[illegible]

W. Bledsoe

Proceedings

Fourth Workshop on Automated Deduction

Austin, Texas
February 1-3, 1979

Proceedings of the Fourth Workshop on Automated Deduction

February 1-3, 1979
Thompson Conference Center
Austin, Texas

Proceedings Editor:

William H. Joyner, Jr.

Foreword

The following contributed papers were selected by the program committee from abstracts submitted by the authors to the Fourth Workshop on Automated Deduction. The final papers were prepared by the authors in camera ready form and have not been refereed. Some of the papers are preliminary reports of continuing research. Many of them will appear in more polished and complete form in scientific journals.

William H. Joyner, Jr.
Proceedings Editor

Fourth Workshop on Automated Deduction

February 1-3, 1979

Table of Contents

AVERAGE CASE COMPLEXITY OF THE SATISFIABILITY PROBLEM

Allen Goldberg
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, N. Y.

Abstract

An average case analysis of the Davis-Putnam procedure is performed with a variety of underlying distributions assumed. For each of these distributions a polynomial bound is obtained on the expected time complexity of the algorithm. When a uniform distribution is placed on the problem instances, the expected time of the Davis-Putnam procedure is shown to be $O(rn^2)$, where n is the number of clauses in the given set and r is the number of distinct atoms in the set. It is shown how to obtain resolution refutations efficiently from a Davis-Putnam refutation.

1. Introduction

In [1] the satisfiability problem for the propositional calculus is shown to be NP-complete. Since then, many other important combinatorial problems have also been shown to be in this class [5]. A fundamental property of the class of NP-complete problems is that if one member of the class possesses a polynomial time algorithm then they all do. Since these problems have been well studied, and as yet no polynomial time algorithm has been found for any of them, they all are considered computationally intractable. Like the satisfiability problem, many NP-complete problems have important practical applications and so algorithms that solve the problems as efficiently as possible are of substantial value.

In a practical context, an algorithm's profile consisting of both a worst case analysis and an average case analysis based on the actual distribution of problem instances, would be most informative for evaluating the performance of the algorithm. Unfortunately, average case results are difficult to obtain because properties of the actual distribution may be unknown, and the technical problems encountered in performing such an analysis can sometimes be formidable. As a result a worst case analysis yielding an upper bound on the expected time complexity of an algorithm is often the best achieved. The upper bound provided by the worst case analysis may be a crude one when the worst case instances are relatively few in number. A prime example of this phenomenon is the simplex method for solving linear programming problems [6]; although a worst case analysis yields an exponential bound [7], acceptably good expected time performance has in fact been observed in practice.

It appears this is the case for the satisfiability problem as well. The NP-completeness of the satisfiability problem suggests that any algorithm to solve the problem will have exponential worst case complexity. In this paper we present an average case analysis of the Davis-Putnam procedure [4], an algorithm to solve the satisfiability problem, that indicates that this NP-complete problem is not intractable. A family of results is obtained that give polynomial bounds for various distributions of problem instances. In particular, if a uniform distribution on the sample space of problem instances is assumed then the average time complexity is shown to be $O(rn^2)$, where r is the number of atoms appearing in the given set of clauses and n is the number of clauses in the given set. In addition, we show how resolution refutations may be efficiently generated from the Davis-Putnam procedure (DPP).

2. Preliminaries

An *atom* is an element of the set $A = \{x_1,\ldots,x_r\}$; a *literal* is an element of the set $L = \{x_1,\ldots,x_r,\bar{x}_1,\ldots,\bar{x}_r\}$. A literal is said to be *negative* if it has a bar, *positive* otherwise. If L is the literal x_i ($\bar{x}_i$) its *complement*, $\bar{L}$, is $\bar{x}_i$ (x_i). A *clause* is a subset of L in which a literal and its complement do not appear. (We choose by this definition to disallow tautological clauses.) A set of clauses $S = \{C_1,\ldots,C_n\}$, $C_i \subset L$ is *satisfiable*, if there is a clause $M \subset L$ such that $M \cap C_i \neq \emptyset$ for $i = 1,\ldots,n$.

The Davis-Putnam procedure determines whether or not a set S of clauses is satisfiable. The procedure consists of three rules:

I. (pure literal rule) If for some literal L, $\{\bar{L}\} \cap C_i = \emptyset$ $(i=1,\ldots,n)$ then letting $S_1 = \{C | C \in S \text{ and } L \notin C\}$, S_1 is satisfiable iff S is.

II. (unit clause rule) If for some clause $C \in S$, $C = \{L\}$ then letting $S_1 = \{C - \{\bar{L}\} \mid C \in S \text{ and } L \notin C\}$, S_1 is satisfiable iff S is.

III. (splitting rule) Choose a literal L. Let $S_1 = \{C - \{\bar{L}\} \mid C \in S \text{ and } L \notin C\}$; $S_2 = \{C - \{L\} | C \in S \text{ and } \bar{L} \notin C\}$. S is satisfiable iff S_1 or S_2 is satisfiable.

DDP can be described as a recursive procedure operating on a set of clauses S:

```
procedure DPP(S);
1. if $S = \emptyset$ then return satisfiable;
            end if;
2. if $\emptyset \in S$ then return unsatisfiable;
            end if;
3. if rule I applies to S then
      if DPP($S_1$)=satisfiable
            then return satisfiable;
         else return unsatisfiable;
            end if;
   end if;
4. if rule II applies to S then
      if DPP($S_1$)=satisfiable
            then return satisfiable;
         else return unsatisfiable;
            end if;
5. else (apply rule III) if DPP($S_1$)
            = satisfiable
      or DPP($S_2$)=satisfiable
            then return satisfiable;
         else return unsatisfiable;
            end if;
   end if;
   end DPP;
```

Termination of DPP is guaranteed by observing that sets S_1 and S_2 have fewer distinct literals than S. Our results show that rapid termination is expected because S_1 and S_2 have many fewer clauses than S.

In performing an average case analysis it is first necessary to choose parameters that define a subset of instances of the problem, and then define a probability distribution function on the sample space of those instances. For example, when analyzing sorting algorithms the parameter chosen is the number of keys, n, to sort. The uniform distribution, which assigns equal probability to the n! possible permutations of the keys, is usually chosen. As Rabin [9] has pointed out, an inherent problem of average case analyses is that the probability distribution assumed by the analysis might not agree with the actual relative frequency of instances. For example, average case analyses of bubble sort and quicksort [8] which assume a uniform distribution yield $O(n^2)$ and $O(n \log n)$ expected time, respectively. However, in a sorting application in which keys are in nearly sorted order initially bubble sort will show better performance than quicksort despite these average case bounds. This problem can be overcome by proving results for many distributions, as we have done. The analysis also aids in identifying those cases in which the algorithm performs poorly and so helps determine the validity of the result in a particular context.

3. The Analysis

Instances of the satisfiability problem are parameterized by the number of clauses and the number of distinct atoms in the clauses. Let S be a set of clauses; $S \in K(n,r)$ if S contains n clauses, composed of literals chosen from a set of r atoms, $\{x_1,\ldots,x_r\}$. Clauses are not assumed distinct. Probability distributions are placed on $K(n,r)$ by specifying a distribution function for $K(1,r)$ and extending it to the n-fold product space of $K(1,r)$, $K(n,r)$. Let $p_{i,r}(q_{i,r})$, $i=1,\ldots,r$ be the probability that the positive (negative) literal x_i $(\bar{x}_i)$ occurs in a clause in $K(1,r)$. For the uniform distribution $p_{i,r} = q_{i,r} = 1/3$, $i=1,\ldots,r$, since for any clause in $K(1,r)$ the three possibilities of an atom not appearing, appearing positively or appearing negatively are equally likely.

<u>Lemma 1</u>. Let $S \in K(n,r)$. Then for some constant c, cnr bounds the time needed to perform one step of DPP. (A *step* of the DPP entails testing if S is empty; if $\emptyset \in S$; if rules I or II are applicable; and the application of the appropriate rule.)

<u>Proof</u>. nr bounds the length of S. Scanning S once the necessary tests are made; in another pass the appropriate rule is applied. □

It might be thought that in order to obtain a good bound on the complexity of the DPP it would have to be shown that the unit rule and the pure literal rule are applied frequently; it is the splitting rule which introduces the possibility of exponentially long computations. We analyze a procedure DPP', which only performs the splitting rule. DPP' is the DPP with statements 3 and 4 removed. A step of either algorithm takes $O(nr)$ time. When executing the DPP the possibility exists that only the satisfiability of S_1 and not S_2 has to be determined. Thus the complexity of DPP' bounds the complexity of DPP. Let $T(n,r)$ be the average case running time of DPP' with the uniform distribution on $K(n,r)$.

<u>Theorem 1</u>. $T(n,r) \le crn^2$.

<u>Proof</u>. We develop a recurrence relation for $T(n,r)$ based on the number of clauses that occur in the sets S_1 and S_2 that result from the application of the splitting rule on an arbitrarily selected atom. The set $S_1 = \{C - \{\bar{L}\} \mid C \in S \text{ and } L \notin C\}$ has size $n - i$, where i is the number of clauses in S in which L occurs. Similarly,

S_2 has n-j clauses, where j is the number of clauses containing $\bar{L}$. Since the probability that L occurs in a clause is 1/3, the probability that i out of n clauses contain the literal L is the Bernoulli probability, $\binom{n}{i}(\frac{1}{3})^i(\frac{2}{3})^{n-i}$. This holds for the uniform distribution regardless of whether L is positive of negative. Hence with the stated probability, S_1 and S_2 have n-i clauses. Since the literal L and its negation are removed in forming S_1 and S_2, these sets have at most r-1 distinct atoms. The recurrence for T(n,r) is

$$T(n,r) = \begin{cases} cnr + 2\sum_{i=1}^{n}\binom{n}{i}(\frac{1}{3})^i(\frac{2}{3})^{n-i}T(n-i,r-1), & n,r \geq 1 \\ 0, & n = 0 \text{ or } r = 0 \end{cases}$$

The first term is the time needed to perform one step of the DPP. The second term is the expected time needed to determine the satisfiability of S_1 and S_2 by the recursive calls to the DPP.

To bound T(n,r), note that T(n,r) is increasing in r. Hence,

$$T(n,r) \leq cnr + 2\sum_{i=1}^{n}\binom{n}{i}(\tfrac{1}{3})^i(\tfrac{2}{3})^{n-i}T(n-i,r)$$

$$= crT'(n) \text{ where}$$

$$T'(n) = n + 2\sum_{i=1}^{n}\binom{n}{i}(\tfrac{1}{3})^i(\tfrac{2}{3})^{n-i}T'(n-i).$$

By a change in index variable

$$T'(n) = n + 2\sum_{i=0}^{n-1}\binom{n}{i}(\tfrac{2}{3})^i(\tfrac{1}{3})^{n-i}T'(i).$$

We show $T'(n) \leq n^2$ by induction on n. Computation verifies that $T'(n) \leq n^2$, for n = 1,...,12 as base cases. For the inductive step, applying the induction hypothesis,

$$T'(n) \leq n+2\sum_{i=0}^{n-1}\binom{n}{i}(\tfrac{2}{3})^i(\tfrac{1}{3})^{n-i}i^2.$$

Using the calculus of finite differences (see [2]), it can be shown that

$$\sum_{i=0}^{n}\binom{n}{i}(\tfrac{2}{3})^i(\tfrac{1}{3})^{n-i}i^2 = \tfrac{2}{3}n + \tfrac{4}{9}n(n-1);$$

and so,

$$T'(n) \leq n+2\left(\tfrac{2}{3}n + \tfrac{4}{9}n(n-1)\right) = \tfrac{13}{9}n + \tfrac{8}{9}n^2$$

which for $n \geq 13$ is less than n^2. This completes the induction. The theorem follows. □

Suppose an arbitrary distribution has been placed on K(l,r). Let $p = \inf_{i,r} p_{i,r}$; $q = \inf_{i,r} q_{i,r}$. If both p and q are not zero then as the number of literals in L grows, the probability that any literal appears in a clause is bounded away from zero. Under these conditions we can bound the average time complexity of the DPP by

$$T_{p,q}(n,r) = cnr + \sum_{i=1}^{n}\binom{n}{i}p^i(1-p)^{n-i}T(n-i) + \sum_{i=1}^{n}\binom{n}{i}q^i(1-p)^{n-i}T(n-i).$$

A proof similar to the one given yields a polynomial bound for $T_{p,q}(n,r)$ for fixed values of p and q. If we set $m = 1 - \min(p,q)$ then $T_{p,q}(n,r) \leq crn^k$, where $k = \left\lceil -\frac{1}{\log_2 m}\right\rceil$

Since the DPP is not a complex procedure, the constant c which appears in our bounds is not large. Thus the algorithm will perform well on small problems as well as having good asymptotic behavior. Actual experience with the algorithm is in agreement with these results [3].

4. Resolution

In this section we show how the DPP can be modified to generate resolution refutations for unsatisfiable sets of clauses with only a constant factor increase in the DPP's running time. The recursive structure of the procedure will be used to recursively generate the refutation. Suppose the DPP, given an unsatisfiable set S, splits S, in accordance with rule III, on literal L into sets S_1 and S_2. Since S_1 and S_2 are each unsatisfiable, inductively the DPP can generate resolution refutations for these sets. Restoring the literal $\bar{L}$ (L) into the appropriate clauses in the derivation, refutations from S_1 (S_2) yield resolution derivations of $\bar{L}$ and L from S. Resolving L and $\bar{L}$ derives the empty clause and so completes the construction of the refutation from S. The complete algorithm is described as the procedure RES(S,deriv).

```
   procedure RES(S,deriv);
1. if S = ∅ then deriv = ( );
      return satisfiable; end if;
2. if ∅ ∈ S then deriv = (∅);
      return unsatisfiable; end if;
3. if rule I applies to S then
         if RES($S_1$,deriv) = satisfiable
            then return satisfiable;
               else return unsatisfiable;
               end if;
   end if;
4. if rule II applies to S then
      if RES($S_1$,$deriv_1$)=satisfiable
               then return satisfiable;
         else deriv=(restore $\bar{L}$ to
              clauses of $deriv_1$ (yielding
              a derivation of $\bar{L}$) append
              a resolution with L);
              return unsatisfiable;
      end if;
5. else (rule III applies) if
      RES($S_1$,$deriv_1$)=satisfiable or
      RES($S_2$,$deriv_2$)=satisfiable then
      deriv = ( ); return satisfiable;
      else deriv=(restore $\bar{L}$ to clauses of
           of $deriv_1$, restore L to clauses
           of $deriv_2$, append them together,
           resolve L with $\bar{L}$);
           return unsatisfiable; end if;
   end if;
   end RES;
```

Theorem 2. There is an algorithm that generates resolution refutations of an unsatisfiable set of clauses, S, whose running time is within a constant factor of the running time of the DPP.

Proof. The above procedure may not achieve the bound specified by the theorem because of the necessity of scanning the derivation repeatedly to restore deleted literals. This problem is easily overcome by initially inserting the complete clause, without any of its literals deleted into the derivation. Hence, a derivation is built from constituent derivations in step 5 by appending the two together and adding an additional resolution between the derived clauses. The refutation constructed in this way will be identical to the one generated by RES. □

Corollary. The average case complexity bounds of Theorem 1 apply to resolution as well.

It is interesting to note that another simple modification of the DPP yields a procedure in which refutation graphs [10] are generated with only a constant factor increase in running time. Shostak proved the existence of refutation graphs, for unsatisfiable sets of clauses, essentially by demonstrating such a DPP based algorithm. He then shows how more restricted resolution refutations, such as t-linear refutations, can be efficiently obtained by walking the refutation graph. Thus, the time bound of Theorem 1 applies to t-linear derivations as well.

References

1. Cook, S. A., "The complexity of theorem proving procedures", Proc. Third ACM Symp. on Theory of Computing, May 3-5, 1971, 151-158.
2. Davis, H. T., "The Summation of Series", The Principia Press, San Antonio, Texas, 1962.
3. Davis, M., Logemann, G., and Loveland, D., "A machine program for theorem proving", Comm. ACM 5 (1962), 394-397.
4. Davis, M., and Putnam, H., "A computing procedure for quantification theory", J. ACM 7 (1960), 201-215.
5. Karp, R. M., "Reducibilities among combinatorial problems", in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher (eds.), Plenum Press, New York, 1972, 85-104.
6. Karp, R. M., "The probabilistic analysis of some combinatorial search algorithms", in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub (ed.), Academic Press, New York, 1976, 1-19.
7. Klee, V., and Minty, G. J., "How good is the simplex algorithm?", Mathematical Note No. 643, Boeing Scientific Research Laboratories (1970).
8. Knuth, D. E., "The Art of Computer Programming, Volume 3: Sorting and Searching", Addison-Wesley, Reading, MA, 1973.
9. Rabin, M. O., "Probabilistic algorithms", in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub (ed.), Academic Press, New York, 1976, 21-40.
10. Shostak, R. E., "Refutation graphs", Artifical Intelligence 7 (1976) 51-64.

