iff its constant is less than or equal to 0 and no coefficient is positive. Observe that if a polynomial is impossible the conclusion of the formula it represents is contradictory. Similarly, if a polynomial is vacuous then the conclusion of its formula is trivially true.

8.3. Converting terms to polynomials

The process of converting a literal into one or more polynomials is called 'linearization'. Linearization implicitly takes place in a context (as does rewriting). The linearization of *lit* is either NIL or a set of sets of polynomials. If the result is NIL, we draw no arithmetic conclusions from assuming *lit*. Otherwise, the answer represents a formula, *form*, obtained by disjoining (across the set) the result of conjoining (across each element) the formulas represented by each polynomial. It is a theorem that *form* is implied by the assumptions in the context.

Below we show some examples of literals linearized and the formulas represented by the answers. In each of the cases below, the *lits* field of the polynomials returned is $\{lit\}$ and the support fields is $\{\}$

lit	formula represented by result of linearization
$I < J \doteq I$	$I \lesssim J \rightarrow 1 + -1 * J + 2 * I \leq 0$
$I = J \doteq I$	$I \lesssim J \rightarrow 0 + -1 * J + 2 * I \leq 0$
$I \neq J \doteq I$	$0 + 1 * J + -2 * I \leq 0$
	$I \lesssim J \wedge I \in \mathcal{N} \rightarrow 1 + -1 * J + 2 * I \leq 0$
	$I \in \mathcal{N} \overset{\vee}{\rightarrow} 1 + 1 * J + -2 * I \leq 0.$

In order to describe the linearization process we need three auxiliary concepts. The first is the notion of the *zero polynomial depending on lit*, which is the polynomial with constant 0, empty *alist*, *hyps*, and *supports* fields, and *lits* field $\{lit\}$.

The other two concepts we mention informally here and define precisely after discussing linearization. One concept is that of 'inserting a hypothesis *hyp* into a polynomial p', which, roughly speaking, is the construction of a polynomial identical to p except that *hyp* is included in the hyps field. The third notion is that of 'adding a term t positively (or

negatively) to a polynomial p'. Roughly speaking, this means constructing a new polynomial by adding t to (or subtracting t from) p. However, when we add a term to a polynomial we may also insert hypotheses.

Careful treatment of the hypotheses is essential to the utility of our linear arithmetic procedure. Omitting a hypothesis that is not known to be true can cause unsoundness. But failure to recognize that a hypothesis is already known to be true or false can cause unnecessary case splitting or infinite looping (as the system may case split repeatedly on the same condition). Exactly how the linearization procedure handles the hypotheses depends upon how the procedure is being used.

It is useful to distinguish two different occasions on which we linearize terms. The first, and simpler of the two, is during the process of rewriting a literal after we have set up our polynomial data base. Linearization is used both to relieve hypotheses of rewrite rules and to augment the data base. But the data base produced by pushing the polynomials is not saved—we are only looking for a contradiction. Hence we need not track our dependency on literals and can use the type alist, TA, supplied by the context of the rewriter, to check the truth or falsity of some linearization hypotheses. If a hypothesis is true under TA we need not include it. If it is false we should avoid producing a polynomial requiring its truth.

The second occasion we use linearization is when we are setting up the initial data base. In this case we must track our dependencies on literals very carefully to avoid tail biting. During the initialization of the data base we therefore use a context in which TA is empty—preventing the unreported use of type set information—and use LITS-THAT-MAY-BE-ASSUMED-FALSE and HEURISTIC-TA to determine the truth or falsity of some linearization hypotheses. For example, if the complement of a required linearization hypothesis is in LITS-THAT-MAY-BE-ASSUMED-FALSE (which, recall, is the clause being proved), then we need not include it in the hyps field of the polynomials but must include it in the supports field.

Looking for a hypothesis or its complement in LITS-THAT-MAY-BE-ASSUMED-FALSE is not as powerful as computing its type set. For example, the type set mechanism could deduce the truth of (NUMBERP t) from the assumption $t = A \oplus B$. Nevertheless, we have adopted this approach because we must know which literals in the clause are being used when a required linearization hypothesis is omitted. However, recall that if we believe a hypothesis is false we simply avoid producing a polynomial requiring its truth. The soundness of the theorem prover is unaffected by the validity of our belief that a hypothesis is false: at worst we deny the system access to information it could have used. In this case it is irrelevant to track dependencies, since no polynomial is produced. Thus, we can afford to use a type alist as a heuristic device to avoid the production of certain polynomials. This is the role of HEURISTIC-TA in the context and it encodes the negations of all the literals of the clause.

Except where noted, all type set computations are done with respect to TA.

We say a term t is *possibly numeric* if the type set of t under HEURISTIC-TA (or, if HEURISTIC-TA is NIL, under TA) is {**NUMBERP**}.

The *positive (or negative) linearization of a literal lit* is either NIL or a set of sets of polynomials as described below. In the description below we handle the positive case only. The negative case is identical to the positive case for the complement of *lit* with one exception: the lits field of all the polynomials constructed contain *lit* rather than its complement.

If any polynomial in any element of the answer contains F in its hyps field we delete that polynomial from the element.

If *lit* is of the form (LESSP *lhs rhs*) the answer is $\{\{poly\}\}$ where *poly* is the result of adding the term (ADD1 *lhs*) positively to the result of adding the term r.h.s. negatively to the zero polynomial for *lit*.

If *lit* is of the form (EQUAL *lhs rhs*) and either *lhs* or *rhs* is possibly numeric, the answer is $\{\{poly_1\, poly_2\}\}$ where $poly_1$ is the result of adding *lhs* positively to the result of adding r.h.s. negatively to the zero polynomial for *lit*, and $poly_2$ is obtained by the symmetric procedure (swapping the roles of *lhs* and *rhs*).

