A Machine Program for Theorem-Proving†

Martin Davis, George Logemann, and Donald Loveland

Institute of Mathematical Sciences, New York University

The programming of a proof procedure is discussed in connection with trial runs and possible improvements.

In [1] is set forth an algorithm for proving theorems of quantification theory which is an improvement in certain respects over previously available algorithms such as that of [2]. The present paper deals with the programming of the algorithm of [1] for the New York University, Institute of Mathematical Sciences' IBM 704 computer, with some modifications in the algorithm suggested by this work, with the results obtained using the completed algorithm. Familiarity with [1] is assumed throughout.

Changes in the Algorithm and Programming Techniques Used

The algorithm of [1] consists of two interlocking parts. The first part, called the *QFl-Generator*, generates (from the formula whose proof is being attempted) a growing propositional calculus formula in conjunctive normal form, the "quantifier-free lines." The second part, the *Processor*, tests, at regular stages in its "growth," the consistency of this propositional calculus formula. An inconsistent set of quantifier-free lines constitutes a proof of the original formula.

The algorithm of [1] used in testing for consistency proceeded by successive elimination of atomic formulas, first eliminating *one-literal* clauses (one-literal clause rule), and then atomic formulas all of whose occurrences were positive or all of whose occurrences were negative (affirmative-negative rule). Finally, the remaining atomic formulas were to have been eliminated by the rule:

III. *Rule for Eliminating Atomic Formulas.* Let the given formula F be put into the form

$$(A \vee p) \ \& \ (B \vee \bar{p}) \ \& \ R$$

where A, B, and R are free of p. (This can be done simply by grouping together the clauses containing p and "factoring out" occurrences of p to obtain A, grouping the clauses containing $\bar{p}$ and "factoring out" $\bar{p}$ to obtain B, and grouping the remaining clauses to obtain R.) Then F is inconsistent if and only if $(A \vee B) \ \& R$ is inconsistent.

After programming the algorithm using this form of Rule III, it was decided to replace it by the following rule:

III*. *Splitting Rule.* Let the given formula F be put in the form

$$(A \vee p) \ \& \ (B \vee \bar{p}) \ \& R$$

where A, B, and R are free of p. Then F is inconsistent if and only if $A \ \& R$ and $B \ \& R$ are both inconsistent.

JUSTIFICATION OF RULE III*. For[1] $p = 0$, $F = A \& R$; for $p = 1$, $F = B \ \& R$.

The forms of Rule III are interchangeable; although theoretically they are equivalent, in actual applications each has certain desirable features. We used Rule III* because of the fact that Rule III can easily increase the number and the lengths of the clauses in the expression rather quickly after several applications. This is prohibitive in a computer if ones fast access storage is limited. Also, it was observed that after performing Rule III, many duplicated and thus redundant clauses were present. Some success was obtained by causing the machine to systematically eliminate the redundancy; but the problem of total length increasing rapidly still remained when more complicated problems were attempted. Also use of Rule III can seldom yield new one-literal clauses, whereas use of Rule III* often will.

In programming Rule III*, we used auxiliary tape storage. The rest of the testing for consistency is carried out using only fast access storage. When the "Splitting Rule" is used one of the two formulas resulting is placed on tape. Tape memory records are organized in the cafeterial stack-of-plates scheme: the last record written is the first to be read.

In the program written for the IBM 704, the matrix and conjunction of quantifier-free lines are coded into cross-referenced associated (or linked) memory tables by the QFL-Generator and then analyzed by the Processor. In particular, the QFL-Generator is programmed to read in the matrix M in suitably coded Polish (i.e., "parenthesis-free") form. The conversion to a quantifier-free matrix in conjunctive normal form requires, of course, a certain amount of pencil work on the formula, which could have been done by the computer. In doing this, we departed from [1], by not using prenex normal form. The steps are:

(1) Write all truth-functional connectives in terms of $\sim$, &, $\vee$.

(2) Move all $\sim$'s inward successively (using de Morgan laws) until they either are cancelled (with another $\sim$) or acting on an atomic formula.

(3) Now, replace existential quantifiers by function symbols (cf. [1], p. 205), drop universal quantifiers, and place in conjunctive normal form. A simple one-to-one assembler was written to perform the final translation of the matrix M into octal numbers.

It will be recalled that the generation of quantifier-free lines is accomplished by successive substitutions of "constants" for the variables in the matrix M. In the program

† The research reported in this document has been sponsored by the Mathematical Sciences Directorate, Air Force Office of Scientific Research, under Contract No. AF 49(638)-777.

[1] As in [1], 1 stands for "truth", and 0 for "falsehood".

the constants are represented by the successive positive integers.