GENERATION AND VERIFICATION OF FINITE MODELS AND COUNTEREXAMPLES USING AN AUTOMATED THEOREM PROVER ANSWERING TWO OPEN QUESTIONS*

by

Steve Winker

Northern Illinois University

ABSTRACT

Two open questions in ternary Boolean algebras [1, 2,6] were answered with the aid of an existing automated theorem-proving program without recourse to any additional programming [6]. The new automated theorem-proving techniques developed in answering the open questions are presented in this paper; essentially the existing theorem prover is used in a nonstandard way to seek and verify small finite models and counterexamples for a first order axiom system. Exhibiting a model of an axiom system proves it consistent; this facility complements traditional theorem-proving methods which can only prove inconsistency.

1. INTRODUCTION

The solution of two open questions in mathematics with the aid of an existing automated theorem-proving program exhibits progress toward the long standing goal that automated theorem provers be of aid to mathematicians doing research. Traditionally automated theorem provers have only focused on the attempt to prove a theorem true, while neglecting the search for a counterexample. Techniques (within the context of our existing theorem-proving program) are described in this paper for the construction of small finite counterexamples in particular and models in general. The user must make some decisions about what sort of model to seek, but much of the work involved in searching for models can be done automatically. To repeat, no reprogramming is required to further a given model search or to attack a new problem or new set of axioms.

The two open questions answered concern independence of axioms in an axiomatization of "ternary Boolean algebras" by A. A. Grau [2]; the results are described fully in Winker and Wos [6]. It is the purpose of this paper to present the new model-finding techniques in detail. No new programming was required to implement the model-finding techniques; rather, the use of certain special clauses (see Appendix II) enabled our existing theorem prover to do the desired operations. The techniques will be discussed in the context of the question to which they were first applied: the independence of axiom 2 of the Grau ternary Boolean algebra axioms (Appendix II) from the remaining axioms.

In the interest of clarity the automatic verification of a single completely defined model will be discussed first; the model search techniques, which are more involved, are deferred until section 3.

2. VERIFICATION OF A FINITE MODEL

This discussion will begin by considering theoretical aspects of model verification before proceeding with a detailed description of "model verification runs". First consider how a model may be specified. A finite model for Grau axioms† 1, 3, 4, and 5, and violating axiom 2, for example, may be specified by giving a set of elements and full tables of values for the functions F and G on those elements. One must then verify that

(1) these values are consistent with axioms 1, 3, 4, and 5, and

(2) that axiom 2 is violated.

Axiom 2 is an equality. In order for it to be violated, the two sides must be unequal for some values (X,Y) and for this it is required that two elements of the model be demonstrably unequal. This is not a trivial matter to arrange. In particular, A¬=B cannot be derived from a set of equalities in A and B; equality of all elements is consistent with any set of positive equalities. Instead of deriving A¬=B one must prove A¬=B to be consistent with the equalities. Consistency cannot be proven by the refutation techniques of traditional automated theorem proving, and so new techniques are needed.

The method of "complete sets of reductions" [3] enables one to prove consistency of, for example, A¬=B with certain sets of equalities, and thus provides a starting point for the new techniques. Essentially, if a set of demodulators [8] form a complete set of reductions and do not demodulate A and B to the same term, then A¬=B plus those demodulators form a satisfiable set of clauses. The procedures given in [3] for generating and

*This work was supported in part by NSF grant MCS77-02703, and in part was performed under the auspices of the U.S. Department of Energy while in residency at Argonne National Laboratory.

†Grau Ternary Boolean Algebra Axioms (see also Appendix I):

```
1:  F(V,W,F(X,Y,Z))=F(F(V,W,X),Y,F(V,W,Z))
2:  F(Y,X,X)=X
3:  F(X,Y,G(Y))=X
4:  F(X,X,Y)=X
5:  F(G(Y),Y,X)=X
```

testing complete sets of reductions are easily performed using the standard paramodulation [7] and demodulation [8] features of our theorem-proving program.

Unfortunately not every system of equalities yields a finite complete set of reductions (by undecidability of the word problem) and even a finite set may be unmanageably large. Indeed in the ternary Boolean algebra problems under consideration, application of the standard procedure for generation of a complete set of reductions from the axioms [3] yielded a set of equalities which exceeded time and memory limitations. This difficulty was overcome as follows: set up another, in some sense simpler set of equalities to define a finite model, prove that the simpler equalities form a complete set of reductions, and finally prove that the original axioms are necessarily satisfied in the model so defined. Two questions then arise: first, how does one choose the simpler set of equalities; second, how does one then verify the axioms? Given a specific model, a simpler set of equalities could be obtained by removing complex equalities (e.g. axiom 1[†]) and adding simple ground equalities (e.g. those of Appendix IB) to fill in for the removed complex equalities in defining the function tables. Actually the models were found by the methods of section 3, which yield simple equalities anyway.

Verification of the axioms is done automatically in a "model verification run"; model verification runs will now be discussed in detail. To specify a model of an equational system, one must specify a set of elements (considered to be distinct) and functions on those elements corresponding to the functions of the system. The following requirements must be satisfied:

(1) Each function must be well-defined.

(2) Each function must be closed.

(3) Each axiom of the system must be satisfied in each instance.

A function on a finite set of elements may be specified by simply tabulating its values. The function tables are supplied to our program in the form of a set of "function defining equalities"; for an example see the model given in Appendix IB. The value of F(t1,...tn)), where t1,...tn are model elements, is then defined to be the result of demodulating F(t1,...tn), using the set of "function defining equalities" as demodulators. For example, in the model of Appendix IB, F(A,G(A),G(A)) receives the value G(A); G(G(G(A))) receives the value G(A); G(G(A)) (G applied to the element G(A)) receives the value G(G(A)) because G(G(A)) does not demodulate. In this way evaluation of functions of model elements is done using our existing demodulation routine. Observe that one equality may stand for several function table entries.

[†]See Appendix I

Closure may be tested by forming, for each function F and each possible n-tuple t1...tn of model elements, the term F(t1...tn), and then demodulating each such term. If only model elements are obtained, F is closed. Clauses for testing closure are given in Appendix IIB.

Well-definedness must hold for functions of a model: any function of given model elements must be given a unique value. Equivalently, demodulation of a term F(t1,...tn) where the arguments are model elements must yield a unique result no matter how the demodulation is done. The simplest way to guarantee a unique result ("unique termination") is to verify that the "function defining equalities" form a "complete set of reductions". This verification may be done by a paramodulation-and-demodulation procedure as described in [3]. An alternative method for verifying unique termination is outlined in Appendix V but has not been needed for the models examined so far. Note that even though the axiom system being modeled may not yield a complete set of reductions, the function defining equalities for a particular model of that system can still form a complete set of reductions.

The condition that each axiom is true in each instance is checked by forming each instance, demodulating both sides of the equality, and checking that the two sides become identical in each case. For example, substituting A/V, G(G(A))/W, G(A)/X, A/Y, G(G(A))/Z in Grau Axiom 1 yields, after demodulation, A=A, verifying axiom 1 in this instance. A method for generating all the instances is given in Appendix IIA. Our theorem prover demodulates each automatically, then tests each for subsumtion by X=X (equivalently, for identity of the two sides). Checking satisfaction of Axiom 1 formed the bulk of the work of verifying each Grau model; 3 to the 5th power = 243 instances must be checked for a three-element model. (The amount of checking can in some cases be reduced if needed by symmetry and other considerations; see Appendix III).

Note: If the function defining equalities form a complete set of reductions, no axiom included among those equalities (for example axioms 3-5 in the model of Appendix IB) needs to be checked for satisfaction in all instances. Proof: The check is trivial. Apply the demodulator s=t to the axiom instance su=tu giving tu=tu.

This "trivial check" is only valid when the axiom is one of the function defining equalities and the function defining equalities form a complete set of reductions. In checking axiom 1 in the model of Appendix IB, for example, axiom 1 must not be used as a demodulator, because axiom 1 is not one of the function defining equalities. As a simple example of an invalid "trivial check" consider a model with elements A and B and equalities J(J(J(X)))=X for all X, J(A)=B, and J(B)=A, with the latter two forming the complete set of reductions. J(J(J(X)))=X seems to be satisfied when the "trivial check" is applied; however when the latter two equalities are applied to J(J(J(A))), B is obtained, not A. In doing the "trivial check" the implicit assumption is made that demodulating using J(J(J(X)))=X gives the same result as demodulating using the other equalities; this is not true however because the three equalities together do not form a complete set of reductions.

3. SEARCHING FOR FINITE MODELS

The NIUTP automatic theorem prover [4,5,7,8] was used extensively in searching for the models we found as well as in verifying them. The model searches proceeded in three stages: preliminary paramodulation runs, partial-model runs, and final model-verification run.

Preliminary paramodulation runs can be used to derive consequences of the axioms being modeled, to suggest what equalities do not follow from the axioms, and to test the effect of adding various function-defining or other equalities to the axiom system. For example:

1) The equality F(X,Y,X)=X was derived by paramodulation from axioms 1, 3, 4, and 5 of Appendix IA, filling in part of the function table for F (in any model of axioms 1, 3, 4, and 5).

2) The failure to derive G(G(X))=X from the above system (axioms 1, 3, 4, and 5) suggested that models be sought in which G(G(A))¬=A for some A (see Section 4, Example of a Model Search).

3) Addition of the axiom F(X,Y,Z)=F(X,Z,Y) to the above system was rejected because paramodulation then derived G(G(X))=X (G(G(A))¬=A for some A was desired).

4) Addition of G(G(G(X)))=G(X) to the above system yielded no undesirable paramodulants; this system thus seemed promising and was studied further by means of "partial-model runs".

Partial-model runs are begun when a model has been partially specified: that is, when the set of model elements (e.g. (A, G(A), G(G(A)))) has been selected, and most but not all entries in the function tables have been filled in.

A partial-model run is similar to a model verification run. The similarities:

A set of model elements is input.

A set of function-defining equalities is input.

Closure, well-definedness, and satisfaction of the axioms in all instances are tested as in model verification.

The differences:

The function defining equalities do not completely define the functions. Rather, the values for some terms (called "undecided terms") are left unspecified.

The closure test will not indicate closure, but rather will yield a list of the undecided terms.

In a partial-model run, the check for satisfaction of the axioms in all instances may yield any or all of the following:

1) Equality of a model element to itself -- indicating satisfaction of an axiom in the particular instance tested.

2) Equality of two model elements -- indicating that the partially defined functions already do not satisfy the axiom tested.

3) Equality of an undecided term to a model element -- any such equality is input as a function defining equality in subsequent partial-model and model-verification runs.

4) More complex ground equalities involving undecided terms -- may be used to eliminate some possible values for the undecided terms.

Example of a partial-model run: Axioms 1, 3, 4, and 5 of Appendix I, plus the equality G(G(G(X)))=G(X), yield the first seven equalities in Appendix IB; these were used as defining equalities for a partial-model run. The model elements A, G(A), and G(G(A)) were also input. The undecided terms were F(A,G(A),G(A)), F(A,G(G(A)),G(G(A))), and F(G(G(A)),A,A). The check for satisfaction of axiom 1 in all instances yielded (among others) the equalities

```
F(A,G(G(A)),G(G(A)))=A
F(G(G(A)),A,A)=G(G(A))
F(F(A,G(A),G(A)),G(G(A)),A)=A
```

The first two were input as function defining equalities in subsequent runs. The last equality eliminates the possibility F(A,G(A),G(A))=G(G(A)) (as this and axiom 4 would demodulate the last equality to G(G(A))=A indicating violation of axiom 1). Each of the other two possibilities, F(A,G(A),G(A))=A and F(A,G(A),G(A))=G(A), led to a valid model. This ends the example.

Finally, when enough information has been gained from paramodulation and partial-model runs to specify a likely model completely, that model is verified in a model-verification run (as described in section 2).

Notes on searching for models:

1) When making a partial-model run, it appears desirable to instantiate axioms using only the elements which are expected to be in the final model. Instantiation using "undecided terms" would yield more axiom instances to check and more complicated ground equalities.

2) A partial-model run with too many undecided terms may yield an unmanageable quantity of complex ground equalities. In this case paramodulation runs might be used to find the consequences of proposed function-defining equalities.

When more function-defining equalities have been tested, found seemingly acceptable, and added to the partial model, partial-model runs may again be attempted.

3) a partial-model run may be considered "promising" if no equality between distinct model elements is derived. Each "promising" partial-model run in the author's brief experience has led to a valid model. However the author doubts that a "promising" partial-model run guarantees that there is a valid model: it might not be possible to complete the function tables consistently with the axioms being modeled.

4) It is possible in modeling certain axiom systems (e.g. semigroups, having associativity as the only axiom) to include all the axioms, plus the function defining equalities, in a complete set of reductions. In this case none of the axioms need to be tested for satisfaction in each instance (see the note at the end of Section 2) and neither partial-model nor model-verification runs are required; the model search may be conducted using paramodulation runs only.

4. AN EXAMPLE OF A MODEL SEARCH

Some of the automated theorem prover runs made in searching for the model of Appendix IB are listed below to indicate the degree of our reliance on the computer. As might be expected, the search includes tests which appear inconclusive or unnecessary in retrospect.

1) Paramodulation runs proved Grau axioms 4 and 5 (see Appendix I) from axioms 1, 2, and 3, and incidentally derived G(G(X))=X from axioms 1, 2, and 3.

2) A paramodulation run attempting to prove axiom 2 from axioms 1, 3, 4, and 5, proved neither axiom 2 nor G(G(X))=X. Incidentally, F(X,Y,X)=X was derived.

The failure to prove axiom 2 motivated the search for a counterexample. The fact that G(G(X))=X was not derived suggested that a model violating G(G(X))=X be attempted. Such a model would necessarily violate axiom 2. It could be based on one generator (an A for which G(G(A)¬=A) rather than two (as A and B for which F(B,A,A)¬=A), possibly requiring fewer elements, fewer defining relations, and less computer time for verification.

3) Paramodulation run seeking consequences of axioms 1, 3, 4, and 5, in conjunction with G(G(G(X)))=G(X), G(F(X,Y,Z))=F(G(X),G(Y),G(Z)), and F(X,Y,Z)=F(X,Z,Y). Axiom 2 was not proved, but the last of these equalities together with axioms 3 and 5 yielded G(G(X))=X.

Because a model with G(G(A))¬=A was being sought, the last equality was not used for subsequent models. The possibility that G(G(G(X)))=G(X) might imply G(G(X))=X was not tested at this time. (The extra equalities were added in an attempt to add enough structure to help get a model, but without satisfying axiom 2. Equalities known to be true in ternary Boolean algebras were added so that the known structure would be approached; for example, G(G(G(X)))=G(X) is a weakening of G(G(X))=X.)

4) A paramodulation run deriving consequences of axioms 1, 3, 4, and 5, in conjunction with G(F(X,Y,Z))=F(G(X),G(Y),G(Z)) and F(A,X,G(G(A)))=A, derived no undesirable consequences. The latter equality, when X=A, is an instance of axiom 4; when X=G(A) it is an instance of axiom 3; the author hoped the generalization to X=G(G(A)) as well, violating axiom 2, would lead to a model violating axiom 2.

5) Partial-model run, using the first seven function defining equalities of Appendix IB. The next two equalities were derived (in checking satisfaction of axiom 1).

6) Partial-model run, using all but the eighth equality of Appendix IB. The eighth equality was derived.

The ninth equality was suggested, before runs 5 and 6 were made, by an examination of the proof of G(G(A))=A from axioms 1, 2, and 3. This proof used the instance F(A,G(G(A)),G(G(A)))=G(G(A)) of axiom 2; if this term had the value A instead, G(G(A))=A would not be proven.

7) Model-verification run verifying the model of Appendix IB.

One goal of future work is to automate more of the interactive process illustrated here, conceivably following the general plan of Fig. 1.

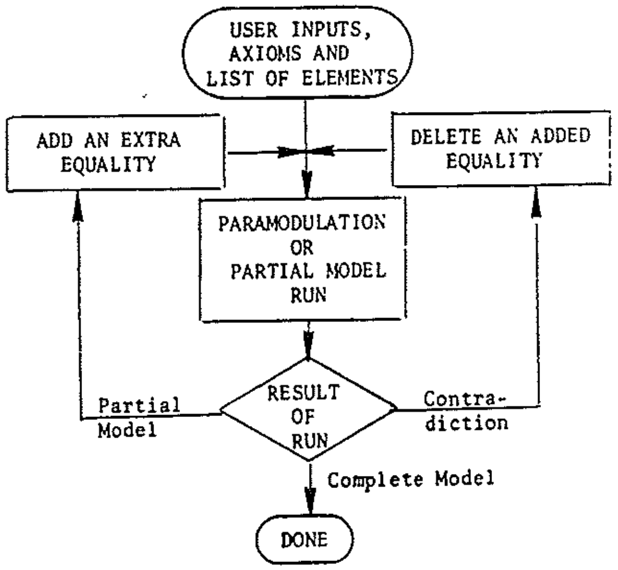


Fig. 1. Flowchart for Model Search

5. THE ROLE OF THE AUTOMATED THEOREM PROVER

The theorem prover served as a "logical calculator", rapidly performing calculations which would have been laborious if done by hand. The calculations were of two types: first, given a set of equalities, obtain a list of consequences; second, test the validity of a model or partial model.