If *lit* is (NOT (LESSP *lhs rhs*)) the answer is $\{\{poly\}\}$ where *poly* is the result of adding *rhs* positively to the result of adding *lhs* negatively to the zero polynomial for *lit*.

If *lit* is (NOT (EQUAL *lhs rhs*)) and either *lhs* or *rhs* is possibly numeric, then let $poly_1$ be the result of inserting (NUMBERP *lhs*) and (NUMBERP *rhs*) hypotheses into the result of adding (ADD1 *lhs*) positively to the result of adding *rhs* negatively to the zero polynomial for *lit*, and let $poly_2$ be obtained by the symmetric procedure (swapping the roles of *lhs* and *rhs*). If $poly_1$ is impossible, the answer is $\{\{poly_2'\}\}$ where $poly_2'$ is obtained from $poly_2$ by adding to its *hyps* field those of $poly_1$; if $poly_2$ is impossible, the answer is $\{\{poly_1'\}\}$ where $poly_1'$ is obtained from $poly_1$ by adding to its *hyps* field those of $poly_2$; otherwise the answer is $\{\{poly_1\}\{poly_2\}\}$.

If none of the above four cases obtains, the answer is NIL.

This concludes the definition of linearization.

The result of adding a term to a polynomial involves manipulating the alist of the polynomial (and possibly the hyps field). The following subsidiary concept is used:

The result of *inserting a term t with a coefficient of n into the alist field of a polynomial poly* is the polynomial that results from *poly* by modifying its alist as follows. If the type set of t does not include **NUMBERP**, do not modify the alist (since our arithmetic functions coerce non-NUMBERP arguments to 0); if there is a pair with multiplicant t in the alist, increment the coefficient of that pair by n; otherwise, add the pair $\langle t, n \rangle$ to the alist (maintaining the previously noted ordering of entries).

The result of *adding a term t with parity p to a polynomial poly* is the

polynomial obtained as follows:

If t is a constant, then if t is a natural number, increment (decrement) the constant of *poly* by t (according to whether p is positive or negative) and return the result; otherwise return *poly* (since non-NUMBERPS are coerced to 0).

If t is (ADD1 x), increment (decrement) the constant in *poly* by 1 (according to whether p is positive or negative) and add x with parity p to the result.

If t is (SUB1 x), then if p is positive: decrement the constant in *poly* by 1 and add x with parity p to the result; otherwise p is negative: insert the hypothesis (NOT (LESSP x 1)) into *poly*, increment the constant in the resulting polynomial by 1, and add x with parity p to the result.

If t is (PLUS x y), add y with parity p to the result of adding x with parity p to poly.

If t is (DIFFERENCE x y), then if p is positive: add x positively to the result of adding y negatively to *poly*; otherwise p is negative: insert the hypothesis (NOT (LESSP x y)) into *poly* and then to the result add x negatively to the result of adding y positively.

If t is (TIMES n x), where n is a natural number, insert x with a coefficient of n (or $-n$) (according to whether p is positive or negative) into the alist field of *poly* and return the result.

Otherwise, insert t with a coefficient of 1 (or -1) (according to whether p is positive or negative) into the alist field of *poly* and return the result.

This completes the definition of how to add a term to a polynomial.

The result of *inserting the hypothesis hyp into a polynomial poly* is obtained as follows.

If *hyp* is (NOT (LESSP x 1)) and the type set of x is {**NUMBERP**}, insert the hypothesis (NOT (EQUAL x 0)) into *poly* instead and return the result.

If *hyp* is (NOT (LESSP x 1)) and the complement of (NUMBERP x) occurs as some literal lit in LITS-THAT-MAY-BE-ASSUMED-FALSE, add lit to the supports field of *poly* and return the result.

If *hyp* is (NOT (EQUAL (DIFFERENCE u v) 0)), insert the hypothesis (LESSP v u) into *poly* and return the result.

If *hyp* is (NOT (EQUAL (ADD1 x) 0)), return *poly*.

If *hyp* is (NOT (EQUAL n 0)) where n is any constant other than 0, return *poly*.

If *hyp* is (NOT (EQUAL 0 0)), insert the hypothesis F into *poly* and return the result.

If the type set of *hyp* is {**TRUE**}, return *poly*.

If the type set of *hyp* does not include **TRUE**, insert the hypothesis F into *poly* and return the result.

If the type set of t (computed with HEURISTIC-TA) does not include **TRUE**, insert the hypothesis F into *poly* and return the result.

If the complement of *hyp* occurs as some member, *lit*, of LITS-THAT-MAY-BE-ASSUMED-FALSE, add *lit* to supports field of *poly* and return the result.

Otherwise, add *hyp* to the hypothesis field of *poly* and return the result.

8.4. Combining polynomials

Suppose p_1 and p_2 are polynomials with the same key multiplicand, t, and opposite signs. Let the coefficients of t in p_1 and p_2 be k_1 and k_2, respectively. By *cross-multiplying and adding* p_1 and p_2 we can form a new polynomial whose key multiplicand, if any, is smaller than t. The polynomial obtained has as its constant $k_2 * c_1 + k_1 * c_2$, where c_1 and c_2 are the constants of p_1 and p_2, respectively. The alist of the new polynomial is obtained from the alists of p_1 and p_2 by multiplying each coefficient in the first by k_2 and each coefficient in the second by k_1, then merging the two alists (adding together the coefficients of identical multiplicands and deleting any pair with a 0 coefficient). The hyps, lits, and supports fields of the new polynomial are the unions of the corresponding fields of p_1 and p_2 (comparing with EQUAL or EQ as appropriate).

Observe that if the formulas represented by p_1 and p_2 are both true in the context then so is the formula represented by the result of cross-multiplying and adding.

For each term t we define the *non-negative assumption* for t to be the polynomial obtained by linearizing the theorem $0 \leq t$, i.e. the polynomial representing $0 + -1 * t \leq 0$, with empty hyps, lits, and supports fields. Any polynomial with a positive first coefficient can be cross-multiplied and added to the appropriate non-negative assumption to obtain a true polynomial differing from the initial polynomial only in that the first pair in the alist is missing.