For a matrix containing n individual variables, the n-tuples of positive integers are generated in a sequence of increasing norm such that all n-tuples with a given norm are in decreasing n-ary numerical order. Here we define the norm of $(j_1, \cdots, j_n) = j_1 + \cdots + j_n = \|j_i\|$. Other norms could have been used. For example, Gilmore [2] takes for $\|j_i\|$ the maximum of $j_i, \cdots, j_n$. In [1] a more complicated norm is indicated.

Substitutions of successive n-tuples into the matrix cause new constants to appear in the matrix. The program numbers constants in their order of appearance. Thus, the constants are ordered by the program in a manner depending upon the input data. By rearranging the clauses of a formula a different order would in general be created. In some cases, whether or not the program could actually prove the validity of a given formula (without running out of fast access storage) depended on how one shuffled the punched-data deck before reading it into the assembler! Thus, the variation in ordering of constants did affect by a factor of 10 (from 500 to 5000) the number of lines needed to prove the validity of:

$$(e)(Ed)(x)(y)[S(x, y, d) \rightarrow T(x, y, e)]$$
$$\rightarrow (e)(x)(Ed)(y)[S(x, y, d) \rightarrow T(x, y, e)]$$

(This valid formula may be thought of as asserting that uniform continuity implies continuity if we set:

$$S(x, y, d) \leftrightarrow |x - y| < d$$
$$T(x, y, e) \leftrightarrow |f(x) - f(y)| < e.)$$

In storing the quantifier-free lines, two tables are used. The first, called the *conjunction table*, is a literal image of the quantifier-free lines in which one machine word corresponds to one literal, i.e., to p or $\sim p$ where p is an atomic formula. The lines in the second, or *formula table* are themselves heads of two chain lists giving the occurrences of p and $\sim p$ respectively in the conjunction table. In addition, included for formula p in the formula table are counts of the number of clauses in which p and $\sim p$ occur and total number of all literals in these clauses; the formula table is itself a two-way linked list. A third short list of those literals is kept in which are entered all formulas to which the one-lateral clause and affirmative-negative rules apply; this is called the *ready list*. If the program tries to enter p and $\sim p$ into the ready list, an inconsistency has been found; the machine stops.

The totality of the processing rules requires only two basic operations: a subroutine to *delete* the occurrences of a literal p or $\sim p$ from the quantifier-free lines, and a routine to *eliminate* from them all the clauses in which p or $\sim p$ occur.

We may observe that only the deletion program can create new one-literal clauses, and likewise applications of the affirmative-negative rule can come only from the elimination program.

The machine thus performs the one literal-clause and affirmative-negative rules as directed by the ready list until the ready list is empty. It is possible that the choice of p to be eliminated first is quite critical in determining the length of computation required to reach a conclusion: a program to choose p is used, but no tests were made to vary this segment of the program beyond a random selection, namely the first entry in the formula table. To perform Rule III*, one saves the appropriate tables with some added reference information in a tape record, then performs an elimination on $\sim p$ and a deletion on p. At a consequent discovery of consistency, one must generate more quantifier free lines; the QFL-generator is recalled. Otherwise, at finding an inconsistency, the machine must check to see if there are any records on the Rule III* tape: if none, the quantifier-free lines were inconsistent; otherwise, it reads in the last record.

If one uses Rule III (which we did in an early version of our program), an entirely different code is needed. The problem is precisely that of mechanizing the application of the distributive law.

Results Obtained in Running the Program

At the time the programming of the algorithm was undertaken, we hoped that some mathematically meaningful and, perhaps nontrivial, theorems could be proved. The actual achievements in this direction were somewhat disappointing. However, the program performed as well as expected on the simple predicate calculus formulas offered as fare for a previous proof procedure program. (See Gilmore [1].) In particular, the well-formed formula

$$(Ex)(Ey)(z)\{F(x, y) \rightarrow (F(y, z) \ \& F(z, z)))$$
$$\& ((F(x, y) \ \& G(x, y)) \rightarrow (G(x, z) \ \& G(z, z)))\}$$

which was beyond the scope of Gilmore's program was proved in under two minutes with the present program. Gilmore's program was halted at the end of 7 "substitutions", (quantifier-free lines) after an elapsed period of about 21 minutes. It was necessary for the present program to generate approximately 60 quantifier-free lines before the inconsistency appeared.[2] Indeed, the "uniform continuity implies continuity" example mentioned above required over 500 quantifier-free lines to be generated and was shown to be valid in just over two minutes. This was accomplished by nearly filling the machine to capacity with generated quantifier-free lines (2000 lines in this case) before applying any of the rules of reduction.

Rather than describe further successes of the program, it will be instructive to consider in detail a theorem that the program was incapable of proving and to examine the cause for this. This particular example is one the authors originally had hoped the program could prove, an elementary group theory problem. In essence, it is to show that in a group a left inverse is also a right inverse.