The program helped the author to operate effectively with an unfamiliar axiom system.

These benefits were obtained without recourse to new programming, attesting to the generality and flexibility of the existing theorem proving program and techniques.

The program did not decide what sort of model to seek; this was up to the user, who made the intelligent decisions. Symmetry and other arguments had to be made by the user (see Appendix III).

The automated theorem prover is limited in the number of instances of an axiom which can be verified. An axiom containing m variables, will when there are n distinct elements have n to the m-th power instances to be checked; when n to the m-th power exceeds 500 to 5000 the cost of computer time becomes high. Presumably various methods of checking many instances at once can be developed to deal with larger models (see Appendix III); one would hope that some of these will be general-purpose rather than problem-dependent.

6. ACKNOWLEDGEMENTS

The author wishes to thank Dr. L. Wos for the encouragement, his enthusiasm in using the techniques of this paper to seek more models [6], our discussions concerning refinements of these methods (Appendix III), and his advice concerning this paper; and to thank Dr. Ross Overbeek and many others for the man years of work devoted to designing, writing, maintaining, and improving the "NIUTP" theorem proving program used in this work.

APPENDIX I. GRAU TERNARY BOOLEAN ALGEBRAS AND MODELS

A. AXIOMS FOR A GRAU TERNARY BOOLEAN ALGEBRA

The ternary boolean algebra discussed herein was first presented in [2]. The axioms are:

Axiom 1:	F(V,W,F(X,Y,Z))=F(F(V,W,X),Y,F(V,W,Z))
Axiom 2:	F(Y,X,X)=X
Axiom 3:	F(X,Y,G(Y))=X
Axiom 4:	F(X,X,Y)=X
Axiom 5:	F(G(Y),Y,X)=X

Axioms 1, 2, and 3 imply 4 and 5 [1,6]. The techniques of this paper were used to establish that Axiom 2 is independent of axioms 1, 3, 4, and 5 and that Axiom 3 is independent of Axioms 1, 2, 4, and 5 [6]. Other results, concerning the above axioms, and discovered using the techniques of this paper, appear in [6].

B. A MODEL FOR AXIOMS 1, 3, 4, AND 5 VIOLATING AXIOM 2

The following form a complete set of reductions, defining closed functions F and G on three elements A, G(A), and G(G(A)). All variables are universally quantified.

F(X,Y,G(Y)) = X	(Axiom 3)
F(X,X,Y) = X	(Axiom 4)
F(G(Y),Y,X) = X	(Axiom 5)
F(X,Y,X) = X	(Consequence of Axioms 1 and 4)
G(G(G(X))) = G(X)	Special hypothesis
F(X,G(G(Z)),G(Z)) = X	Implied equality
F(G(Z),G(G(Z)),X) = X	Implied equality
F(A,G(G(A)),G(G(A))) = A	Implied equality
F(G(G(A)),A,A) = G(G(A))	Implied equality
F(A,G(A),G(A)) = G(A)	Special hypothesis

The functions thus defined satisfy axiom 1 in all instances, as was demonstrated using our automated theorem-proving program. The last three equalities violate axiom 2; the model thus shows axiom 2 to be independent of axioms 1, 3, 4, and 5.

C. A MODEL FOR AXIOMS 1, 3, 4, AND 5, VIOLATING AXIOM 2, BUT SATISFYING G(G(X)) = X

Elements: A, G(A), C, G(C)

Function defining equalities:

```
Unit clauses      F(X,Y,G(Y)) = X;  F(X,X,Y) = X;
F(G(Y),Y,X) = X;  F(X,Y,X) = X;     G(G(X)) = X;
F(X,G(Z),Z) = X;  F(Z,G(Z),X) = X;
F(A,C,C) = A      and seven variants;
F(A,C,G(A)) = A   and seven variants.
```

The model has 8-fold symmetry given by the permutations (A G(A)), (C G(C)), and (A C)(G(A) G(C)). The variants of the last two equalities are obtained by applying these symmetries; thus F(G(A),C,C) = G(A). The equality F(A,C,C) = A and its variants violate axiom 2.

APPENDIX II. CLAUSES USED FOR MODEL CHECKING RUNS

A. GENERATION OF AXIOM INSTANCES

All instances of axiom 1 (distributive axiom) which may be formed by substituting the elements A, G(A), and G(G(A)) for the variables, may be generated by forming all hyperresolvents of the following clauses:

```
    Units  Q(A);  Q(G(A));  Q(G(G(A)));  nucleus
      ¬Q(V)  ¬Q(W)  ¬Q(X)  ¬Q(Y)  ¬Q(Z)  or
  EQUAL (F(V,W,F(X,Y,Z)), F(F(V,W,X),Y,F(V,W,Z)))
```

The general principles are: Write a clause Q(e) for each element e in the proposed model. These clauses serve as hyperresolution electrons. Write the hyperresolution nucleus as the disjunction of the equality to be instantiated and, for each variable v appearing in the equality, the literal ¬Q(v). Hyperresolution will generate n to the m-th power instances, where n is the number of model elements and m is the number of variables appearing in the equality being checked.

Note: It would appear that generation of instances could be done almost equally well by P1 deduction, or by forward chaining using an appropriate "nucleus" like

    Q(X) -> (Q(Y) -> ... -> EQUAL(...,...)...)) .

The demodulation (simplification, reduction) of the instances could then be achieved by any general system of algebraic simplification. The author would appreciate hearing from other workers who repeat these experiments with their programs.

B. TESTING OF CLOSURE

To test closure of a ternary function F, on elements A, G(A), and G(G(A)), form all hyperresolvents of the following clauses:

    Q(A); Q(G(A)); Q(G(G(A))); and nucleus
    ¬Q(X) ¬Q(Y) ¬Q(Z) or Q(F(X,Y,Z)) .

Demodulate each derived clause using the equalities defining F. If each resulting clause is identical to Q(e) for some element e in the model, then F is closed. Otherwise F is not closed: for example if Q(F(A,G(A),G(A)) is derived and does not simplify, F(A,G(A),G(A)) has not been defined. The general principles are: Write a clause Q(e) for each element e of the proposed model. Write the nucleus (for an n-ary function F) as:

    ¬Q(V1)... ¬Q(Vn) or Q(F(V1,...,Vn)) .

APPENDIX III. METHODS FOR REDUCING RUN TIME OF PARTIAL-MODEL AND MODEL-VERIFICATION RUNS

If a fairly large model is being verified, the number of instances of axioms to be verified may be reduced by symmetry and other considerations.

For example, the four element model of Grau axioms 1, 3, 4, and 5, described in Appendix IC, has a set of symmetries mapping any element into any other. Thus in checking the instances of axiom 1, only instances in which A is substituted for V need be checked; instances in which G(A), C, or G(C) is substituted for V will behave analogously. The clauses used to generate the restricted set of instances were:

    Q(A); Q(G(A)); Q(C); Q(G(C));
    Q2(A); and nucleus
    ¬Q2(V) ¬Q(W) ¬Q(X) ¬Q(Y) ¬Q(Z) or
    EQUAL (F(V,W,F(X,Y,Z)), F(F(V,W,X),Y,F(V,W,Z))).

The use of ¬Q2(V) and Q2(A) causes only A to be substituted for V. The set of instances to be checked may be further reduced by examining what happens when V=W or V=G(W). When V=W,

    EQUAL (F(V,V,X),Y,F(V,V,Z)), F(V,V,F(X,Y,Z)))

reduces to EQUAL(F(V,Y,V),V) by Axiom 4, and thence to EQUAL(V,V) by F(X,Y,X)=X. Thus (assuming the defining equalities form a complete set of reductions) instances where V and W are given the same value need not be checked. Similarly when V=G(W),

    EQUAL (F(F(G(W),W,X),Y,F(G(W),W,Z)),
           F(G(W),W,F(X,Y,Z)))

reduces to EQUAL(F(X,Y,Z),F(X,Y,Z)) by Axiom 5; again instances satisfying V=G(W) (e.g. G(A)/V, A/W; A/V, G(A)/W if G(G(A))=A) need not be checked individually. Thus in checking the four element model, only A/V with C/W or G(C)/W need be checked in full; this may be done by generating all hyperresolvents of the following clauses:

    Q(A); Q(G(A)); Q(C); Q(G(C));
    Q2(A);
    Q3(C); Q3(G(C)); and nucleus
    ¬Q2(V) ¬Q3(W) ¬Q(X) ¬Q(Y) ¬Q(Z) or
    EQUAL (F(V,W,F(X,Y,Z)), F(F(V,W,X),Y,F(V,W,Z)))

It is hoped that such tricks can be incorporated in a general-purpose, fully automated package, but at present the desired tricks are discovered and set up by the user. Tricks of various kinds will presumably become very important in verifying large models, as the numbers of instances of certain axioms become enormous.

APPENDIX IV. CLAUSES FOR GENERATING A LIST OF ELEMENTS AND A FUNCTION TABLE

To generate a list of elements of a model, generate all hyperresolvents of the following clauses and their hyperresolvent consequences (demodulating each using the function defining equalities):

    ¬Q(X) or Q(G(X))
    ¬Q(X) ¬Q(Y) ¬Q(Z) or Q(F(X,Y,Z))
    ...(one such clause for each function, as in
       closure test)
    and units Q(A); Q(B); ...
    (one such clause for each generator of the
     model).

This is useful when a model has been found using paramodulation runs only (Section 3, Note 4).

To generate a function table, generate all hyperresolvents of the following clauses (demodulating each using the function defining equalities):

    ¬Q(X) ¬Q(Y) ¬Q(Z) or PF(X,Y,Z,F(X,Y,Z))
    (for a ternary function F)
    and units Q(A); Q(G(A)); ...
    (one such clause for each element of the model).

The derived "PF" clauses give the function table entries for "F".

APPENDIX V. AN ALTERNATIVE METHOD OF CHECKING WELL-DEFINEDNESS IN A MODEL-VERIFICATION RUN

If the function defining equalities for a proposed model do not form a complete set of reductions, the following test for well-definedness may be used instead. (The reader is urged to contact the author for details of this method if desired; space considerations do not permit a full presentation here.) First, design the demodulation algorithm to satisfy two criteria:

1) In demodulating a given term, fully demodulate each subterm before applying demodulators to the full term. For example, in demodulating F(G(G(G(A))),A,A), demodulate G(G(G(A))) to G(A) before applying a demodulator to the term whose major symbol is F.

2) Demodulate a given ground term in the same way each time it appears. If two demodulators apply, choose the same one each time. These criteria insure that demodulation will act as a well-defined function.

Then check the function defining equalities for satisfaction in each instance, using the method applied to Grau axiom 1 in Appendix IIA with the above demodulation algorithm.

APPENDIX VI. APPLICATION TO MODELS OF FIRST ORDER NON-EQUATIONAL SYSTEMS

The techniques of this paper might be applied to non-equational systems by (1) rewriting each predicate, "OR", and "NOT" as functions; (2) supplying function defining equalities for the above which yield "T" and "F" as values, e.g. LT(1,2)=T, OR(T,F)=T, NOT(T)=F; (3) seeking and verifying models as in the preceding part of the paper, omitting T and F from the list of elements. The author has not tried this technique and makes no claim for its practical utility.

REFERENCES

1. Chinthayamma, Sets of independent axioms for a ternary Boolean algebra (preliminary report), Notices of the American Math. Soc., Vol. 16, No. 4, p. 654, June 1969.

2. A. A. Grau, Ternary Boolean Algebra, Bulletin of the American Math. Soc., Vol. 53, No. 6, pp. 567-572, June 1947.

3. D. E. Knuth and P. B. Bendix, Simple Word Problems in Universal Algebras, Computational Problems in Abstract Algebra, pp. 263-297, (Edited by J. Leech), Pergamon Press, Oxford (1970).

4. J. D. McCharen, R. A. Overbeek, and L. Wos, Problems and Experiments for and with Automated Theorem-Proving Programs, IEEE Transactions on Computers, Vol. C-25, No. 8, August 1976, pp. 773-782.

5. R. A. Overbeek, "An implementation of hyper-resolution", Comput. Math. Appl., Vol. 1, pp. 201-214, 1975.

6. Steve Winker and L. Wos, Automated Generation of Models and Counterexamples and its Application to Open Questions in Ternary Boolean Algebra, Proc. Eighth Int. Symposium on Multiple-Valued Logic, pp. 251-256. Rosemont, Illinois (1978); IEEE (1978).

7. L. Wos and G. A. Robinson, Paramodulation and Set of Support, Proc. IRIA Symposium on Automatic Demonstration, pp. 276-310. Versailles, France (1968); Springer-Verlag (1970).

8. L. Wos, G. A. Robinson and D. F. Carson, The Concept of Demodulation in Theorem Proving, J. Assn. Comput. Mach. 14, 4 687-709 (October 1967).

Conflicting Bindings and Generalized Substitutions*

Mabry Tyson
W. W. Bledsoe

University of Texas
Austin, Texas 78712

Problems arise combining conjunctive subgoals whose solutions require conflicting bindings. Using a generalization of substitution, a method is given that allows the combination of the solutions.

INTRODUCTION

One of the most productive methods of problem solving is problem reduction. If a problem can be split into two independent parts each of which may be solved separately, finding solutions to the smaller problems is a much simpler task [1]. While problem reduction is very basic to human problem solving, it is perhaps more important to problem solving by machines ([2],[3]). The recursion of algorithms that solve problems by reducing them to subproblems (to which the algorithm is reapplied) contributes to the clarity and conciseness of the algorithm. Also, present computer programs are not as adept at separating out the chaff as humans are and are therefore more susceptible to combinatorial explosions.

Solutions to independent problems are orthogonal and can be combined without interference. This is not so if the problems are not completely independent. If two parts of a problem are somewhat interdependent, it is necessary to confirm that their two solutions can be combined to return a single solution for the whole problem. Actually it is not the two parts that must be independent, it is their solutions. Since any problem may have a number of different solutions, for two problems, one pair of solutions may be independent while another pair may be mutually exclusive. (Example - If you need a number that is both odd and prime you would lose if you picked the first odd number for one subgoal and the first prime number for the other subgoal.)

The divide and conquer methodology is central to the method of theorem proving often referred to as "natural deduction" ([2],[4],[5],[6]). One of its principal proof techniques is splitting a single goal into multiple goals and later combining the results. Thus to prove

$$H \Rightarrow A \wedge B$$

the two subgoals

$$H \Rightarrow A \quad \text{and} \quad H \Rightarrow B$$

are proved. Unfortunately it is not quite as simple as this due to the presence of variables that may occur in H, A, B, or even in higher subgoals. This paper will examine the problems of independence of subgoals and their solutions within the context of a particular methodology of theorem proving. We will also show how the solutions must be combined to provide a solution of the higher level goal. Most of the time two subgoals may be combined rather simply. We present a theorem that defines the necessary conditions for this. We also present a theorem that covers the situations where the two solutions interfere with each other. In order to do this we will generalize the concept of substitutions. By combining substitutions θ and λ into forms (called generalized substitutions) such as $(\theta \vee \lambda)$ or $(\theta \wedge \lambda)$, conflicting solutions may be joined into a single solution.

OVERVIEW

The UT interactive theorem prover is a natural deduction system developed by our group at the University of Texas which has been used over a number of years to prove theorems in such areas as set theory, topology, program verification, and limit theorems of calculus and analysis.** The following discussion will be presented in terms of this implementation although the ideas and techniques extend much further. For a complete discussion of this prover see [7]. Proofs of the theorems presented below and further relevant information may be found in Appendix 3 of [7].

When a closed formula, E, is given to the prover, it is skolemized into an open (quantifier free) formula, S. By the nature of the skolemization, if there exists some substitution, θ, such that $S\theta$ is ground and true then the original formula E is true. Likewise if there is some set of substitutions $\theta_1, \theta_2, \ldots, \theta_m$ such that

$$(1) \qquad S\theta_1 \vee S\theta_2 \vee \ldots \vee S\theta_m$$

is both ground and true then E is true.

---

* This work was supported by National Science Foundation Grant MCS 77-20701.

** We will refer to this as our "traditional" prover to distinguish it from one which uses generalized substitutions.

We will use the term ground-true to describe a formula which is true for every ground interpretation of its free variables. A proof of such a formula would treat its variables as ground terms. An example would be the tautology

$$P(X) \rightarrow P(X).$$

The goal of the prover is to find some substitution θ such that if S is the skolemized form of the input then $S\theta$ is ground-true. For example, the skolemized form of

$$\exists Y\{\forall X(P(X,Y)) \rightarrow P(A,Y)\}$$

is

(2) $\quad P(X,Y) \rightarrow P(A,Y).$

The prover will attempt to find a substitution such as $\{A/X\}$ which when applied to (2) results in a ground-true formula

$$P(A,Y) \rightarrow P(A,Y).$$

A goal such as (2) will be described as provable if there is some substitution that will make the goal ground-true.