8.5. Pushing polynomials into the data base

Conceptually, our linear arithmetic data base is just a set of polynomials. To make it easier to find all the polynomials with a given key multiplicand and sign, we actually partition the data base into 'pots' according to their key multiplicands and further partition each pot according to the sign of the polynomials. We then store the pots in order according to the weight of the key multiplicands. However, for the purposes of this paper we treat the data base simply as a set of polynomials.

The fundamental operation on the data base is to add new polynomials to it and deduce the consequences by cross-multiplying and adding. However, recall that during the simplification of a given literal we wish not to use polynomials that descend from LITS-TO-BE-IGNORED-BY-LINEAR. We say a polynomial poly is *available* if no element of LITS-TO-BE-IGNORED-BY-LINEAR is EQ to any element of the lits or supports fields of poly.

The result of *pushing a set of polynomials s into a data base db* is the closure of the union of *db* and *s* under the following two operations:

1. For any available member polynomial x with positive sign, include the result of cross-multiplying and adding x to the non-negative assumption for the key multiplicand of x, provided that result is non-vacuous.
2. For any two available member polynomials x and y with the same key multiplicand and opposite signs, include the result of cross-multiplying and adding x and y, provided that result is non-vacuous.

The above description fails to describe our code in three respects. First, because the initial data base is closed under the operations above, it suffices to consider only the new polynomials and their consequences. Second, the order in which we combine polynomials is not specified. Third, since we are seeking to derive an impossible polynomial the code that closes the data base halts when a cross multiply and add produces an impossible polynomial. The hyps, lits, and supports fields of the impossible polynomial found influence the subsequent proof attempt. Thus, if more than one impossible polynomial can be derived from the assumptions, the order in which polynomials are combined is relevant.

8.6. Augmenting the data base

In this subsection we explain how we use previously proved theorems to augment the data base of polynomials.

A *linear rule* is a four-tuple ⟨*name, hyps, concl, max-term*⟩, where *name* is the user-supplied name of a formula, *hyps* is a list of terms, *concl* is a term of the form (LESSP *x y*) or (NOT (LESSP *x y*)), the positive linearization of *concl* (under empty TA, LITS-THAT-MAY-BE-ASSUMED-FALSE, and HEURISTIC-TA) is a singleton set containing a singleton set containing a polynomial *poly*, and *max-term* is one of the multiplicands in the *alist* of *poly* and has the following properties: (a) *max-term* is not a variable symbol; (b) the set of variables occurring in *concl* is a subset of the union of those occurring in *max-term* and those occurring in *hyps*; and (c) no other multiplicand in the *alist* of *poly* has larger size and contains a superset of the variables occurring in *max-term*.

Roughly speaking, linear rules are interpreted as follows. Whenever a new key multiplicand is introduced into the polynomial data base we search for applicable linear rules, finding each rule whose max-term can be instantiated to yield the key multiplicand in question. When we find such a rule we attempt to establish, by rewriting, the corresponding instance of each of the hypotheses in the hyps of the rule. Provided we succeed, we rewrite the appropriate instance of the concl of the rule and linearize it to obtain a polynomial. We then modify the hyps and supports fields of the polynomial to take into account the hypotheses assumed and lemmas and literals used during the rewriting. Then, provided certain

heuristic conditions are met, we push the resulting polynomial into the data base.

The restrictions on max-term above are motivated by two considerations. First, we want to ensure that once the variables in a maximal term are instantiated (by the pattern match with a key multiplicand) and the hypotheses are relieved (possibly instantiating variables occurring in hyps but not in max-term), then every variable in concl is instantiated. Second, since the polynomial produced from the instantiated conclusion can only be used to cancel its heaviest multiplicand, we try to select as our max-terms only those terms which might, under suitable instantiation, become the largest.

Linear rules are added to the system's library of rules whenever certain user-supplied formulas are proved. Suppose the user submits to the theorem prover a conjecture named *name* of the form (IMPLIES *hyp concl*). Suppose further the conjecture was tagged as a rewrite rule. Let *hyps* be the result of flattening the AND structure of *hyp*, i.e. the conjunction over *hyps* is *hyp*. If the conjecture is proved, we store in our library each four-tuple ⟨*name, hyps, concl, t*⟩ that is a linear rule. Actually, the recognition of candidate theorems is more sophisticated. For example, a simple (LESSP *x y*) or (NOT (LESSP *x y*)) theorem is recognized as a candidate and *hyp* defaults to *T*. If *concl* is a conjunction, we strip out the individual conjuncts and look for inequalities. These details are unimportant in this paper.

Before linearizing the instantiated conclusion of a linear rule we rewrite it to put the terms into normal form under the current set of rewrite rules. However, rather than rewrite the entire conclusion, we rewrite merely the two sides of the inequality to avoid applying linear arithmetic to the conclusion before we have normalized the terms.

The *rewritten form of term under substitution s*, where *term* is a term of the form (LESSP *lhs rhs*) or (NOT (LESSP *lhs rhs*)) is obtained as follows. Let *lhs'* and *rhs'* be obtained by rewriting *lhs* and *rhs*, respectively, under the substitution *s*. If *concl* is (LESSP *lhs rhs*), the rewritten form is (LESSP *lhs' rhs'*), otherwise, it is (NOT (LESSP *lhs' rhs'*)).