[2] In [1], a hand calculation of this example using the present scheme showed inconsistency at 25 quantifier-free lines. The discrepancy is due to a different rule for generation of constants.

It is, in fact, quite easy to follow the behavior of the proof procedure on this particular example as it parallels the usual approach to the problem. The problem may be formulated as follows:

Axioms: 1. $e \cdot x = x$

2. $I(x) \cdot x = e$

3. $(x \cdot y) \cdot z = w \Rightarrow x \cdot (y \cdot z) = w$

4. $x \cdot (y \cdot z) = w \Rightarrow (x \cdot y) \cdot z = w$

Conclusion: $x \cdot I(x) = e$

The letter e is interpreted as the identity element and the function I as the inverse function. The associative law has been split into two clauses for convenience.

A proof is as follows:

1. $I(I(x)) \cdot I(x) = e$ by Axiom 2
2. $e \cdot x = x$ by Axiom 1
3. $I(x) \cdot x = e$ by Axiom 2
4. $I(I(x)) \cdot e = x$ by Axiom 3, taking $(I(I(x)), I(x), x)$ for (x, y, z)
5. $e \cdot I(x) = I(x)$ by Axiom 1
6. $I(I(x)) \cdot I(x) = e$ by Axiom 2
7. $I(I(x)) \cdot e = x$ step 4
8. $x \cdot I(x) = e$ by Axiom 4, taking $(I(I(x)), e, I(x))$ for (x, y, z)

Step 8 is the desired result.

To formalize this proof would require adjoining axioms of equality. To avoid this, one can introduce the predicate of three arguments $P(x, y, z)$, interpreted as $x \cdot y = z$. The theorem reformulated becomes:

Axioms: 1. $P(e, x, x)$

2. $P(I(x), x, e)$

3. $\sim P(x, y, u) \vee \sim P(u, z, w) \vee \sim P(y, z, v) \vee P(x, v, w)$

4. $\sim P(y, z, v) \vee \sim P(x, v, w) \vee \sim P(x, y, u) \vee P(u, z, w)$

Conclusion: $P(x, I(x), e)$.

The theorem to be proved valid is the implication of the conjunction of the four axioms with the conclusion, the universal quantifiers appearing outside the matrix.

To complete the preparation of the well-formed formula for encoding for the assembler, it is necessary to negate the conclusion. (cf. [1], p. 204.)

The single existential quantifier has no dependence on the universal quantifiers, hence leads to the constant function s when this existential quantifier is replaced by a function symbol. (cf. [1], p. 205.]

The conclusion then becomes

$$\sim P(s, I(s), e).$$

The conjunction of this with the four axioms gives the desired form.

As seen from the proof previously noted the quantifier-free clauses needed to produce the inconsistency are

1. $P(I(I(s)), I(s), e)$
2. $P(e, s, s)$
3. $P(I(s), s, e)$
4. $\sim P(I(I(s)), I(s), e) \vee \sim P(e, s, s) \vee \sim P(I(s), s, e) \vee P(I(I(s)), e, s)$
5. $P(e, I(s), I(s))$
6. $\sim P(e, I(s), I(s)) \vee \sim P(I(I(s)), I(s), e) \vee \sim P(I(I(s), e, s) \vee P(s, I(s), e)$
7. $\sim P(s, I(s), e)$

(It is quite clear in this case that successive applications of the one-literal clause rule reducing this set to

$$P(s, I(s), e) \,\&\, \sim P(s, (I(s), e).)$$

The question to be considered is: how many quantifier-free lines must be generated by the present program to realize these required lines? The constants are generated in the following:

1. e
2. s
3. $I(s)$
4. $I(e)$
5. $I(I(s))$

etc.

(The constants are identified directly with their index e.g. the 6-tuple (1, 1, 1, 1, 1, 1) represents (e, e, e, e, e, e). As this is the first substitution, the program assigns in order, reading the well-formed formula backwards and from the inside out for nesting functions: e, s, $I(s)$, $I(e)$, $I(I(s))$. The $I(I(s))$ appears when x is assigned $I(s)$, no new entries occurring until this time. Note that there are 6 free variables (u, v, w, x, y, z) in the matrix).

The program generates the needed n-tuples by producing all possible n-tuples of integers whose sum N of entries is fixed, $N = n, n + 1, \cdots$. Thus it is only necessary to consider the n-tuple which has the maximum sum of entries. In this case, the substitution $u = s$, $v = I(s)$, $w = e$, $x = I(I(s))$, $y = e$, $z = I(s)$ (required for axiom 4 to produce the clause 6 in the "proof" above in a quantifier-free line) gives the n-tuple with maximum sum. The n-tuple is seen to be (2, 3, 1, 5, 1, 3), the sum equals 15. The combinatorial expression $\binom{N}{n}$ gives the total number of n-tuples of positive integers whose sum is less than or equal[3] to N.