Consider what happens when the prover is given the theorem

(3) $\quad (\forall X\ P(X)) \rightarrow (P(A) \wedge P(B)).$

The skolemized form is

$$P(X) \rightarrow P(A) \wedge P(B)$$

which is given to the routine IMPLY. IMPLY is the recursive routine that has the task of determining a substitution (which IMPLY returns as its value) which makes its input ground-true. For this example IMPLY will recur on the two subgoals

$$P(X) \rightarrow P(A) \quad \text{and} \quad P(X) \rightarrow P(B)$$

The two subgoals are proved with a substitution $\{A/X\}$ for the first and $\{B/X\}$ for the second.

But is this enough for IMPLY to report it has proved its original input? The solutions were not independent as they both contained the variable X so they cannot be easily combined. In fact, no ordinary substitution exists which makes the original input to IMPLY ground-true. As we shall see, in this particular case the conflict in the substitutions for X does not lead to problems that prevent the original input from being proved true but there are non-theorems whose downfall is due to a similar step in attempted proofs.

Remember that interdependent goals require some special manipulations in order for their separate solutions to be combined to provide a solution for the combined goal. The prover does check to see that solutions returned from subgoals are such that they may be combined.

In doing an AND-SPLIT such as above, the prover does not actually prove the two subgoals independently. Instead, it waits until the first subgoal succeeds and then uses the result of that in setting up the second subgoal. This significantly reduces the chance that the second subgoal would be proved inconsistently with the first. If IMPLY is given the goal

(4) $\quad H \Rightarrow (A \wedge B)$

it will first prove

$$H \Rightarrow A$$

using some substitution θ. Then it will form the second subgoal

$$H \Rightarrow (B\theta)$$

and attempt to prove it.*

When the second subgoal returns a substitution of λ, the prover will return the composition of the two substitutions, $\theta\lambda$, as the substitution that proves the input goal (4).

SOUNDNESS OF OUR TRADITIONAL PROVER

An input formula is proved by the prover if IMPLY returns a substitution when applied to the result of skolemizing the original formula. By using an inductive proof we will show that IMPLY is sound. One possible requirement on IMPLY for this is that if IMPLY returns a substitution for a formula then that substitution will make the formula ground-true. The original input to the prover is proved if the initial (top-level) call to IMPLY returns a substitution satisfying this requirement since in that case (1) can be easily established.

Now we need to show that IMPLY actually has this property. All of the non-recursive IMPLY rules (eg., matching by unification) trivially satisfy this requirement. We shall only analyze the AND-SPLIT rule as all the recursive rules (subgoaling) are similar. In order to prove it we would need to show that if

$$(H \Rightarrow A)\theta$$

and

$$(H \Rightarrow B\theta)\lambda$$

are both ground-true (inductive hypothesis) then

$$(H \Rightarrow A \wedge B)\theta\lambda$$

----

* If one considers the proof of

$$\exists X\ (P(A) \wedge Q(B) \wedge Q(A) \rightarrow (P(X) \wedge Q(X)))$$

then it should be clear why the θ needs to be applied to the conclusion of the second subgoal.

is also ground-true. Unfortunately we need further restrictions in order to prove this.

Difficulties arise when the two substitutions contain a conflict such as the ones above ({A/X} and {B/X}). Two substitutions are in conflict if they substitute different terms for the same variable. Problems are also encountered if a substitution is such that some element of its domain occurs in its range, that is, if the composition of the substitution with itself differs from original substitution. So we will put conditions on the solutions returned by the subgoals in order to make the theorem provable.

Definition. A substitution θ is called <u>normal</u> if the composition of θ with itself is θ again.

Substitutions of the form {F(X)/X} or {F(Y)/X, A/Y} are not normal substitutions.

Definition. Two substitutions θ and λ are said to <u>conflict</u> if their domains are not disjoint.

Theorem: If θ, λ, and $\theta\lambda$ are all normal and θ and λ do not conflict, then if

$$(H \rightarrow A)\theta$$

is ground-true and

$$(H \rightarrow B\theta)\lambda$$

is ground-true, then

$$(H \rightarrow A \wedge B)\theta\lambda$$

is also ground-true.

The condition imposed on IMPLY (by this theorem) is that it return a substitution which not only makes the goal ground-true but also satisfies certain conditions of form. We then have to restrict the combining of solutions in an AND-SPLIT so that only solutions of the required form are returned.

Slightly modifying the above theory has allowed our traditional prover to prove theorems that require multiple substitutions for a variable. Previously the prover would note conflicts in substitutions and would halt a proof if the conflicts might give rise to problems. The above example (3) with the conflicting substitutions was handled by returning the self-conflicting substitution {A/X, B/X}. The prover would not allow this substitution to be applied to any other formula containing an X. By using this method the prover handles most cases. However, the simplest example of a theorem the traditional prover would halt on is

(5) $Q(A) \wedge Q(B) \rightarrow$
$\exists X((P(X) \rightarrow P(A) \wedge P(B)) \wedge Q(X))$.

The proof of the first half of this conclusion proceeds as in the previous example but the prover halts when trying to substitute {A/X, B/X} into Q(X).

GENERALIZED SUBSTITUTIONS

We have developed a theory that allows the prover to combine any substitutions so proofs (such as the proof of (5)) may proceed despite possible conflicts. The central idea of the theory is the notion of generalized substitutions. Basically a generalized substitution contains both substitutions and information about the relationship of these substitutions.

Definition. θ is a <u>generalized substitution</u> if
(1) θ is an ordinary substitution, or
(2) θ has the one of the forms

$$(\theta_1 \vee \theta_2), \quad \text{or} \quad (\theta_1 \wedge \theta_2)$$

where θ_1 and θ_2 are generalized substitutions.

Definition. If θ is a generalized substitution, then we define θ' by
(1) $\theta' = \theta$ if θ is an ordinary substitution,
(2) $(\theta_1 \vee \theta_2)' = (\theta_1' \wedge \theta_2')$,
(3) $(\theta_1 \wedge \theta_2)' = (\theta_1' \vee \theta_2')$.

Definition. A generalized substitution is said to be a <u>pure disjunction</u> (<u>conjunction</u>) if it contains no $\wedge$ symbols ($\vee$ symbols).

Ordinary substitutions are both pure disjunctions and pure conjunctions.

Definition. If A is a formula and θ is a generalized substitution, then $A\theta$ is the formula gotten by applying θ from left to right, ie,
(1) $A\theta$ is the usual result if θ is an ordinary substitution,
(2) $A(\theta_1 \vee \theta_2) = A\theta_1 \vee A\theta_2$,
(3) $A(\theta_1 \wedge \theta_2) = A\theta_1 \wedge A\theta_2$.

Properties: If θ and λ are generalized substitutions, λ is a pure disjunction, and A and B are formulas then λ' is a pure conjunction and
(1) $(\theta')' = \theta$
(2) $-(A\theta) = (-A)\theta'$
(3) $(A \vee B)\lambda = A\lambda \vee B\lambda$
(4) $(A \wedge B)\lambda' = A\lambda' \wedge B\lambda'$
(5) $(A \rightarrow B)\lambda = (A\lambda' \rightarrow B\lambda)$

IMPLY can now be expressed in a form such that it returns a pure disjunctive substitution that makes its input ground-true. As before, the original input to the prover is proved if the top-level call returns such a substitution. We will again use an inductive proof to show that IMPLY has the desired properties. The non-recursive rules of IMPLY pose no problem since ordinary substitutions are pure disjunctions. The proof of the soundness of the AND-SPLIT rule is typical of the recursive IMPLY rules.

Theorem: If θ and λ are pure disjunctive generalized substitutions then if θ is such that

$$(H \rightarrow A)\theta$$

is ground-true and λ is such that

$$(H \rightarrow B\theta')\lambda$$

is ground-true then $(\theta\lambda \vee \lambda)$ is a pure disjunctive generalized substitution and

$$(H \rightarrow A \wedge B)\ (\theta\lambda \vee \lambda)$$

is ground-true.

If IMPLY is given the goal of

$$(H \Rightarrow A \wedge B)$$

it will first form the subgoal

$$(H \Rightarrow A).$$

If this subgoal now returns the substitution θ, IMPLY will form the second subgoal

$$(H \Rightarrow B\theta').$$

If this subgoal returns the substitution λ, IMPLY will return the substitution of

$$(\theta\lambda \vee \lambda)$$

for the original goal.

Thus the two conflicting solutions of the subgoals generated by the earlier example

$$P(X) \Rightarrow P(A) \wedge P(B)$$

can be combined into the single generalized substitution

$$(\{A/X\} \vee \{B/X\}).$$

When this method is applied to the example

$$Q(A) \wedge Q(B) \Rightarrow \exists X(\ (P(X) \rightarrow P(A) \wedge P(B)) \wedge Q(X))$$

the second subgoal becomes

$$Q(A) \wedge Q(B) \Rightarrow Q(X)(\{A/X\} \vee \{B/X\})'$$

which is just

$$Q(A) \wedge Q(B) \Rightarrow Q(X)\{A/X\} \wedge Q(X)\{B/X\}$$

and is proved.

At first glance it appears that the use of generalized substitutions increases the amount of work in the simpler cases one level up. If one considers

$$(6) \qquad H \Rightarrow ((P \wedge Q) \wedge R),$$

and if the substitution proving

$$H \Rightarrow P$$

is $\{A/X\}$ and the substitution proving

$$H \Rightarrow Q\{A/X\}'$$

is $\{B/Y\}$ then by combining these two, the generalized substitution proving the first subgoal of (6)

$$H \Rightarrow P \wedge Q$$

is $(\{A/X, B/Y\} \vee \{B/Y\})$. To finish the proof of (6) we need to prove

$$H \Rightarrow R(\{A/X, B/Y\} \vee \{B/Y\})'$$

which is

$$H \Rightarrow R\{A/X, B/Y\} \wedge R\{B/Y\}.$$

It is easy to see that this is equivalent to

$$H \Rightarrow R\{A/X, B/Y\}$$

which is what would have to be proved using the procedure that does not allow conflicts. The prover can detect this rather simply.

CONCLUSION

We have given a proof of the soundness of the natural deduction prover that has been in use at the University of Texas for a number of years. By developing generalized substitutions we have extended the capability of the prover to allow proofs where conjunctive subgoals require conflicting substitutions. We feel that both the method of combining conflicting bindings and the proof technique presented here may be useful in other fields.

REFERENCES

1. Nils J. Nilsson. Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, 1971.

2. A. Newell, J.C. Shaw and H.A. Simon. Empirical explorations of the logic theory machine: a case study in heuristics. RAND Corp. Memo P-951, Feb. 28, 1957. Proc. Western Joint Computer Conf. 1956, 218-239. Computers and Thought, Feigenbaum and Feldman, 134-152.

3. James R. Slagle. A heuristic program that solves symbolic integration problems in freshman calculus, JACM, Vol 10, 1963, 507-520.

4. H. Gelernter. Realization of a geometry theorem-proving machine. Proc. Int'l Conf. Information Processing, 1959, Paris UNESCO House, 273-282.

5. Raymond Reiter. A semantically guided deductive system for automatic theorem proving. Proc. Third IJCAI, 1973, 41-46.

6. Richard Fikes and Gary Hendrix. A network-based knowledge representation and its natural deduction system, Proc. Fifth IJCAI, 1977, 235-246

7. W.W. Bledsoe and Mabry Tyson. The UT interactive theorem prover. The Univ. of Texas at Austin Math. Dept. Memo ATP-17a, June 1978. (Appendix 3 available separately upon request.)

GENERAL MATINGS

Peter B. Andrews

Department of Mathematics
Carnegie-Mellon University
Pittsburgh, Pennsylvania, 15213

§1 Introduction

Much of the research in automatic theorem-proving has been focused on developing efficient methods for deriving contradictions from sets of clauses, which represent the conjuncts (disjunctions of literals) of a wff whose matrix is in conjunctive normal form. The advantages of conjunctive normal form were pointed out in [6], and incorporated into the widely studied resolution method [12].

Many theorems of mathematics and other disciplines lend themselves naturally to representation as sets of clauses. However, experience [2] with a wide variety of theorems has shown that in many more cases, the use of clausal form has serious disadvantages, since the repeated use of the distributive law $[P \vee (Q \wedge R)] \equiv [(P \vee Q) \wedge (P \vee R)]$ involved in the reduction to conjunctive normal form often causes wild proliferation of literals.

Example 1 in Figure 1 illustrates this point. Line (1) is the proposed theorem, (2) is equivalent to its negation, and (4) is the result of introducing Skolem functions and dropping quantifiers. Lines (c1)-(c16) are the clauses of the conjunctive normal form of line (4) (with four tautologous clauses deleted). Line (4) contains 12 literals, but there are 56 literal-occurrences in clauses (c1)-(c16).

(Example 1 concerns a theorem of second order logic, but it can be transformed into a theorem of first order logic in a trivial way: Replace each atom Ft of Example 1 by KFt, where K is a binary predicate constant, and regard all predicate constants or variables in Example 1 as individual constants or variables. However, this transformation merely clutters the notation, so we shall continue to use the notation of Fig. 1 for this example.)

Fig. 1
Example 1

Proposed theorem:

(1) $\exists S\, \forall x[[Sx \vee \underline{P}x] \wedge [\sim Sx \vee \underline{Q}x]]$
$\quad \equiv \forall y[\underline{P}y \vee \underline{Q}y]$

Negate (1):

(2) $[\exists S\, \forall x[[Sx \vee \underline{P}x] \wedge [\sim Sx \vee \underline{Q}x]]$
$\quad \wedge \exists z[\sim \underline{P}z \wedge \sim \underline{Q}z]] \vee [\forall y[\underline{P}y \vee \underline{Q}y]$
$\quad \wedge \forall S\, \exists w[[\sim Sw \wedge \sim \underline{P}w]$
$\quad \vee [Sw \wedge \sim \underline{Q}w]]]$.

Skolemize (2):

(3) $[\forall x[[\underline{R}x \vee \underline{P}x] \wedge [\sim \underline{R}x \vee \underline{Q}x]] \wedge \sim \underline{Pa}$
$\quad \wedge \sim \underline{Qa}] \vee [\forall y[\underline{P}y \vee \underline{Q}y] \wedge \forall S[[\sim S[\underline{f}S]$
$\quad \wedge \sim \underline{P}[\underline{f}S]] \vee [S[\underline{f}S] \wedge \sim \underline{Q}[\underline{f}S]]]]$

Drop quantifiers:

(4) $[[\underline{R}x \vee \underline{P}x] \wedge [\sim \underline{R}x \vee \underline{Q}x] \wedge \sim \underline{Pa}$
$\quad \wedge \sim \underline{Qa}] \vee [[\underline{P}y \vee \underline{Q}y] \wedge [[\sim S[\underline{f}S]$
$\quad \wedge \sim \underline{P}[\underline{f}S]] \vee [S[\underline{f}S] \wedge \sim \underline{Q}[\underline{f}S]]]]$

Clauses:

(c1)	$\underline{R}x \vee \underline{P}x \vee \underline{P}y \vee \underline{Q}y$
(c2)	$\underline{R}x \vee \underline{P}x \vee \sim S[\underline{f}S] \vee \sim \underline{Q}[\underline{f}S]$
(c3)	$\underline{R}x \vee \underline{P}x \vee \sim \underline{P}[\underline{f}S] \vee S[\underline{f}S]$
(c4)	$\underline{R}x \vee \underline{P}x \vee \sim \underline{P}[\underline{f}S] \vee \sim \underline{Q}[\underline{f}S]$
(c5)	$\sim \underline{R}x \vee \underline{Q}x \vee \underline{P}y \vee \underline{Q}y$
(c6)	$\sim \underline{R}x \vee \underline{Q}x \vee \sim S[\underline{f}S] \vee \sim \underline{Q}[\underline{f}S]$
(c7)	$\sim \underline{R}x \vee \underline{Q}x \vee \sim \underline{P}[\underline{f}S] \vee S[\underline{f}S]$
(c8)	$\sim \underline{R}x \vee \underline{Q}x \vee \sim \underline{P}[\underline{f}S] \vee \sim \underline{Q}[\underline{f}S]$
(c9)	$\sim \underline{Pa} \vee \underline{P}y \vee \underline{Q}y$
(c10)	$\sim \underline{Pa} \vee \sim S[\underline{f}S] \vee \sim \underline{Q}[\underline{f}S]$
(c11)	$\sim \underline{Pa} \vee \sim \underline{P}[\underline{f}S] \vee S[\underline{f}S]$
(c12)	$\sim \underline{Pa} \vee \sim \underline{P}[\underline{f}S] \vee \sim \underline{Q}[\underline{f}S]$
(c13)	$\sim \underline{Qa} \vee \underline{P}y \vee \underline{Q}y$
(c14)	$\sim \underline{Qa} \vee \sim S[\underline{f}S] \vee \sim \underline{Q}[\underline{f}S]$
(c15)	$\sim \underline{Qa} \vee \sim \underline{P}[\underline{f}S] \vee S[\underline{f}S]$
(c16)	$\sim \underline{Qa} \vee \sim \underline{P}[\underline{f}S] \vee \sim \underline{Q}[\underline{f}S]$

Another disadvantage of representing a wff by clauses is that one's attention, and methods, tend thereby to be focused on certain parts of the wff in isolation from their relation to the rest of it.