Pushing a linear rule ⟨*name, hyps, concl, max-term*⟩ *for multiplicand t into a data base db* produces a data base as follows. If *db* contains an impossible polynomial, return *db*. If there is no substitution *s* on the variables of *max-term* such that *s* applied to *max-term* is *t*, return *db*. Otherwise, push new frames onto both the lemma stack and the hyps stack, and, using *db* as DB, attempt to relieve the hypotheses *hyps*. This either fails or succeeds and delivers an extension *s'* of *s* and modifies the top frames of the two stacks. If the attempt fails, pop and discard the two frames added and return *db*. Otherwise, pop the lemma stack and let lemmas be the resulting set of items. Pop the hyps stack and let *hyps* be the resulting set of terms. Let {{*poly*}} be the positive linearization of

the rewritten form of concl under s'. If for any reason the linearization does not produce such a structure or if there is a multiplicand in the alist of *poly* that is distinct from, as large as, and 'worse than' every key multiplicand in *db*, then return *db*. (We use the same sense of 'worse than' defined on p. 110 of Boyer and Moore, 1979.) Otherwise, let *poly'* be obtained from *poly* by setting the hyps field to the union of the hyps of *poly* and *hyps*, and setting the supports field to the union of {*name*} and lemmas. Return the result of pushing *poly* into *db*.

The result of *augmenting a data base db with linear rules for a set of multiplicands s* is a polynomial data base constructed as follows. If *db* contains an impossible polynomial, return *db*. If *s* is empty, return *db*. Otherwise, let *db'* be the result of iteratively expanding *db* by pushing into it each linear rule about any multiplicand in *s*. Return the result of augmenting *db'* with linear rules for every non-variable key multiplicand in *db'* that is not a key multiplicand in *db*.

The resulting of *pushing a set of polynomials s into a data base db and augmenting with linear rules* is the data base constructed as follows. Let *db'* be the result of pushing *s* into *db*. Return the result of augmenting *db'* with linear rules for all non-variable key multiplicands in *db'* that are not key multiplicands in *db*.

8.7. The interface between linear arithmetic and rewriting

In this subsection we describe the operation of pushing terms (as opposed to polynomials) into a data base and augmenting with lemmas. This operation is the entry to the linear arithmetic procedure from the rest of the simplifier. It is used both to construct the initial data base and to rewrite inequalities by showing they contradict our previous assumptions.

Given a data base encoding our current linear assumptions and a list of terms to assume true (or false) we desire to construct a new data base containing the conjunction of the old and new assumptions. If each term linearized into a conjunction of polynomials the task would be simple: linearize each term, push each polynomial produced and then augment the data base with linear rules. However, some terms, e.g. $I \neq J$, linearize to a disjunction of polynomials: either $I < J$ or $J < I$. A single data base cannot, in general, represent the assumption $I \neq J$. However, if $I < J$ contradicts other assumptions, we can push $J < I$, and vice versa. Our initial implementation simply ignored disjoined polynomials, but we found several cases where that prevented proofs. We dismissed as too expensive (without even implementing it) the much stronger approach of producing a data base for each combination of alternatives and carrying out the desired simplifications in each of them.

The result of *pushing the list of terms s positively (or negatively) into the data base db and augmenting with linear rules* is the data base

obtained as follows: linearize each term in *s* (positively or negatively, as indicated). Each answer can be classified into one of three categories: it is a singleton list containing a list of polynomials, in which case we say the polynomials are *conjuncts*; it is a doubleton list containing two lists of polynomials, in which case the doubleton is said to be a *pair of alternatives*; or it is neither of the above, in which case the linearized term was not recognized as an arithmetic equality or inequality. Let *db'* be the result of pushing all of the conjunct polynomials into *db* and augmenting with linear rules. Then, iteratively expand *db'* by considering each pair of alternatives {*poly-lst*$_1$ *poly-lst*$_2$} and doing the following: if the result of pushing *poly-lst*$_1$ into *db'* and augmenting with linear rules contains an impossible polynomial, modify the hyps and supports fields of the polynomials in *poly-lst*$_2$ by unioning into them the hyps and supports fields (respectively) of the impossible polynomial found, and then replace *db'* by the result of pushing the modified *poly-lst*$_2$ into *db'* and augmenting with linear rules; otherwise (if pushing *poly-lst*$_1$ produced no contradiction), perform the symmetric test with *poly-lst*$_2$ and modify and push *poly-lst*$_1$ if a contradiction is found; otherwise, do not expand *db'* on this iteration. When all alternatives have been considered, return the final *db'*.

As described above the consideration of the alternatives is needlessly expensive: if pushing *poly-lst*$_1$ into *db'* does not lead to a contradiction but pushing *poly-lst*$_2$ does, we push *poly-lst*$_1$ into *db'* again after modifying its hyps and supports. Of course, the data base produced by pushing the modified *poly-lst*$_1$ is exactly the same as that produced by pushing *poly-lst*$_1$ except that the consequences derived from the modified *poly-lst*$_1$ have additional hyps and supports. But pushing and augmenting can be quite expensive since it causes conditional rewriting and backwards chaining. Our implementation avoids the near-duplicated pushes by putting a unique mark in the supports field of *poly-lst*$_1$ before it is pushed the first time. Because of the way the supports field is propagated by cross-multiplication and adding, every consequence deduced from members of *poly-lst*$_1$ is marked in the resulting data base, db'_1. If db'_1 does not contain a contradiction but pushing *poly-lst*$_2$ into *db'* does, we visit every marked polynomial in db'_1 and update the hyps and supports fields with those from the contradiction found with *poly-lst*$_2$.

8.8. Rewriting terms and relieving hypotheses

We now complete our description of how the rewriter has been modified to use linear arithmetic.

At the point in rewriting where we used to 'rewrite with lemmas' (p. 122 of Boyer and Moore, 1979) we now try linear arithmetic first, provided the atom of the term being rewritten is a LESSP or EQUAL

expression and the objective of rewriting is either to show the term T or to show it F. If the objective is to show the term T, we push the term negatively into DB and augment with linear rules. If the resulting data base contains an impossible polynomial, *poly*, we add to the top frame of the *hyps* stack the terms in the *hyps* field of *poly*, add to the top frame of the lemma stack the items in the *supports* field of poly, and return T as the value of the rewritten term. If, on the other hand, the objective is to show the term *F*, we do the symmetric operation.