[3] To see this, consider a sequence of $N+1$ ones. Flag n of these. The flag is to be interpreted "sum all 1's, including the flagged 1, to the next flag and consider this sum as an entry in the n-tuple". Placing an unflagged 1 on the extreme left, leaving it fixed, consider the possible permutations of all other symbols. The different sequences total $\binom{N}{n}$ and, regarding the set of 1's starting with the last flagged 1 as overflow, this is seen to represent precisely the desired n-tuples.

It is seen that to prove this theorem at least $\binom{14}{6} = 3003$ lines must be generated and that the inconsistency will be found on or before $\binom{15}{6} = 5005$ lines have been generated.[4]

The present program generated approximately 1300 quantifier-free lines. This number of quantifier-free lines was accomplished holding all major tables simultaneously in core memory, limited to 32,768 "words". (This was done to insure a reasonable time factor for any problem, within possible scope of the program. For this reason also, the entire program was coded in SAP with many time-saving devices employed.)

The authors believe that a reprogramming to make use of tape storage of tables might realize a factor of 4 for the total number of quantifier-free lines attainable before running time became prohibitive. This would be just sufficient for this problem. That realizing this extra capacity is really uninteresting is seen by noting that if the conclusion was placed before the axioms, altering the validity of the matrix not at all, the element $I(c)$ would be generated before $I(s)$ and the needed n-tuple would sum to 16. Then $\binom{16}{6} = 8008$ becomes the upper bound, beyond the capacity of the projected program. Other formulations of the same problem result in quite unapproachable figures for the number of quantifier-free lines needed. (For another example illustrating the same situation, see Prawitz [3].)

The existing program allows one to think of working with a capacity of 1000 or 2000 quantifier-free lines instead of a capacity of 10 or 20, the previous limit. The time required to generate additional quantifier-free lines is independent of the number of quantifier-free lines already existing. Against this linear growth of number of quantifier-free lines generated, there is, in a meaningful sense, an extreme nonlinear growth in the number of quantifier-free lines to be considered with increasingly more "difficult" problems. This is true of simple enumeration schemes of the nature considered here. It seems that the most fruitful future results will come from reducing the number of quantifier-free lines that need be considered, by excluding, in some sense, "irrelevant" quantifier-free lines. Some investigation in this area has already been done (see Prawitz [3]).

[4] If the rule for generating n-tuples had been, for each m, to generate all n-tuples possible such that each entry assumes a positive integral value less than or equal to m, it is clear that at least 4^6 (= 4096) quantifier-free lines would be needed and 5^6 (= 15625) lines would suffice to guarantee a solution. If no more information were available, one sees an intuitive advantage, in this case, for using the previous method. In general, the authors see no preference for either method, in contrast to some previous suggestions that the latter method seemed intuitively better.

REFERENCES

1. Davis, Martin, and Putman, Hilary. A computing procedure for quantification theory. *J. ACM 7* (1960), 201–215.
2. Gilmore, P. C. A proof method for quantification theory. *IBM J. Res. Dev. 4* (1960), 28–35.
3. Prawitz, Dag. An improved proof procedure. *Theoria 26*, 2 (1960), 102–139.

Nonlinear Regression and the Solution of Simultaneous Equations

Robert M. Baer

University of California, Berkeley

If one has a set of observables $(z_1, \cdots, z_m)$ which are bound in a relation with certain parameters $(a_1, \cdots, a_n)$ by an equation $\zeta(z_1, \cdots ; a_1, \cdots) = 0$, one frequently has the problem of determining a set of values of the a_i which minimizes the sum of squares of differences between observed and calculated values of a distinguished observable, say z_m. If the solution of the above equation for z_m,

$$z_m = \eta(z_1, \cdots ; a_1, \cdots)$$

gives rise to a function η which is nonlinear in the a_i, then one may rely on a version of Gaussian regression [1, 2] for an iteration scheme that converges to a minimizing set of values. It is shown here that this same minimization technique may be used for the solution of simultaneous (not necessarily linear) equations.

Modifications of the technique, while necessary for convergence in some problems, are extraneous to the argument and shall be ignored. The Gaussian procedure may then be defined as follows.

If $a_i(h)$ denotes the values of the parameters at the hth iteration, then $a_i(h + 1) = a_i(h) + \epsilon_i(h)$, where the corrections $\epsilon_i(h)$ are the solution to the set of equations

$$\sum_j A_{ij}\epsilon_j + B_i = 0 \qquad (i = 1, \cdots, n) \tag{1}$$

where

$$A_{ij} = \sum_k \frac{\partial \eta_k}{\partial a_i}\frac{\partial \eta_k}{\partial a_j} \tag{2}$$

and

$$B_i = \sum_k [\eta_k - z_m(k)]\frac{\partial \eta_k}{\partial a_i}. \tag{3}$$