The use of clauses seems to be fundamental to the resolution method, and variants of it. However, the basic ideas underlying matings [1] can easily be applied to wffs not represented by clauses. It was shown in [1] that matings are naturally induced by resolution-style refutations. Actually, it appears that all refutation and proof procedures for first order logic tacitly involve the construction of matings, which embody much of the essential logical structure of the final refutations or proofs. Our present purpose is to present the logical foundations of refutation procedures based on matings of arbitrary wffs, and to discuss one such procedure.

§2 Logical Preliminaries

In our formal development we shall be concerned with wffs of a system of first order logic whose primitive connectives are $\sim$ (not), $\wedge$ (and), $\vee$ (or), $\forall$ (for all), and $\exists$ (there exists...such that). In examples of wffs, underlined letters are constants, and other letters are variables; $\supset$ and $\equiv$ are to be regarded as abbreviations. We use Church's dot convention [4] for omitting brackets. We write θA for the result of applying a substitution θ to an expression A.

A wff C is in negation normal form (nnf), and is a negation normal formula (nnf), iff the scope of each occurrence of $\sim$ in C is atomic. A wff can easily be transformed into an equivalent nnf by using the laws $\sim\sim M \equiv M$, $\sim[M \wedge N] \equiv [\sim M \vee \sim N]$, $\sim[M \vee N] \equiv [\sim M \wedge \sim N]$,

This research was supported by NSF Grants MCS76-06087 and MCS78-01462.

$\sim \forall x\, M \equiv \exists x \sim M$, and $\sim \exists x\, M \equiv \forall x \sim M$. In general, the use of normal forms (such as negation, conjunctive, disjunctive, and prenex normal forms) provides conceptual simplicity which facilitates theoretical discussions, but it may make particular examples more cumbersome. (Example 1 illustrated this phenomenon for the case of conjunctive normal form. Another example is provided by increases in the degrees of Skolem functions which may be caused by putting a wff into prenex normal form before Skolemizing.) However, when one puts a wff into nnf one can obtain a wff no longer than the original one, and having the same essential logical structure. Therefore, without any real loss of generality we may often confine our attention to wffs in nnf.

An occurrence of a quantifier or a well formed part of a wff C is positive [negative] in C iff it is in the scope of an even [odd] number of occurrences of $\sim$. A wff is universal iff all its universal quantifiers occur positively, and all its existential quantifiers occur negatively, in it. It is well known how to introduce Skolem functions into a wff C so as to obtain a universal wff D (called the Skolemized form of C) such that C has a model if and only if D has a model. (We say that a wff has a model iff its universal closure is satisfiable; for wffs with free variables, this is not quite the same as satisfiability, though the two phrases are sometimes confused in the literature of theorem-proving.)

Given a wff B which we wish to show is valid, we let C be a negation normal form of $\sim \overline{B}$ (where $\overline{B}$ is the universal closure of B), and let D be the Skolemized form of C. Then D is a universal sentence in nnf which has no model if and only if B is valid. Thus we shall concentrate on the problem of refuting universal sentences in nnf.

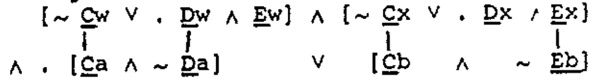
We say that a wff F is normal iff no quantifier occurrence of F contains a variable which occurs free in F or occurs in any other quantifier occurrence of F. By appropriate alphabetic changes of bound variables, any wff E can be transformed (or normalized) into a normal wff F (called a normal form of E) such that $\models [E \equiv F]$. ($\models A$ means that A is valid.)

Let D be a universal sentence in nnf. We next define the set of amplifications of D. We say that a wff H' is obtained from a wff H by quantifier duplication if H' is the result of replacing some wf part of H of the form $\forall x\, M$ by $\forall x\, M \wedge \forall x\, M$. If there is a sequence $D_1, \ldots, D_n$ of wffs (where $n \geq 1$) such that D_{i+1} is obtained from D_i by quantifier duplication for each $i < n$, we say that D_n is obtained from D_1 by a sequence of quantifier duplications.

Fig. 2
Example 2

(B) $\forall x[\underline{C}x \supset . \underline{D}x \wedge \underline{E}x]$
$\quad \supset . \forall y[\underline{C}y \supset \underline{D}y] \wedge \forall z[\underline{C}z \supset \underline{E}z]$
(C) $\forall x[\sim \underline{C}x \vee . \underline{D}x \wedge \underline{E}x]$
$\quad \wedge . \exists y[\underline{C}y \wedge \sim \underline{D}y] \vee \exists z[\underline{C}z \wedge \sim \underline{E}z]$
(D) $\forall x[\sim \underline{C}x \vee . \underline{D}x \wedge \underline{E}x]$
$\quad \wedge . [\underline{C}a \wedge \sim \underline{D}a] \vee [\underline{C}b \wedge \sim \underline{E}b]$
(E) $\forall x[\sim \underline{C}x \vee . \underline{D}x \wedge \underline{E}x]$
$\quad \wedge \forall x[\sim \underline{C}x \vee . \underline{D}x \wedge \underline{E}x]$
$\quad \wedge . [\underline{C}a \wedge \sim \underline{D}a] \vee [\underline{C}b \wedge \sim \underline{E}b]$
(F) $\forall w[\sim \underline{C}w \vee . \underline{D}w \wedge \underline{E}w]$
$\quad \wedge \forall x[\sim \underline{C}x \vee . \underline{D}x \wedge \underline{E}x]$
$\quad \wedge . [\underline{C}a \wedge \sim \underline{D}a] \vee [\underline{C}b \wedge \sim \underline{E}b]$
(G) $[\sim \underline{C}w \vee . \underline{D}w \wedge \underline{E}w] \wedge [\sim \underline{C}x \vee . \underline{D}x \wedge \underline{E}x]$
$\quad \wedge . [\underline{C}a \wedge \sim \underline{D}a] \vee [\underline{C}b \wedge \sim \underline{E}b]$

Mating for G :


[~ Cw ∨ . Dw ∧ Ew] ∧ [~ Cx ∨ . Dx ∧ Ex]
∧ . [Ca ∧ ~ Da] ∨ [Cb ∧ ~ Eb]

Now suppose E is obtained from D by some sequence of quantifier duplications, F is a normal form of E, and G is the result of deleting all quantifiers of F. Then G is called an amplification of D. (See Example 2 in Fig. 2.)

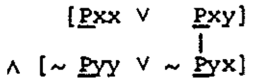
Let G be a quantifier-free wff. We let $\mathcal{L}(G)$ be the set of occurrences of literals in G. A mating $\mathfrak{m}$ of G is a binary relation on $\mathcal{L}(G)$ such that there is a substitution θ such that $\theta K = \sim \theta L$ whenever $L\mathfrak{m}K$ (i.e., whenever L and K are mated literal-occurrences). In first order logic, whenever such a substitution θ exists, there is an essentially unique most general such substitution $\theta_{\mathfrak{m}}$, which we call the substitution associated with $\mathfrak{m}$. We say that $\mathfrak{m}$ is a refutation mating of G iff G is false with respect to every assignment of truth values to atoms that gives opposite truth values to literals which have mated literal-occurrences.

Continuing with Example 2, we present a mating of G at the bottom of Fig. 2 by drawing lines between mated literal-occurrences. This mating is a refutation mating, as we shall see.

Note that if $\mathfrak{m}$ is a refutation mating, then $\theta_{\mathfrak{m}}G$ is truth-functionally contradictory. That the converse of this statement is not quite true can be seen from Fig. 3. (Of course, the mating of Fig. 3 can be extended to a refutation mating in a natural way.)

The following fundamental result may be regarded as a form of Herbrand's Theorem:

Fig. 3



Theorem 1. Let D be a universal sentence of first order logic in nnf. D has no model iff some amplification of D has a refutation mating.

§3 *Acceptability of Matings*

Constructing a mating involves two processes: (a) the *pairing* process, which decides which pairs of literal-occurrences to mate, and (b) the *unification* process, which determines whether there is a substitution which makes mated pairs complementary. Both of these processes involve computational effort, and each generates information which can be useful to the other. If the unification process determines that there is no substitution which unifies the atoms of a pair of literals and is compatible with the mating as constructed so far, then the pairing process need not consider adding that pair of literals to the mating. On the other hand, if the pairing process decides directly that it would not be useful to add a given pair of literals to the mating, then the unification algorithm need not consider whether an appropriate substitution exists.

Thus, it may often be desirable for these processes to work in parallel and to interact with each other, with the balance of effort being determined by the relative efficiencies of the algorithms, and the complexities of their tasks. When this is done, the pairing process may work with a set of pairs of literal-occurrences which is not known to be a mating, since it is not yet known whether an associated substitution exists. Such a set of pairs will be called a *potential mating*. Sometimes, speaking loosely, we shall refer to a potential mating as a mating.

The pairing process will need criteria, which we shall call *acceptability criteria*, to decide whether a given mating is a refutation mating. We shall discuss one acceptability criterion below, but it is clear that the problem of devising new and better acceptability criteria provides a rich field for future research. It is useful for acceptability criteria to have the following properties:

(1) When a mating fails the criteria, they should suggest ways in which the mating might profitably be altered.

(2) The criteria should be compatible with a step-by-step construction of a mating, so that information acquired in checking the criteria at one step can be used at the next step.

(3) The criteria should be compatible with a process in which the construction of a refutation mating is combined with choosing an appropriate amplification of the sentence to be refuted.

We next describe an acceptability

Fig. 4

$$\begin{bmatrix} \underline{R}x \lor \underline{P}x \\ \sim \underline{R}x \lor \underline{Q}x \\ \sim \underline{P}a \\ \sim \underline{Q}a \end{bmatrix} \lor \begin{bmatrix} \underline{P}y \lor \underline{Q}y \\ \begin{bmatrix} \sim S[\underline{f}S] \\ \sim \underline{P}[\underline{f}S] \end{bmatrix} \lor \begin{bmatrix} S[\underline{f}S] \\ \sim \underline{Q}[\underline{f}S] \end{bmatrix} \end{bmatrix}$$

criterion which is related to disjunctive normal form, and which is discussed, in slightly different terminology, by Bibel [3].

We find it useful to display wffs in nnf in a two-dimensional format, with disjunctions being displayed horizontally but with conjunctions being displayed vertically. Thus the wff (4) of Example 1 may be displayed as in Fig. 4.

Motivated by this representation, we can define a *vertical path* through a quantifier-free nnf G to be a sequence of members of $\mathcal{L}(G)$ which corresponds to one of the disjuncts (conjunctions of literals) in the disjunctive normal form of G. Intuitively, one chooses a vertical path through G by choosing one disjunct (M or N) from each disjunction [M ∨ N] of G, and deleting all parts of G which are not chosen. One vertical path through the wff of Fig. 4 has literals $\underline{P}x$, $\sim \underline{R}x$, $\sim \underline{P}a$, $\sim \underline{Q}a$, and another has literals $\underline{P}y$, $S[\underline{f}S]$, $\sim \underline{Q}[\underline{f}S]$.

Let $\mathfrak{M}$ be a mating of a quantifier-free nnf G. We say that $\mathfrak{M}$ is *p-acceptable* (path-acceptable) iff every vertical path through G contains a mated pair of literal-occurrences. Every p-acceptable mating is a refutation mating. The converse is not quite true, since the mating of Fig. 5 is a refutation mating, but is not p-acceptable. However the mating in Fig. 6 is p-acceptable. This illustrates the following theorem:

Theorem 2. Let G be a quantifier-free nnf. G has a p-acceptable mating iff G has a refutation mating.

Combining Theorems 1 and 2, we have:

Theorem 3. Let D be a universal sentence of first order logic in nnf. D has no model iff some amplification of D has a p-acceptable mating.

Fig. 7 exhibits a p-acceptable mating of line (4) of Example 1 (which was displayed in Fig. 4). Thus we see that line (3) of Example 1 has no model, and line (1) is valid. The advantages of seeking a p-acceptable mating of Fig. 4 over trying to derive □ from clauses (c1)-(c16) in Fig. 1 seem evident. A computer program which tried to refute the clauses of Example 1 by the methods of [1] repeatedly

Fig. 5

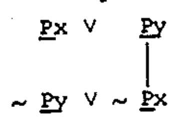


Fig. 6

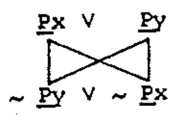
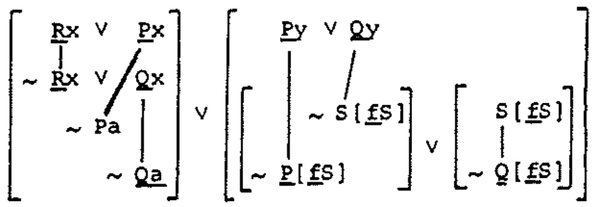


Fig. 7



ran out of storage space, but a modified version of it with smaller storage allocations quickly found a p-acceptable mating for this problem.

§4 A Refutation Procedure

We now provide a general outline of a refutation procedure based on p-acceptability. For the sake of generality and expository simplicity, the basic procedure described here will be rather naive, although we shall offer a few suggestions about ways in which it could be made more sophisticated in an actual implementation. Our main purpose here is to provide a framework for discussion of ideas, and a starting point for future research.

In particular, we shall speak as though unifying substitutions are always to be directly computed and applied to the appropriate wffs. In an actual implementation one should consider more sophisticated ways of handling substitutions, as discussed in [13], [14], and [5]. We have already mentioned that unification may be performed in parallel with other processes, and this seems essential when one deals with wffs of higher order logic, where unifying substitutions may not be unique, and the unification algorithm may not terminate. However, we shall regard these matters as implementation details beyond the scope of the present discussion.

Choices must be made at various points in the procedure described below, and appropriate heuristics must be used to make these choices. Although the success of a working program will depend crucially on these heuristics, we shall say little about them here, since not enough experience is yet available to justify a preference for one heuristic over another.

The fundamental data structures involved in the refutation procedure are the following:

(1) The wff B to be proved.
(2) A normal universal sentence D in nnf which is produced from B by Step 1 below, and thereafter remains fixed. D is called the initial wff of the refutation process.
(3) A normal universal sentence F in nnf obtained from D by quantifier duplications and alphabetic changes of bound variables. F changes as the procedure progresses. The quantifiers of F simply serve as markers to facilitate additional quantifier duplications, and will be ignored much of the time. Thus we shall regard F as an amplification of D. In an obvious sense, each literal-occurrence of F corresponds to a literal-occurrence of D.
(4) A mating $\mathfrak{m}$ of F, and the associated substitution $\theta_{\mathfrak{m}}$.
(5) A connection graph $C(D)$ of D.
(6) A connection graph $C(\mathfrak{m},F)$ of F relative to $\mathfrak{m}$.

The connection graphs are defined as follows. Let F be a nnf, and let M and N be in $\mathcal{L}(F)$. M and N are potential mates with respect to a mating $\mathfrak{m}$ of F iff some vertical path contains both M and N, and there is a substitution σ such that $\sigma(\theta_{\mathfrak{m}}N) = \sim \sigma(\theta_{\mathfrak{m}}M)$. We define

$$C(\mathfrak{m},F) = \{(M,N) \in \mathcal{L}(F) \times \mathcal{L}(F) \mid M \text{ and } N \text{ are potential mates with respect to } \mathfrak{m}\}.$$

This is a binary relation, which can be represented as a graph, as in [10]. We define $C(D)$ to be the connection graph of D with respect to the empty mating.

THE REFUTATION PROCEDURE

The refutation procedure is summarized in Fig. 8.

We are given a wff B which we wish to show is valid. Let C be a negation normal form of $\sim \overline{B}$, where $\overline{B}$ is the universal closure of B.

Step 1. Preprocessing.

(1a) Simplify and normalize C, so as to obtain a normal sentence C' in nnf which is provably equivalent to C, while minimizing the number of literal-occurrences in C', and also minimizing the degrees of the Skolem functions to be introduced in Step 1b. Choose between alternative simplified forms.