In an earlier implementation we tried using linear arithmetic to simplify LESSP or EQUAL terms even when the objective was not T or F. In particular, we first tried pushing the term positively and if that produced no contradiction, we tried pushing it negatively. To our surprise, this increased the total number of conses used during the proofs of the theorems in Appendix A of Boyer and Moore (1979) from roughly 6 million to roughly 10 million without significantly shortening the proofs produced. We therefore abandoned the idea of using linear arithmetic except when we had a clear objective to establish.

The remaining changes to the rewriter are motivated by the need to track accurately which literals are being used when we augment the initial data base with linear rules. To prevent surreptitious use of type information, we set TA to NIL during the construction of the data base. This cripples the rewriter described in Boyer and Moore (1979) since it has no assumptions with which to work while trying to relieve the hypotheses of linear rules. We use LITS-THAT-MAY-BE-ASSUMED-FALSE to encode assumptions in a way that permits us to track dependencies.

As noted on p. 124 of Boyer and Moore (1979), just before the rewriter returns its answer, *ans*, it asks whether *ans* has typeset {TRUEP} or {FALSEP} under TA and, if so, returns T or F, as appropriate, instead. Now we ask, in addition, whether *ans* is EQUAL to some member, *lit*, of LITS-THAT-MAY-BE-ASSUMED-FALSE. If so, we return F instead, but we add LIT to the top frame of the lemma stack. That literal will ultimately be deposited in the supports field of any polynomial depending on this rewrite. Similarly, if the complement of *ans* occurs in LITS-THAT-MAY-BE-ASSUMED-FALSE we return T instead and store the corresponding *lit* in the lemma stack, provided that *ans* is Boolean valued or that the sense of equality to be preserved by this rewrite is propositional equivalence.

As noted on p. 122 of Boyer and Moore (1979), when we are trying to relieve a hypothesis *hyp* under some substitution *s* and *s* does not instantiate every variable of *hyp* we use TA to try to extend *s* to make the instantiation of *hyp* true. We now use LITS-THAT-MAY-BE-ASSUMED-FALSE in an analogous way, recording on the lemma stack the literals used. In addition, if any hypothesis to be established is on LITS-THAT-MAY-BE-ASSUMED-FALSE we abandon the attempt to relieve the hypotheses.

8.9. Deriving equalities from the data base

In this subsection we define the concepts necessary to describe how we generate from the polynomial data base equality literals to add to the clause being proved.

We say a polynomial *p isolates* t_i *positively* (*or negatively*) iff t_i is a multiplicand in the *alist* of *p*, the coefficient, k_i, of t_i is positive (negative), the constant of *p* and all coefficients other than that of t_i are negative (positive) and multiples of k_i, and the lits field of *p* is not a singleton set containing a negated equality.

Note that if a polynomial with constant *c* and *alist* $(\langle t_1, k_1\rangle, \ldots, \langle t_n, k_n\rangle)$ isolates t_i positively then the concluding inequality in the formula represented by the polynomial can be put into the form:

$$t_i \le c' + k_1' * t_1 + \cdots + k_{i-1}' * t_{i-1} + k_{i+1}' * t_{i+1} + \cdots + k_n' * t_n,$$

where c' and the k_i''s are all natural numbers. We call t_i the *isolated term* of the polynomial and

```
(PLUS c'
      (TIMES k1' t1) ...
      (TIMES ki-1' ti-1)
      (TIMES ki+1' ti+1) ...
      (TIMES kn' tn))
```

the *conglomerated term corresponding to t*.

The *result of multiplying* (*or dividing*) *a polynomial poly by an integer n* is a five-tuple <*c, alist, hyps, lits, support* >, where *c* is the constant of *poly* multiplied (or divided) by *n*, *alist* is obtained from the alist of *poly* multiplied (or divided) by *n*, and the remaining fields are those of the same name in *poly*. Multiplying a polynomial by *n* produces a polynomial. Dividing a polynomial by *n* produces a polynomial only if *n* divides the constant and each coefficient.

We say a polynomial $poly_2$ is a *complementary multiple* of a polynomial $poly_1$ iff there is a negative integer *n* such that the result of multiplying $poly_1$ by *n* is $poly_2$.

We say two polynomials, $poly_1$ and $poly_2$ are *mates on a term t* iff $poly_1$ isolates *t* (positively or negatively) and $poly_2$ is a complementary multiple of the result of dividing $poly_1$ by the coefficient of *t* in $poly_1$.

If a data base contains two mates, $poly_1$ and $poly_2$, on a term *t* then, under the conjunction over the union of the hypotheses in the two polynomials, we can derive an equation between *t* and its corresponding conglomerated term. In the next subsection we describe how we process mated polynomials.

8.10. Simplifying clauses

Roughly speaking, to simplify a clause we first set up a polynomial data base derived by assuming all the literals of the clause false. If the data base contains an impossible polynomial we are done. Otherwise, we look for mated polynomials and process them. If we find no mates, we sweep the clause from left to right rewriting each literal in turn, using the polynomial data base previously set up but ignoring certain polynomials in it. At each stage we must deal with the linearization hypotheses arising from polynomials we have used.

The *polynomial data base for the clause cl* is constructed as follows. First, we bind LITS-THAT-MAY-BE-ASSUMED-FALSE to *cl*, HEURISTIC-TA to the type alist encoding the falsity of every term in *cl*, TA to NIL, and LITS-TO-BE-IGNORED-BY-LINEAR to NIL. Then we push *cl* negatively into the empty data base and augment it with linear rules and return the result.

If a clause is a consequence of simple linear arithmetic, the polynomial data base will contain an impossible polynomial. However, because TA is NIL during the augmentation of the polynomial data base we sometimes fail to find contradictions (involving linear rules) that would be found if TA contained the negations of all the literals. Therefore, when we simplify a clause we take time out to augment the data base under the stronger TA, hoping to generate an impossible polynomial. If no contradiction is found we discard the resulting data base since it contains 'hidden' dependencies.

The control structure of clause simplification exploits the fact that clauses are represented by sequences, not sets. In addition, we must agree upon a way to mark the 'current literal' in a clause. We will continue to use set brackets to denote clauses but will consider the objects described to be sequences and will attach importance to the order of the literals. When we use the 'union' operator, $\cup$, in connection with clauses, we mean concatenation. The 'current literal' of a clause will be enclosed in square brackets. Thus, $\{\neg p\ [q]\ r\}$ is a clause whose first literal is $\neg p$ and whose current literal is q.

The result of *splitting the clause* $\{\ldots\ p\ [q]\ r\ \ldots\}$ *on* $h_1, \ldots, h_n$ is the set consisting exactly of each clause of the form $\{\ldots\ p\ h_i\ q\ [r]\ \ldots\}$, where $1 \le i \le n$. If there is no literal r to the right of the selected literal q in the input, the clauses in the output set have no selected literal.

The result of *adding the hypotheses* $h_1, \ldots, h_n$ *to the clause* $\{\ldots\ p\ [q]\ r\ \ldots\}$ is the clause $\{\ldots\ p\ \neg h_1 \ldots \neg h_n\ [q]\ r \ldots\}$.

The result of *splicing the clause segments* $seg_1, \ldots, seg_n$ *in place of the selected literal in* $\{\ldots\ p\ [q]\ r\ \ldots\}$ is the set consisting exactly of the clauses $\{\ldots p\} \cup seg_i \cup \{[r]\ldots\}$, where $1 \le i \le n$. If there is no literal r to the right of the selected literal q in the input, the clauses in the output set have no selected literal.

We now define the heuristic for controlling the introduction of derived equalities and the way we handle the hypotheses generated by the derivation.

The *heuristics for equality introduction* for two terms t and t' is the condition that the type set of both t and t' contains **NUMBERP** and that no clause in the ancestry of the clause being simplified is a 'result of adding the equation of t and t'' as defined below. (Every clause processed by the theorem prover comes with a complete history of its derivation, including its parent and the operations that produced it.)

The result of *introducing into a clause cl the equality between t and t' derived from two polynomials* $poly_1$ *and* $poly_2$ is the union of S_1 and S_2 defined below. Let *hyps* be the result of unioning together the hyps fields of the two polynomials and then adding the term (NUMBERP t) (unless the type set of t is {**NUMBERP**}) and the term (NUMBERP t') (unless the type set of t' is {**NUMBERP**}). S_1 is the singleton set containing the result of adding the hypothesis $t = t'$ to the result of adding the hypotheses *hyps* to *cl*. S_2 is the result of splitting *cl* on *hyps*. We say every clause in S_1 and S_2 is a *result of adding the equation of t and t'*.

To *sweep a clause cl*, $\{\ldots\ p\ [q]\ r\ \ldots\}$, construct a set of clauses as follows. If there is no selected literal, return $\{cl\}$. Otherwise, let TA be the type alist obtained by assuming false every literal in *cl* except the selected literal, q. Let LITS-THAT-MAY-BE-ASSUMED-FALSE and HEURISTIC-TA be NIL. Let LITS-TO-BE-IGNORED-BY-LINEAR be the list containing q and every literal to its left in CL, that 'rewrote to F' as defined below. Push empty literal to its left in CL, that 'rewrote to F' as defined below. Push empty frames onto both the hyps stack and the lemma stack. Let q' be the result of rewriting q. If q' is F, we say q *rewrote to* F. Pop and discard the top frame of the lemma stack. (Actually, the names in that frame are accumulated and eventually printed as part of a description of the proof. In addition, they are used to build a dependency graph when the system's library is updated at the end of successful proofs.) Pop the top frame of the hyps stack and let *hyps* be the set of terms in that frame.

Let *segs* be the set of clause segments obtained by normalizing the IFS in q' and splitting out each branch, as shown on page 124 of Boyer and Moore (1979). For example, if q' is $(G\ (\text{IF}\ a\ b\ c))$ then we obtain two clause segments $\{\neg a\ (Gb)\}$ and $\{a\ (Gc)\}$. The final answer is obtained by recursively sweeping each clause in the union of S_1 and S_2 (defined below) and unioning together the results. S_1 is the result of splicing *segs* in place of the selected literal in the clause obtained by adding *hyps* to *cl*. S_2 is the result of splitting *cl* on *hyps*.

To *simplify a clause, cl,* construct a set of clauses as follows. Select the first literal of *cl* as the current literal. Let DB be the polynomial data base for *cl*. If there is an impossible polynomial, *poly*, in DB, return the result of splitting *cl* on the hypotheses of *poly*. Otherwise, let TA be the type *alist* obtained by assuming all literals of *cl* false. If there is an impossible

polynomial, *poly*, in the result of augmenting DB with linear rules for every key multiplicand in DB, return the result of splitting *cl* on the hypotheses of *poly*. If there are two polynomials in DB that are mates on some term *t* with conglomerated term *t'* and the heuristics for equality introduction are satisfied, return the result of introducing into *cl* the equality between *t* and *t'* derived from the two polynomials. Otherwise, sweep *cl* and return the result.

9. EFFICIENCY

The incorporation of the linear procedure sketched above has dramatically improved the performance of our theorem prover on arithmetic problems. For example, compared to the theorem prover described in Boyer and Moore (1979) the system spends 40% less time processing the theorems and definitions in Appendix A of Boyer and Moore (1979). Furthermore, we have been able to eliminate the need for the user to state explicitly linear facts. Thus, our original objective was achieved. Among the theorems proved by the latest version of the theorem prover are the invertibility of the Rivest, Shamir, and Adleman public key encryption algorithm (Boyer and Moore, 1984), Wilson's theorem (Rusinoff, 1983), Gauss's law of quadratic reciprocity (David M. Rusinoff led the theorem prover to Gauss's law) and the Church-Rosser theorem (Shankar, 1985). All of these proofs involved a substantial amount of linear arithmetic reasoning.