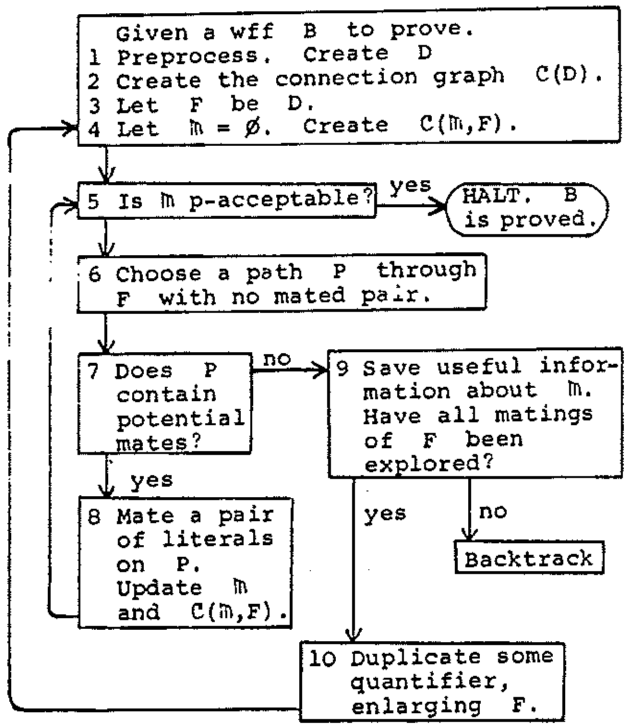
Thus a wf part of C of the form $\forall w\, \forall x\, \exists y\, \exists z[\underline{P}wy \wedge \underline{Q}xz]$ should be replaced by $\forall w\, \forall x[\exists y\, \underline{P}wy \wedge \exists z\, \underline{Q}xz]$. However, it serves no purpose to replace $\forall w\, \exists y\, \exists z[\underline{P}wy \wedge \underline{Q}yz]$ by $\forall w\, \exists y[\underline{P}wy \wedge \exists z\, \underline{Q}yz]$, since in each case the Skolemized form is $\forall w[\underline{P}w(\underline{f}w) \wedge \underline{Q}(\underline{f}w)(\underline{g}w)]$.

(1b) Skolemize to eliminate existential quantifiers. To do this, proceeding sequentially from left to right, replace each wf part $\exists yM$ of C' by the result of substituting $\underline{f}x_1 \ldots x_n$ for y in M, where $x_1,\ldots,x_n$ are the free variables of $\exists yM$, and $\underline{f}$ is a new n-ary function constant. (If $n = 0$, $\underline{f}$ is a new individual constant.)

(1c) Push in universal quantifiers, so as to obtain a provably equivalent normal sentence in nnf in which the scopes of universal quantifiers are as small as possible. For example, replace $\forall x\, \forall y[\underline{P}xy \vee \underline{Q}y]$ by $\forall y[\forall x\, \underline{P}xy \vee \underline{Q}y]$.

Fig. 8

THE REFUTATION PROCEDURE


Given a wff B to prove.
1 Preprocess. Create D
2 Create the connection graph C(D).
3 Let F be D.
4 Let 𝔪 = ∅. Create C(𝔪,F).
5 Is 𝔪 p-acceptable?
yes
HALT. B is proved.
6 Choose a path P through F with no mated pair.
7 Does P contain potential mates?
no
9 Save useful information about 𝔪. Have all matings of F been explored?
yes
8 Mate a pair of literals on P. Update 𝔪 and C(𝔪,F).
yes
no
Backtrack
10 Duplicate some quantifier, enlarging F.

Thus, if C contains a wf part of the form $\forall x\, \exists y[\underline{P}xy \wedge \underline{Q}xy]$, this is replaced by $\forall x[\underline{P}x(\underline{f}x) \wedge \underline{Q}x(\underline{f}x)]$ in Step (1b), and by $\forall x\, \underline{P}x(\underline{f}x) \wedge \forall z\, \underline{Q}z(\underline{f}z)$ in Step (1c).

(1d) Let the sentence obtained by Step 1 be called D.

<u>Step 2</u>. Create the connection graph C(D).

In an actual implementation of this procedure, it might be useful not to create C(D) all at once, but to compute and store parts of it as the information is needed. The important point is to avoid the necessity of computing this information more than once. A similar comment applies to C(𝔪,F), below.

<u>Step 3</u>. Let F be D.

<u>Step 4</u>. Let 𝔪 be the empty mating of F. Create C(𝔪,F), using C(D).

<u>Step 5</u>. Test 𝔪; is 𝔪 p-acceptable? If so, the refutation is complete. Otherwise, continue to Step 6.

We remark that since the number of vertical paths in a wff can be quite large, considerable attention should be paid to the efficiency with which this test is carried out. Of course, one need not examine the vertical paths separately and completely. As soon as one has found a pair of mated literals on a sub-path of a vertical path, one can exclude from further consideration all extensions of that sub-path. Also, since 𝔪 is tested and constructed in stages, information about which vertical paths contain mated pairs can be saved from one stage to the next. At each stage, all one really does is search for one new vertical path which does not contain any mated pair, starting where one left off at the last stage.

Various methods from propositional calculus can be used to temporarily simplify F as the construction of 𝔪 progresses, and we digress briefly to discuss one of these. Let M be the wff

$$([L_1 \wedge P_1] \vee \ldots \vee [L_n \wedge P_n]) \wedge$$
$$([L_1^{\prime} \wedge \ldots \wedge L_n^{\prime} \wedge Q] \vee R)$$

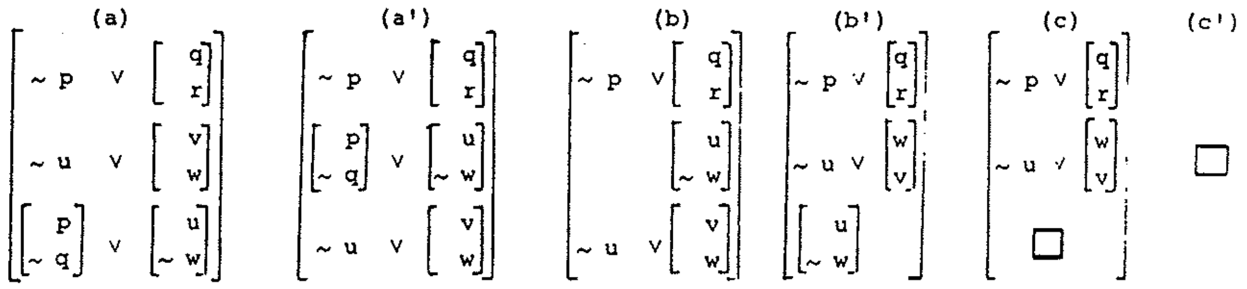
where $n \geq 1$, the L_i and $L_i^{\prime}$ are literals, and the P_i, Q, and R are arbitrary wffs; in particular, they may be the empty conjunction $\triangle$ (which is true), or the empty disjunction $\Box$ (which is false). Let N be the wff $([L_1 \wedge P_1] \vee \ldots \vee [L_n \wedge P_n]) \wedge R$. Suppose F contains an occurrence of M, and F* is obtained from F by replacing that occurrence of M by an occurrence of N. (See Fig. 9) (Note that if $L_i^{\prime} = \sim L_i$ for each i, then $\models F \equiv F^*$.) Let 𝔪 be a mating of F such that for each $i \leq n$, $L_i \mathfrak{m} L_i^{\prime}$ or $L_i^{\prime} \mathfrak{m} L_i$. We may regard $\mathcal{L}(F^*)$ as a subset of $\mathcal{L}(F)$, and define $\mathfrak{m}^*$ to be the restriction of 𝔪 to $\mathcal{L}(F^*)$. Then $\mathfrak{m}^*$ is a mating of F*. It is easy to see that there is a p-acceptable mating of F which is an extension (superset) of 𝔪 if and only if there is a p-acceptable mating of F* which is an extension of $\mathfrak{m}^*$. Thus we may reduce F and 𝔪 to F* and $\mathfrak{m}^*$ in our search for a p-acceptable mating (though F and 𝔪 may have to be restored later if no p-acceptable extension of $\mathfrak{m}^*$ is found).

Of course, we shall feel free to use elementary laws of conjunction and disjunction, such as commutativity, associativity, $M \vee \Box \equiv M$, $M \wedge \Box \equiv \Box$, and

Fig. 9

$$\begin{array}{cc} F \text{ contains} & F^* \text{ contains} \\ \begin{bmatrix} \begin{bmatrix} L_1 \\ P_1 \end{bmatrix} \vee \ldots \vee \begin{bmatrix} L_n \\ P_n \end{bmatrix} \\ \begin{bmatrix} L_1^{\prime} \\ \cdot \\ \cdot \\ \cdot \\ L_n^{\prime} \\ Q \end{bmatrix} \vee \quad R \end{bmatrix} & \begin{bmatrix} \begin{bmatrix} L_1 \\ P_1 \end{bmatrix} \vee \ldots \vee \begin{bmatrix} L_n \\ P_n \end{bmatrix} \\ \\ R \\ \\ \end{bmatrix} \end{array}$$

Fig. 10



$M \wedge \Delta \equiv M$. Using these laws and reductions of the sort just discussed, we can often drastically simplify F, or even reduce it to $\Box$, which has a p-acceptable mating since it has no vertical paths. This is illustrated in Figures 10 and 11, where we assume complementary literals are mated. In both figures (a') is obtained from (a) by elementary laws, (b) is obtained from (a') by a reduction, etc. Dotted lines surround the parts of Fig. 11 involved in the reduction steps. Fig. 10a represents the mating for Example 2 at the bottom of Fig. 2, and Fig. 11a represents the mating for Example 1 in Fig. 7.

<u>Step 6</u>. Choose a vertical path P through F with no mated pair.

Presumably at least one such path can be found in Step 5. However, it may be worthwhile to choose this path carefully. For example, with the aid of the connection graph $C(\mathfrak{m},F)$ one can seek such a path with a minimum number of pairs of potential mates.

<u>Step 7</u>. Is there at least one pair of potential mates on P? If so, go to Step 8. If not, go to Step 9, and give P high priority for consideration by Step 6 at later stages of the procedure.

<u>Step 8</u>. Choose a pair (M,N) of potential mates on P, and replace $\mathfrak{m}$ by $\mathfrak{m} \cup \{(M,N)\}$. (We call this process <u>mating</u> M and N.) Adjust $\theta_{\mathfrak{m}}$ and $C(\mathfrak{m},F)$ appropriately. Return to Step 5.

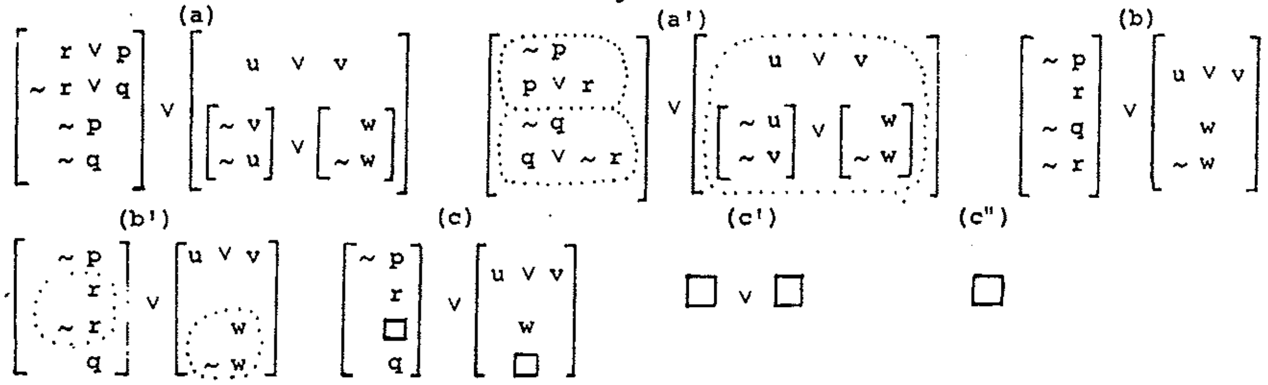
We remark that in Steps 5-8 we simply check whether $\mathfrak{m}$ is acceptable, and change it if it is not. The particular way we change it is guided by our criterion of acceptability. We add a pair of literals to $\mathfrak{m}$ because it makes $\mathfrak{m}$ closer to being acceptable.

Of course, if an inappropriate choice is made at Step 8, the system will eventually have to backtrack. To alleviate this problem, more sophisticated methods of deciding how to alter $\mathfrak{m}$ than those implicit in Steps 6-8 may be needed. We briefly mention one such method, which was suggested by Eve Cohen.

Let us say that we <u>fix</u> a vertical path P by mating a pair of literal-occurrences on P. While there may be many ways of fixing a given path, some of these may not be compatible with any way of fixing another path (because the associated substitutions are not compatible), and so should not be used. By considering all possible ways of simultaneously fixing all paths in a set of vertical paths, starting with a unit set and progressively enlarging it, one can eliminate many inappropriate ways of fixing paths. If, within the computational resources allocated, one can eliminate all but one way of simultaneously fixing all paths in the set, one can fix all those paths simultaneously without fear of error.

We can also apply a heuristic to Steps 8 and 6 which is analogous to the Set-of-Support strategy [15] for resolution. Not all literal-occurrences need have mates in a p-acceptable mating (particularly if inappropriate quantifier duplications have occurred), but it is sometimes clear from the statement of a theorem that certain literal-occurrences

Fig. 11



must have mates, or that most literal-occurrences in a certain set must have mates. In such cases, high priority should be given to mating such literal-occurrences.

Step 9. Have all matings of F been explored?

If so, go to Step 10. If not, backtrack and try another mating. In either case, record information about the current mating which may eventually be useful in choosing a quantifier to duplicate, or in reconstructing this mating after a quantifier has been duplicated.

Step 10. Choose a quantifier of F, duplicate it, normalize the resulting wff, and call it F. Go to Step 4.

The choice of a quantifier to duplicate should be made as intelligently as possible, since every quantifier duplication enlarges the wff with which the mating program must deal. Nevertheless, an inappropriate quantifier duplication does not prevent an acceptable mating from eventually being found, so we never backtrack past this point. Of course, the heuristic used to choose quantifiers to duplicate should be designed so that the procedure will be complete.

Sometimes it may be desirable to go from Step 9 to Step 10 even if further matings of F remain to be explored, or to go from Step 10 directly to Step 7, and continue work on the current mating in the enlarged wff. Of course, such acts may complicate backtracking, and necessitate special measures to assure the completeness of the procedure.

§5 Remarks

A computer program based on these ideas has been constructed, largely by Eve Cohen. It handles sentences of type theory as well as those of first order logic, but of course in automatic mode it is not logically complete for type theory. With its aid various heuristics for the refutation procedure are being investigated.

REFERENCES

[1] Peter B. Andrews, "Refutations by Matings", IEEE Transactions on Computers C-25 (1976), 801-807.

[2] Peter B. Andrews and Eve Longini Cohen, "Theorem Proving in Type Theory", 5th International Joint Conference on Artificial Intelligence, MIT, August 1977, 566.

[3] W. Bibel, "Tautology Testing with an Improved Matrix Reduction", Technische Universität München, Institute für Informatik, report Tum-Info-7706, 1977, 22 pp.

[4] Alonzo Church, Introduction to Mathematical Logic, Vol. I, Princeton University Press, 1956.

[5] Philip T. Cox, "Locating the Source of Unification Failure", Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence, Toronto, July 1978, 20-29.

[6] Martin Davis and Hilary Putnam, "A Computing Procedure for Quantification Theory", J.A.C.M. 7 (1960), 201-215.

[7] Martin Davis, "Eliminating the Irrelevant from Mechanical Proofs", Experimental Arithmetic, High Speed Computing and Mathematics. Proceedings of Symposia in Applied Mathematics XV, American Mathematical Society, 1963, 15-30.

[8] Paul C. Gilmore, "A Proof Method for Quantification Theory: Its Justification and Realization", IBM Journal of Research and Development 4 (1960), 28-35.

[9] Lawrence J. Henschen, "Theorem Proving by Covering Expressions", J.A.C.M., to appear.

[10] Robert Kowalski, "A Proof Procedure Using Connection Graphs", J.A.C.M. 22 (1975), 572-595.

[11] Dag Prawitz, "A Proof Procedure with Matrix Reduction", Symposium on Automatic Demonstration, Lecture Notes in Mathematics 125, Springer-Verlag, 1970, 207-214.

[12] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", J.A.C.M. 12 (1965), 23-41.

[13] Sharon Sickel, "A Search Technique for Clause Interconnectivity Graphs", IEEE Transactions on Computers C-25, 1976, 823-835.

[14] Jophien van Vaalen, "An Extension of Unification to Substitutions with an Application to Automatic Theorem Proving", Fourth International Joint Conference on Artificial Intelligence, 1975, 77-81.

[15] L. Wos, G. A. Robinson and D. F. Carson, "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving", J.A.C.M. 12 (1965), 536-541.

NON-MONOTONIC LOGIC I
(Extended Abstract)

Drew McDermott
Department of Computer Science
Yale University
New Haven, Connecticut 06520

Jon Doyle
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract: "Non-monotonic" logical systems are logics in which the introduction of new axioms can invalidate old theorems. Such logics are very important in modeling the beliefs of active processes which, acting in the presence of incomplete information, must make and subsequently revise predictions in light of new observations. We present the motivation and history of such logics. We develop model and proof theories, a proof procedure, and applications for one important non-monotonic logic. The main results are the completeness of the non-monotonic predicate calculus and the decidability of the non-monotonic sentential calculus. We also discuss characteristic properties of this logic. This paper is an abbreviated version of MIT AI Memo 486, in which all proofs are given. The full paper also discusses the relationship of non-monotonic logic to stronger logics, logics of incomplete information, and truth maintenance systems.

Acknowledgements: We wish to thank Gerald Jay Sussman, Rohit Parikh, David Harel, Mitch Marcus, Lucia Vaina, and Vaughan Pratt for helpful criticisms and comments. Drew McDermott acknowledges the support of a Josiah Willard Gibbs instructorship. Jon Doyle thanks the Fannie and John Hertz Foundation for supporting his research with a graduate fellowship. This research was conducted in part at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643, and in part by NSF Grant MCS77-04828.