We now turn to our observation that theoretical efficiency is not a good measure of the utility of a linear procedure in a larger system. Let us reconsider our decision to use the simple 'cross-multiply and add' algorithm instead of more efficient ones. Might our handling of arithmetic be sped up by the use of another propagation algorithm? The answer is no; an insignificant portion of the time is devoted to the problem of propagating polynomials through the data base. Consider what else must be done. Terms must be linearized, the key multiplicands in the data base determined, interesting lemmas must be selected and instantiated (and their hypotheses must be relieved by nonarithmetic reasoning) with due caution for avoiding traps like 'pumps', the lemmas, literals, and hypotheses supporting the derivation of each inequality must be recorded and maintained, one must avoid biting one's own tail, one must be able to 'pop' or 'undo' the effect of pushing an inequality and all of the linear rules it introduced, and when a linear contradiction is found one must handle the additional cases raised by the particular contradiction found.

Consider the MS proof sketched above. Of the total time spent in arithmetic reasoning in the second phase of the proof (5.8 seconds) only 1.8% (0.119 seconds) is spent propagating polynomials. The rest is spent taking care of the issues listed above. Thus, the availability of an

instantaneous oracle for linear arithmetic problems would speed up the MS proof by an insignificant amount.

Perhaps more realistic data is that obtained during the proof of the verification conditions for the FORTRAN version of our fast string searching algorithm. We regard this set of 53 lemmas and verification conditions to be quite representative of the verification of practical programs. The verification conditions establish that the preprocessor correctly sets up a global COMMON array and that the search algorithm correctly computes the location of the first occurrence of the pattern in the text, if there is an occurrence, or else correctly announces that no occurrence exists. Furthermore, the v.c.s establish that there are no array bounds violations, arithmetic overflows, or other run time errors, and that both subroutines terminate. To prove these v.c.s the system must first establish several important lemmas about strings and string searching. These lemmas are proved inductively from the definitions of such concepts as 'a string over a finite alphabet', 'leftmost occurrence' and our DELTA1 function. The definitions themselves are proved satisfiable by the system before they are admitted. The admission of the definitions, proofs of the lemmas, and proofs of the v.c.s all require both arithmetic and non-arithmetic reasoning, as is common in the verification of programs that compute non-arithmetic functions on arrays and tables (e.g. searching, sorting, hashing).

The total time taken is 1417 c.p.u. seconds (23.6 c.p.u. minutes). We push terms into the polynomial data base 2637 times. Relatively few linear rules are available for instantiation. Only once in every six calls does the augmentation procedure find a lemma that is judged to be relevant to the data base. (Thus, one can infer that not an inordinate amount of time is spent pursuing instantiations.) The total time consumed while pushing terms into the data base is 357 c.p.u. seconds, 25% of the total proof time. But only 38.5 seconds is spent pushing polynomials. That is, in this fairly representative verification problem, an instantaneous oracle for linear inequalities would reduce the time in arithmetic reasoning by 10.7% and would reduce the time for the overall proof by only 2.7%.

One should not get the idea that the linear procedure is not doing anything for us. As we have already said, the presence of built-in arithmetic speeds up the theorem prover dramatically and makes the system far more rugged when applied to arithmetic problems. But the timing difference between the simple algorithm and theoretically more efficient ones is insignificant. Furthermore, it is not necessarily the case that a more efficient propagation algorithm would make the interface run faster. In particular, if the faster algorithm used a more complicated data structure for the data base and required a destructive push operation, it is probably the case that the interface would spend more time than it does

now in such activities as popping the data base and exploring it for the key multiplicands.

10. CONCLUSION

We have made and documented three observations: linear arithmetic is inadequate for the arithmetic needs of program verification; integrating a linear arithmetic procedure into a theorem prover for a richer theory is surprisingly difficult; and the theoretical efficiency of a linear arithmetic procedure is a poor measure of its utility to a larger system.

We believe these same observations can be made about decision procedures in general. Let us quickly review our observations while considering decision procedures.

Decidable theories are inadequate for the specification of most programs. The situation is improved somewhat by the work of Oppen and Nelson (1979) which shows how one can construct a system of co-operating decision procedures for disjoint theories. But in our experience most theories of use to program verification are not disjoint. For example, the function LEN connects the theory of lists to that of the naturals and DELTA1 connects character strings to naturals.

But what makes it hardest to apply the work on decision procedures to program verification is the presence of user defined functions. DELTA1 is one example of a function that cannot be anticipated by the designer of the decision procedure. Other examples we have seen recently are: 'the number of times X occurs in Y', 'the number of processors that voted for X', and 'the length of the non-circular path defined by tracing the non-0 indices stored in the array A starting at location I'. Such functions are introduced not by the designer of the theorem prover but by the user when he is confronted with the need to specify a given program. Since decision procedures for these extended theories are not generally available, one must have more powerful proof techniques or be forced to assume the more doubtful conjectures behind a program's correctness.

But decidable theories are common fragments of the theories used in the specification of programs. It is thus useful to integrate decision procedures with the more powerful methods. A natural goal is to make it unnecessary for the more powerful system to derive from explicit axioms and lemmas the theorems of the decidable theory. To achieve such integration is very difficult because one must identify each use to which the heuristic theorem prover puts axioms and lemmas and make the decision procedure serve in each of those roles.