The Problem of Incomplete Knowledge

The relation between formal logic and the operation of the mind has always been unclear. Some of the more striking differences between properties of formal logics and mental phenomenology occur in situations dealing with perception, ambiguity, common-sense, causality and prediction. One common feature of these problems is that they seem to involve working with incomplete knowledge. Perception must account for the noticing of overlooked features, common-sense ignores myriad special exceptions, assigners of blame can be misled, and plans for the future must consider never-to-be-realized contingencies. It is this apparently unavoidable making of mistakes in these cases that leads to some of the deepest problems of the formal analysis of mind.

Some studies of these problems occur in the philosophical literature, the most relevant here being Rescher's [1964] analysis of counterfactual conditionals and belief-contravening hypotheses. In artificial intelligence, studies of perception, ambiguity and common-sense have led to knowledge representations which explicitly and implicitly embody much information about typical cases, defaults, and methods for handling mistakes. [Minsky 1974, Reiter 1978] Studies of problem-solving and acting have attempted representing predictive and causal knowledge so that decisions to act require only limited contemplation, and that actions, their variations, and their effects can be conveniently described and computed. [Hayes 1970, 1971, 1973, Doyle 1978] Indeed, one of the original names applied to these efforts, "heuristic programming", stems from efficiency requirements forcing the use of methods which occasionally are wrong or which fail. The possibility of failure means that formalizations of reasoning in these areas must capture the process of revisions of perceptions, predictions, deductions and other beliefs.

In fact, the need to revise beliefs also occurs in deductive systems working within traditional logics. Much work has been done on mechanized proof techniques for the first-order predicate calculus. [Robinson 1965, Nevins 1974, Moore 1975] Incomplete information is represented in these systems as disjunctions of the several possibilities where the individual disjuncts may be independent of the axioms being used, that is, cannot be proven or contradicted by arguments from the axioms. Thus, proof procedures engage in *case-splitting*, in which disjuncts are considered in a case-by-case fashion. At any given time, the proof procedure will have some set of current assumptions, from which the current set of formulas has been derived. If failures in the proof attempt lead to investigating new splits, and so change the set of current assumptions, the current set of derived formulas must also be updated, for it is the current set of formulas on which the proof procedure bases its actions.

Classical symbolic logic lacks tools for describing how to revise a formal theory to deal with inconsistencies caused by new information. This lack is due to a recognition that the general problem of finding and selecting among alternate revisions is very hard. (For an attack on this problem, see Rescher [1964]. Quine and Ullian [1978] survey the complexities.) Although logicians have been able to ignore this problem, philosophers and researchers in artificial intelligence have been forced to face it because humans and computational models are subject to a continuous flow of new information. One important insight gained through computational experience is that there are at least two different problems involved, what might be called "routine revision" and "world-model reorganization".

World-model reorganization is the very hard problem of revising a complex model of a situation when it turns out to be wrong. Much of the complexity of such models usually stems from parts of the model relying on descriptions of other parts of the model, such as inductive hypotheses, testimony, analogy, and intuition. An example of such large-scale reorganization would be the revision of a Newtonian cosmology to account for perturbations in Mercury's orbit. Less grand examples are children's revisions of their world-models as discovered by Piaget, and the revision of one's opinion of a friend upon discovering his dishonesty.

Routine revision, on the other hand, is the problem of maintaining a set of facts which, although expressed as universally true, have exceptions. For example, a program may have the belief that all animals with beaks are birds. Telling this program about a platypus will cause a contradiction, but intuitively not as serious a contradiction as those requiring total reorganization. The relative simplicity of this type of revision problem stems from the statement itself expressing what revisions are appropriate by referring to possible exceptions. Such relatively easy cases include many forms of inferences, default assumptions, and observations.

Classical logics, by lumping all contradictions together, has overlooked the possibility of handling the easy ones by expanding the notation in which rules are stated. That is, we could have avoided this problem by stating the belief as "If something is an animal with a beak, then *unless proven otherwise*, it is a bird." If we allow statements of this kind, the problem becomes how to coordinate sets of such rules. Each such statement may be seen as providing a piece of advice about belief revision; for our approach to make sense, all the little pieces of advice must determine a unique revision. This is the subject of this paper. Of course, even if we are successful, the world-model reorganization problem will still be unsolved. But we hope factoring out the routine revision problem will make the more difficult problem clearer.

Approaches to Non-Monotonic Logic and the Semantical Difficulties

The study of the problem of formalizing the process of revision of beliefs has been almost completely confined to the practical side of artificial intelligence research, where much work has been done. [Hewitt 1972, McDermott 1974, Stallman and Sussman 1977, Doyle 1978] Theoretical foundations for this work have been lacking. This paper studies the foundations of these forms of reasoning with revisions which we term *non-monotonic* logic.

Traditional logics are called *monotonic* because the theorems of a theory are always a subset of the theorems of any extension of the theory. (This name for this property of classical logics was used, after a suggestion by Pratt, in Minsky's [1974] discussion. Hayes [1973] has called this the "extension" property.) In this paper, by *theory* we will mean a set of axioms. A more precise statement of monotonicity is this: If A and B are two theories, and $A \subseteq B$, then $Th(A) \subseteq Th(B)$, where $Th(S) = \{p : S \vdash p\}$ is the set of theorems of S. We will be even more precise about the definition of $\vdash$ later.

Monotonic logics lack the phenomenon of new information leading to a revision of old conclusions. We obtain non-monotonic logics from classical logics by extending them with a modality ("consistent") well-known in artificial intelligence circles, and show that the resulting logics have well-founded, if unusual, model and proof theories. We introduce the proposition-forming modality M (read "consistent"). Informally, Mp is to mean that p is consistent with everything believed. (See [McCarthy and Hayes 1969].) One small theory employing this modality is

(1) noon $\wedge$ M[sun-shining] $\supset$ sun-shining

(2) noon

(3) eclipse $\supset$ $\neg$sun-shining,

in which we can prove

(4) sun-shining.

If we add the axiom

(5) eclipse

then (4) is inconsistent, so (4) is not a theorem of the extended theory.

The use of non-monotonic techniques has some history, but until recently the intuitions underlying these techniques were inadequate and led to difficulties involving the semantics of non-monotonic inference rules in certain cases. We mention some of the guises in which non-monotonic reasoning methods and belief revising processes have appeared.

In PLANNER [Hewitt 1972], a programming language based on a negationless calculus, the THNOT primitive formed the basis of such reasoning. THNOT, as a goal, succeeded only if its argument failed, and failed otherwise. Thus if the argument to THNOT was a formula to be proved, the THNOT would succeed only if the attempt to prove the embedded formula failed. In addition to the non-monotonic primitive THNOT, PLANNER employed antecedent and erasing procedures to update the data base of statements of beliefs when new deductions were made or actions taken. Unfortunately, it was up to the user of these procedures to make sure that there were no circular dependencies or mutual proofs between beliefs. Such circularities could lead to, for example, errors of groundless belief (due to two mutually supporting beliefs) or non-terminating programs (a more technical but no less irritating problem).

Two related forms of non-monotonic deductive systems are those described by McCarthy and Hayes [1969] and Sandewall [1972]. McCarthy and Hayes give some indications of how actions might be described using modal operators like "normally" and "consistent", but present no detailed guidelines on how such operators might be carefully defined. Sandewall, in a deductive system applied to the frame problem (which is basically the problem of efficiently representing the effects of actions; see [Hayes 1973]) used a deductive representation of non-monotonic rules based on a primitive called UNLESS. This was used to deduce conditions of situations resulting from actions except in those cases where properties of the action changed the extant conditions. Thus one might say that things retain their color unless painted.

Sandewall's interpretation of UNLESS was in accord with then current intuitions: UNLESS(p) is true if p is not deducible from the axioms using the classical first-order inference rules. Unfortunately, this definition has several problems, as pointed out by Sandewall. One problem is that it can happen that both p and UNLESS(p) are deducible, since from a rule like "from UNLESS(C) infer D" D can be inferred, but at the same time UNLESS(D) is also deducible since D is not deducible by classical rules. These problems are partly due to the dependence of the notion of "deducible" on the intention of deduction rules based on "not deducible". This question-begging definition leads to perplexing questions of beliefs when complicated relations between UNLESS statements are present. For example, given the axioms

A
$A \wedge \text{Unless}(B) \supset C$
$A \wedge \text{Unless}(C) \supset B$,

we are faced with the somewhat paradoxical situation that either B or C can be deduced, but not both simultaneously. On the other hand, in the axiom system

A
$A \wedge \text{Unless}(B) \supset C$
$A \wedge \text{Unless}(C) \supset D$
$A \wedge \text{Unless}(D) \supset E$,

one would expect to see A, C and E believed, and B and D not believed.

One might be tempted to dismiss these anomalous cases as uninteresting. In fact, such cases are not perverse; rather, they occur naturally and are very important in many applications. One common way they are introduced is by employing assumptions which require further assumptions to be made. Of course, such hierarchical relations between choices can be avoided in any fixed theory by rephrasing the system in terms of one universal state variable, but such a solution is practically undesirable and inefficient. Instead, it is necessary to employ systems which allow such patterns of dependency relationships to occur.

Spurred by Sandewall's presentation of the problems arising through such non-monotonic inference rules, Kramosil [1975] considered sets of inference rules of the form

"From $\vdash p$, $\nvdash q$, infer $\vdash r$",

where $\vdash$ and $\nvdash$ are tokens of the meta-language and the number of antecedents can be arbitrary. Kramosil defined the set of theorems in such a system as the intersection of all subsets of the language closed under the inference rules. He noted that this set may not itself be closed under the inference rules, and showed that in the special case in which the inference rules preserve truth values (that is, are effectively monotonic) that if the set of theorems of the monotonic inference rules alone is also closed with respect to the non-monotonic inference rules, then this set is the set of non-monotonic theorems. Kramosil's conclusion was that a set of inference rules defines a formalized theory (one in which all formulas have a well-defined truth value) if and only if this same theory is that of the monotonic inference rules alone, which he interprets to mean that the non-monotonic rules are either useless or meaningless.

As we will show in this paper, Kramosil's interpretation was too pessimistic with regard to the possibility of formalizing such rules and their unusual properties. As we have argued above, the purpose of non-monotonic inference rules is not to add certain knowledge where there is none, but rather to guide the selection of tentatively held beliefs in the hope that fruitful investigations and good guesses will result. This means that one should not *a priori* expect non-monotonic rules to derive valid conclusions independent of the monotonic rules. Rather one should expect to be led to a set of beliefs which while perhaps eventually shown incorrect will meanwhile coherently guide investigations.

Non-monotonic inference rules need not appear in the explicit forms discussed by Kramosil. Many authors have described artificial intelligence programs which exhibit non-monotonic behavior only implicitly. Non-monotonicity in these systems stems typically from extra-logical devices like conflict resolution strategies, which use production-rule orderings and specificity criteria to determine the next system action. Pratt [1977] and Joshi [1978] term this property of their systems "non-monotonicity".

One class of non-monotonic inferences consist of what might be called "minimal" inferences, in which a minimal model for some set of beliefs is assumed by assuming the set of beliefs to be a complete description of a state of affairs. Joshi and Rosenschein [1975] describe a partial-matching procedure based on the operation of taking least upper bounds in a lattice of sets of beliefs. This has the effect of assuming just enough additional information to allow a desired partial match to succeed. McCarthy [1977] outlines a procedure called "circumscription", in which the current partial extension of some predicate is assumed to be the complete extension. Of course, new examples of the predication can invalidate previous completeness assumptions. Reiter [1977] analyzes the related technique of assuming false all elementary predications not explicitly known true. He outlines some conditions under which data bases remain consistent under this "closed world assumption", and shows certain forms of data bases to be naturally consistent with this assumption. However, the closed world assumption does not seem to allow for any locality of definition of defaults, since it applies this assumption to all primitive predicates, and does not allow defaults applied to defined predicates. Circumscription, on the other hand, would seem to be applicable to any predicate whatever. Although they describe tools for non-monotonic reasoning, none of these authors discuss the problem of revision of beliefs.

These problems were mostly resolved in the Truth Maintenance System (TMS) of Doyle [1978] and subsequent related systems [London 1977, McAllester 1978] in which each statement has an associated set of justifications, each of which represents a reason for holding the statements as a belief. These justifications are used to determine the set of current beliefs by examining the recorded justifications to find well-founded support (non-circular proofs) whenever possible for each belief. When hypotheses change, these justifications are again examined to update the set of current beliefs. This scheme provides a more accurate version of antecedent and erasing procedures of PLANNER without the need to explicitly check for circular proofs. The non-monotonic capability appears as a type of justification which is the static analogue of the PLANNER THNOT primitive. Part of the justification of a belief can be the lack of valid justifications for some other possible program belief. This allows, for example, belief in a statement to be justified whenever no proof of the negation of the statement is known. This representation of non-monotonic justifications, in combination with the belief revision algorithms, produced the first system capable of performing the routine revision of apparently inconsistent theories into consistent theories. Part of this revision process is a backtracking scheme called dependency-directed backtracking. [Stallman and Sussman 1977] We will analyze this system in more detail later, but first we provide some theoretical foundations for this work.

In outline, our analysis of these questions will proceed as follows. We first define a standard language of discourse including the non-monotonic modality M ("consistent"). The semantics of the language is based on models constructed from *fixed points* of a formalized non-monotonic proof operator. Provability in this system is then defined, and a proof of completeness for this system is presented. This is augmented by a proof procedure for a restricted class of theories and an analysis of some of the structure of models of non-monotonic theories.

Linguistic Preliminaries

We settle on a language L which will be the language of all theories mentioned in the following. L has an infinite number of constant letters, variable letters, predicate letters, and propositional constant letters. The formation rules of the language are as follows:

The *atomic formulas* of L are the propositional constant letters and the strings of the form $g(x_1,\ldots,x_n)$ for predicate letter g and variables or constants $x_1, \ldots, x_n$. The *formulas* of L are either atomic formulas or, for formulas p, q and variable letter x, strings of the form Mp, $\neg p$, $p \supset q$, and $\forall xp$. We use the usual abbreviations of $p \wedge q$ for $\neg[p \supset \neg q]$, $p \vee q$ for $\neg p \supset q$, $\exists xp$ for $\neg \forall x \neg p$, and abbreviate $\neg M \neg p$ as Lp. A *statement* is a formula with no free variables. The usual criteria for determining free variables apply (see [Mendelson 1964]). In addition, a variable x is free in Mp if and only if x is free in p.

In this paper, the letters C, D, E and F will be used as syntactic variables ranging over propositional constant letters. The letters p, q and r will be used for formulas. Implicit quasi-quotation is used throughout.

The inferential system used defines a first-order theory to be a set of axioms including the following infinite class of axioms: For all formulas p, q and r:

(6) (i) $p \supset [q \supset p]$

(ii) $[p \supset [q \supset r]] \supset [[p \supset q] \supset [q \supset r]]$

(iii) $[\neg q \supset \neg p] \supset [[\neg q \supset p] \supset q]$

(iv) $\forall xp(x) \supset p(t)$

where $p(x)$ is a formula and t is a constant or a variable free for x in $p(x)$ and $p(t)$ denotes the result of substituting t for every free occurrence of x in $p(x)$, and

(v) $\forall x[p \supset q] \supset [p \supset \forall xq]$

if p is a formula containing no free occurrence of x. (These axioms are from [Mendelson 1964].) These are the *logical* axioms. All other axioms are called *proper*, or non-logical axioms. The theory with no proper axioms is called the *predicate calculus* (PC). (Note that this theory also contains strings containing the letter M, so it is actually not strict PC.) The *sentential calculus* (SC) consists of axioms which are instances of (i), (ii) and (iii) only. A theory consisting only of the sentential calculus plus a finite number of statements is called a *statement theory*.

In this paper, the letters A and B will be used to stand for theories.

Proof-Theoretic Operators

The monotonic rules of inference we will use (also from [Mendelson 1964]) are

(7) Modus Ponens: from p and $p \supset q$, infer q

Generalization: from p, infer $\forall xp$.

If S is a set of formulas, and p follows from S and the axioms of A by the rules (7), we say $S \vdash_A p$. We abbreviate $\vdash_{PC}$ by $\vdash$ alone. We define $Th(S) = \{p : S \vdash p\}$.

The particular inference rules (7) are not very important. Later in the paper, when we concentrate on statement theories, the rule of generalization will be dropped without much fanfare. All that is important is that the operator Th have the following properties, which together are called *monotonicity*:

(8) (i) $A \subseteq Th(A)$

(ii) If $A \subseteq B$, then $Th(A) \subseteq Th(B)$,

and the property (9) of *idempotence*

(9) $Th(Th(A)) = Th(A)$.

Clearly, any classical inference system satisfies these conditions. Condition (9) can also be viewed as a fixed point equation, stating that the set of theorems monotonically derivable from a theory is a fixed point of the operator which computes the closure of a set of formulas under the monotonic inference rules. A well-known property of the monotonic inference rules is that Th(A) is the smallest fixed point of this closing process; in fact, that Th(A) is the intersection of all S such that $A \subseteq S$ and $Th(S) = S$.