Furthermore, the black box nature of the decision procedure is frequently destroyed by the need to integrate it. The integration forces into the theorem prover much knowledge of the inner workings of the procedure and forces into the procedure many features that are unnecessary when the problem is considered in isolation. Thus it is not possible to substitute one decision procedure for another nor can the selection (much less the implementation) of the original procedure be entirely independent of the needs of the larger system.

Finally, the time spent in the interface between the heuristic theorem prover and the decision procedure may dominate that spent in the decision procedure itself. Since efficiency in the decision procedure may not gain much overall, it is often not worth the effort to select more efficient procedures because of the complicated data structures and inflexible control strategies they employ to gain efficiency.

When sufficiently powerful theorem provers for program verification are finally produced they will undoubtedly contain many integrated decision procedures. But despite the fact that work on decision procedures is elegant, easily published, mathematically pleasing, and demands rather limited computational resources, the usefulness of that work to program verification is not easily evaluated. The difference between a black box and an integrated decision procedure is a lot of work. It is probably the case that much hard work on any given black box will be scrapped when the box is torn apart and reassembled inside a larger system. Indeed, we believe that the work on many procedures is simply irrelevant to the goal of constructing useful mechanical theorem provers since the use of a faster procedure will not necessarily speed up the overall system. We believe that the development of useful procedures for program verification must take into consideration the problems of connecting those procedures to more powerful theorem provers.

Acknowledgments

The research reported here was supported by National Science Foundation Grant MCS-8202943 and Office of Naval Research Contract N00014-81-K-0634.

REFERENCES

Bledsoe, W. W. (1975) A new method for proving certain Presburger formulas. Advance Papers, *Fourth Int. Joint Conf. on Artificial Intelligence*, Tbilisi, Georgia, USSR, pp. 15–20.

Bledsoe, W. W. and Hines, L. M. (1980) Variable elimination and chaining in a resolution-based prover for inequalities. In *5th Conference on Automated Deduction, Lecture Notes in Computer Science* (eds W. Bibel and R. Kowalski) pp. 70–87. Springer-Verlag, Berlin.

Boyer, R. S. and Moore, J. S. (1977) A fast string searching algorithm. *Commun. ACM* **20,** 762–772.

Boyer, R. S. and Moore, J. S. (1979) *A computational logic.* Academic Press, New York.

Boyer, R. S. and Moore, J. S. (1981a) Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The correctness problem in computer science* (eds R. S. Boyer and J. S. Moore). Academic Press, London.

Boyer, R. S. and Moore, J. S. (1981b) A verification condition generator for FORTRAN. In

The correctness problem in computer science (eds R. S. Boyer and J. S. Moore) Academic Press, London.
Boyer, R. S. and Moore, J. S. (1984) Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly* **91,** 181–189.
Cooper, D. C. (1972) Theorem proving in arithmetic without multiplication. In *Machine Intelligence* 7 (eds B. Meltzer and D. Michie), pp. 91–99. Edinburgh University Press, Edinburgh.
Gloess, P. Y. (1980) An experiment with the Boyer–Moore theorem prover: a proof of the correctness of a simpler parser of expressions. In *5th Conference on Automated Deduction, Lecture Notes in Computer Science,* pp. 154–169. Springer-Verlag, Berlin.
Hodes, L. (1971) Solving problems by formula manipulation. *Proc. Second Int. Joint Conf. on Artificial Intelligence,* 553–559. The British Computer Society.
King, J. C. (1969) A program verifier. Ph.D. Thesis, Carnegie–Mellon University.
Moore, J. S. (1979) A mechanical proof of the termination of Takeuchi's function. *Information Processing Letters* **9,** 176–181.
Nelson, G. and Oppen, D. C. (1979) Simplification by cooperating decision procedures. *ACM Transactions of Programming Languages* **1,** 245–257.
Russinoff, D. M. (1983) A mechanical proof of Wilson's theorem, Department of Computer Sciences, University of Texas at Austin.
Shankar, N. (1985) A mechanical proof of the Church–Rosser theorem. ICSCA-CMP-45, Institute for Computing Science, University of Texas at Austin.
Shostak, R. (1977) On the SUP-INF method for proving Presburger formulas. *JACM* **24,** 529–543.
Shostak, R. (1978) Deciding linear inequalities by computing loop residues, Computer Science Laboratory, SRI International, Menlo Park, Calif.
Shostak, R. (1979) A practical decision procedure for arithmetic with function symbols. *JACM* **26,** 351–360.

6

A
th
Cc

R.
Dep
Univ

Approach
for
oblems

Abstract

Many AI tasks can be formulated as constraint-satisfaction problems (CSPs), i.e. the assignment of values to variables subject to a set of constraints. Recognition of three-dimensional objects, puzzle solving, electronic circuit analysis and truth-maintenance systems are examples of such problems, and these are normally solved by various versions of backtrack search. In this work we show how advice can be automatically generated to guide the order in which the search algorithm assigns values to the variables, so as to reduce the amount of backtracking. The advice is generated by consulting relaxed models of the subproblems created by each value-assignment candidate. The relaxed problems are chosen to yield backtrack-free solutions, and the information retrieved from these models induces a preference order among the choices pending in the original problem.

We identify a class of CSPs whose syntactic and semantic properties make them easy to solve. The syntactic properties involve the structure of the constraint graph while the semantic properties guarantee some local consistencies among the constraints. In particular, tree-like constraint graphs can be easily solved and are chosen therefore as the target model for the relaxation scheme. Optimal algorithms for solving easy problems are presented and analysed. A scheme for constructing a 'best' constraint-tree approximation to a given constraint graph is introduced and, finally, the utility of using the advice is evaluated in a synthetic domain of CSP instances.

1. BACKGROUND AND MOTIVATION

1.1. Introduction

An important component of human problem-solving expertise is the ability to use knowledge about solving easy problems to guide the solution of difficult ones. Only a few works in AI (Sacerdoti, 1974; Carbonell 1983) have attempted to equip machines with similar capabi-