In order to deal with non-monotonic logic, we need a new inference rule like this one (which we will take back immediately):

(10) "If $\nvdash_A \neg p$, then $\vdash_A$ Mp."

That is, if a formula's negation is not derivable, it may be inferred to be consistent. As it stands, however, this rule is of no value because it is circular. "Derivable" means "derivable from axioms by inference rules", so we cannot define an inference rule in terms of derivability so casually.

Instead, we retain the definition of $\vdash$ as meaning monotonic derivability, and define the operator NM as follows: for any first-order theory A and any set of formulas $S \subseteq L$ (L, recall, is the entire language), let

(11) $NM_A(S) = Th(A \cup As_A(S))$,

where $As_A(S)$, the set of *assumptions* from S, is given by

(12) $As_A(S) = \{Mq : q \in L \text{ and } \neg q \notin S\} - Th(A)$.

Notice that theorems of A of the form Mq are never counted as assumptions. NM_A takes a set S and produces a new set which includes Th(A) but also includes much more: everything provable from the enlarged set of axioms and assumptions which is the original theory together with all assumptions not ruled out by S. We would like to define TH(A), the set of theorems non-monotonically derivable from A, by analogy with the monotonic case as

(13) "TH(A) = the smallest fixed point of NM_A."

This "definition" tries to capture the idea of adding the non-monotonic inference rule (10) to a first-order theory A. This is plausible, since it demands a set such that all of its elements may be proven from axioms and assumptions not wiped out by the proofs. Unfortunately, there is in general no appropriate fixed point of NM_A. It can happen that a theory has no fixed point under the operator NM_A. Even if there are fixed points, there need not be a smallest fixed point.

For example, consider the theory T1 obtained as

(14) $T1 = PC \cup \{ MC \supset \neg D, MD \supset \neg C \}$,

where C and D are propositional constants. NM_{T1} has two fixed points, which can be called F1 and F2. F1 contains ¬C but not ¬D, and F2 contains ¬D but not ¬C. Since ¬D is not in F1, MD is in F1, and so ¬C is in F1. Similarly, the presence of ¬D in F2 keeps ¬C out and MC in F2. The problem is that neither $F1 \cap F2$ nor $F1 \cup F2$ is a fixed point of NM_{T1}. Since neither ¬C nor ¬D is in $F1 \cap F2$, MC and MD are both in $NM_{T1}(F1 \cap F2)$, so ¬C and ¬D are in $NM_{T1}(F1 \cap F2)$, so $F1 \cap F2 \neq NM_{T1}(F1 \cap F2)$. Similarly, both ¬C and ¬D are in $NM_{T1}(F1 \cup F2)$, so applying NM_{T1} to the union results in a smaller set. So in this case there is no natural status for ¬C and ¬D.

An example of a theory with no fixed point of the corresponding operator is the theory T2 obtained as

(15) $T2 = PC \cup \{ MC \supset \neg C \}$.

In this case, NM_{T2} has no fixed point, since alternate applications of the operator to any set produce new sets in which either both MC and ¬C exist or neither exist.

Therefore, we must accept a somewhat less elegant definition of TH. Let us define TH as follows:

(16) $TH(A) = \cap(\{L\} \cup \{S : NM_A(S) = S\})$.

That is, the set of provable formulas is the intersection of all fixed points of NM_A, or the entire language if there are no fixed points. We will use the abbreviation $A \vdash p$ to indicate that $p \in TH(A)$. With this definition, neither MC nor MD is a theorem of T1 in (14), but MC∨MD is. In the following, we will abbreviate $\{S : NM_A(S) = S\}$ as FP(A), and (somewhat abusing the terms) call the elements of this set *fixed points* of the theory A.

This definition of the provable statements is quite similar in some respects to the definition of compatibility-restricted entailment given by Rescher [1964]. In that system, a set S of formulas is said to CR-entail a formula p if p follows in the standard fashion from each of one or more "preferred" maximal consistent subsets of S. In the present case, we obtain the preferred subsets of formulas as fixed points of the operator NM_A (the "compatible subsets"), but in contrast to normal deducibility where the empty set always suffices, there need not be any such subsets. This case produces the entire language as the set of provable formulas by vacuous fulfillment of the condition of derivability.

One unusual consequence of this definition of provability is that the deduction theorem does not hold for non-monotonic logic. For example, while $\{ C \} \vdash MLC$, it is not true that $\vdash C \supset MLC$. This failure of the deduction theorem is to be expected, however, since the non-monotonic provability of a formula depends on the completeness of the set of hypotheses, that is, on the fact that no other axioms are available. The deduction theorem, however, would if valid produce implications valid no matter what other axioms were added to the system, even if these axioms would invalidate the completeness condition used in the derivation of the implication. One should note that although the deduction theorem does not hold in general in non-monotonic logic, there are many particular cases in which it does hold. For instance, if some conclusion follows classically from some hypotheses, then the expected implication will also hold. In addition, not all properly non-monotonic theories are such that the deduction theorem fails. It is an interesting open problem to characterize the precise cases in which the deduction theorem is valid in non-monotonic theories.

So far, we have defined "provability" without defining "proof". For a formula to be provable in a theory, it must have a standard proof from axioms and assumptions in each fixed point of the theory, and, as yet, we have no way of enumerating fixed points or even of describing one. It is worth note that when a theory has more than one fixed point, the fixed points are inaccessible in the sense that the sequence $Th(A)$, $NM_A(Th(A))$, $NM_A(NM_A(Th(A)))$, ... does not converge to a fixed point. We have a proof, which we do not present here, that if NM_A has exactly one fixed point, then the fixed point is the limit of successive applications of NM_A to the sequence of sets starting with A. We will eventually attend to defining non-monotonic proof, but first we turn our attention to the topic of semantics.

Model Theory

An *interpretation* V of formulas over a language L is a pair $\langle X, U \rangle$, where X is a nonempty set, and U is a function which associates relations and values over the domain X with each predicate, variable, constant and propositional constant letter in the usual fashion. That is, for each n-ary predicate letter P, $U(P) \subseteq X^n$; for each variable or constant x, $U(x) \in X$; and for each propositional constant letter C, $U(C) \in \{0, 1\}$. Using this mapping function U we define the value V(p) of a formula p in the interpretation V to be an element of $\{0, 1\}$ satisfying the following conditions: For an atomic formula $p(x_1, ..., x_n)$, the value is 1 if $\langle U(x_1), ..., U(x_n) \rangle \in U(p)$, and is 0 otherwise. $V(\neg p) = 1$ if $V(p) = 0$, and is 0 otherwise. $V(p \supset q) = 1$ if either $V(p) = 0$ or $V(q) = 1$, and is 0 otherwise. $V(\forall xp) = 1$ if for all $y \in X$, $V'(p) = 1$, where $V' = \langle X\ [y/x]U \rangle$, where $[y/x]U$ is the mapping derived from U by changing its value at the point x to the value y. $V(\forall xp) = 0$ otherwise. If $V(p) = 1$, we say that V *satisfies* p, and write $V \vDash p$.

A *monotonic model* of a set of formulas $S \subseteq L$ is an interpretation V which satisfies each formula in S, that is, $V(p) = 1$ for each formula $p \in S$. A *non-monotonic model* of a theory A is a pair $\langle V, S \rangle$, where V is a monotonic model of S, and $S \in FP(A)$. When the context makes the intended meaning clear, we will use the term *model* of A to mean either a non-monotonic model, a monotonic model, or an element of FP(A) for the theory A.

Although unorthodox, this definition provides a meaning for formulas Mp which reflects the proof-theoretic property that "p is consistent with what is believed". This notion is made precise by including in the model a set of "current assumptions" (namely, $As_A(S)$). A model for a theory must assign 1 to all of these assumptions, so the effect is that Mp is assigned 1 in a model if $\neg p$ is not derivable and $\neg Mp$ is not derivable from the current assumptions and the original theory, that is, if p is consistent with what is "believed" in the model. Unfortunately, Mp may be assigned 1 in some model even when $\neg p$ is derivable (for example, when no axiom mentions Mp at all). This indicates that the logic is too weak. We discuss this problem in the full paper. We present a stronger logic, with a more elegant model theory, in a forthcoming paper.

Much of the unorthodoxy of this semantics stems from the nature of non-monotonicity itself. Because the intended meaning of the operator M makes reference to the other formulas of the theory, an unusual holistic semantics results in which the meanings of formulas involving M depend on the theory as a whole. Thus the semantics is quite unlike the Kripkean semantics developed for the standard modal logics.

With this definition of model, we can justify the definition of provability.

Theorem 1. (Soundness) If $A \vdash p$, then $V \models p$ for all models $\langle V, S \rangle$ of A.

Proof: Assume $A \vdash p$. If there are no models of A, the theorem follows trivially. Otherwise, p is a member of every fixed point of A. But since every model of A is a monotonic model of a fixed point of A, every model assigns 1 to p. ∎

Theorem 2. (Completeness) If $V \models p$ for all models $\langle V, S \rangle$ of A, then $A \vdash p$.

Proof: Assume that it is not true that $A \vdash p$. Thus there is a fixed point S of NM_A which does not contain p. Now $Th(S) = S$ by idempotence, so $S \nvdash p$. But the predicate calculus is complete, so some monotonic model V of S has $V(p) = 0$. ∎

It is not surprising that we have completeness, since the definition of truth makes reference to provability. The proof was for first-order theories, but it can easily be generalized to any complete formal logic. For example, if we take care not to confuse M with the S5 operator "possibly", we can easily get a complete non-monotonic extension of S5. However, none of these observations are very interesting unless we have some assurance that provability is decidable. We will shortly present a proof procedure for non-monotonic statement theories.

Fixed Points of Theories

Non-monotonic theories may have varying numbers of fixed points. Classically inconsistent theories have just one fixed point (the entire language *L*) and thus no models. The theory T2 in (15) also has no models due to the lack of a fixed point. Theories formulated in strictly classical language have exactly one fixed point, as does the theory

(17) $T3 = PC \cup \{ MC \supset C \}$.

Some theories have several fixed points, e.g. T1 in (14). It is also possible for a theory to have an infinite number of fixed points. This is exemplified (we assume equality and an infinite set of unequal constant letters) by

(18) $T4 = PC \cup \{ \forall x[Mp(x) \supset [p(x) \wedge \forall y[x \neq y \supset \neg p(y)]]] \}$.

Even in theories having only one fixed point, the non-monotonically provable statements need not coincide with the classically provable statements. Theory T3 above is an example, for $C \in TH(T3)$, but $C \notin Th(T3)$. Some statements will be provable in theories with multiple fixed points, but will have different proofs in each fixed point. For example, $MC \vee MD \in TH(T1)$, and $\exists x Mp(x) \in TH(T4)$.

The classical results concerning truth and provability for logical languages are that, for a given theory A, a formula is *valid* in A (true in all models of A) if and only if it is *provable* in A, and that the theory has a model if and only if it is *consistent* (cannot be used to derive a contradiction). In non-monotonic logic, somewhat different circumstances obtain. As Theorems 1 and 2 have shown, validity in a theory remains equivalent to provability. However, from the definition of models of non-monotonic theories, it follows that a non-monotonic theory A has a model only if the operator NM_A has a classically consistent fixed point. Non-monotonic theories can lack fixed points (e.g. the theory T1), but we have defined such theories to be inconsistent.

The basic structure theorem states that all fixed points of a non-monotonic theory A are (set inclusion) minimal fixed points.

Theorem 3. If $S_1, S_2 \in FP(A)$ and $S_1 \subseteq S_2$, then $S_1 = S_2$.

This result suggests that strict set-theoretic minimality is not a particularly interesting distinction among fixed points. In the following sections we will make steps towards more interesting classifications, but without a fully satisfactory solution. Important applications of this theorem are the following two corollaries.

Corollary 4. If *L* is a fixed point of A, then it is the only fixed point of A.

Note that if *L* is a fixed point of A, then A is classically inconsistent, that is, $Th(A) = L$.

Corollary 5. If $p, \neg p \in TH(A)$, then $TH(A) = L$.

With these results, we can study the notion dual to provability in non-monotonic theories. We say that a formula p is *arguable* from A if $p \in \cup FP(A)$, that is, if some fixed point of A contains p. Clearly, all provable formulas are arguable. Our next theorem shows that in consistent theories, provability and arguability are almost dual notions.

Theorem 6. If A is consistent and p is provable in A, then $\neg p$ is not arguable.

Unfortunately, the converse of this theorem is not true. For example, in the theory with no proper axioms, $\neg C$ is not arguable, but C is not provable. We will term the notion dual to provability *conceivability*. Thus all arguable formulas are conceivable, but not *vice versa*. We say *doubtless* p if and only if $\neg p$ is not arguable. In PC, C is doubtless yet not arguable, and in the theory

(19) $T5 = PC \cup \{ MC \supset C, M\neg C \supset \neg C \}$

C is arguable yet not doubtless.

It is worthy of note that the provable and arguable statements of a consistent theory cannot be classified as the monotonic theorems of the theory augmented by some set of assumptions. That is, the set of arguable statements may be inconsistent yet not sum to the entire language L, and the set of provable statements may involve assumptions that vary from fixed point to fixed point, as in the theory T2 above, where neither the assumption MC nor the assumption MD is present in both fixed points.

Another natural classification is that of "decision". We say that p is *decided* by a consistent theory A if and only if for all $S \in FP(A)$, either $p \in S$ or $\neg p \in S$. The dual to this notion is just its negation. In this case we say that A is *ambivalent* about p if p is not decided by A.

Corollary 7. If p is doubtless yet decided by A, p is provable.

The Evolution of Theories

We now turn to analyzing inter-theory relationships. These are important in describing the effects of incremental changes in the set of axioms, and this is the task of practical systems like the TMS [Doyle 1978], which has the task of maintaining a description of a model of a changing set of axioms. As we shall see, there are many unusual phenomena which occur when theories change. The most striking result shows that the analogue of the compactness theorem of classical model theory does not hold for non-monotonic theories. This has important repercussions on the methods useful in constructing "models" of theories incrementally.

Theorem 8. There exists a consistent theory with an inconsistent subtheory.

Proof: Consider the consistent theory

(20) $T6 = PC \cup \{ MC \supset \neg C, \neg C \}$.

The subtheory $PC \cup \{ MC \supset \neg C \}$ is inconsistent. ∎

Note, however, that the theory T6 in (20) has as a thesis the formula $\neg MC$, which makes it quite different than some previously considered theories.

In many cases, the changes in fixed points induced by changes in theories is less drastic than those apparent in the previous theorem. The simplest cases are as follows.

Theorem 9. If A is consistent, and p is arguable in A, then $A' = A \cup \{p\}$ is consistent, and $FP(A') \cap FP(A) \neq \emptyset$.

Unfortunately, this theorem cannot be strengthened to conclude that FP(A') is contained in FP(A), since in the theory

(21) $T7 = PC \cup \{ MC \supset \neg D, MD \supset \neg C, \neg C \supset E \}$

there are two fixed points, call them F1 and F2, with $\neg C \in F1$, $E \in F1$ and $\neg D \in F2$, $E \notin F2$. Extending this theory by adding the axiom E produces a theory also with two fixed points, one of which is F1, but the other fixed point F3 differs from F2 in that $E \in F3$ and $M\neg E \notin F3$.

Theorem 10. If A and $A' = A \cup \{p\}$ are consistent and $FP(A) \cap FP(A') \neq \emptyset$, then p is arguable in A.

Theorem 11. If A and $A' = A \cup \{p\}$ are consistent, then p is provable in A if and only if $FP(A') = FP(A)$.

The import of these theorems is that if a new axiom is already implicit in the current axioms, either no change of fixed point is necessary, or a simple shift to a different fixed point of the previous axioms is allowable. When considering changes which delete axioms from theories, the basic problem is the non-compactness result mentioned above. Other interesting questions are of the form "how few axioms must be added or removed to remove p". Answers to these questions will in general depend on the specific theory in question.

Another important phenomenon is the "hierarchy of assumptions" [Doyle 1978], in which some non-monotonic choices depend on others. This manifests in terms of fixed points as the addition of new axioms increasing the number of fixed points of the theory. For example, adding the axiom E to the theory

(22) $T8 = PC \cup \{ [E \wedge MC] \supset \neg D, [E \wedge MD] \supset \neg C \}$

increases the number of fixed points from one to two. In this case, E can be interpreted as the reason for choosing between $\neg C$ and $\neg D$.

To get a global view of theory evolution, we consider the set of all consistent theories containing a consistent theory A as a subtheory. For a formula p, we can consider the evolution of the properties of p of being arguable, provable, or decided over sequences of extensions of the theory A. The evolution of arguability is mainly a question of control structures; this is the point of the encoding of control primitives in non-monotonic dependency relationships given by Doyle [1978]. We have at present no way of describing the evolution of decision. However, analysis of the relationships between the theories and their extensions will shed light on how our semantics for Mp matches the intuitive notion of "p can be added consistently to the theory".

We say that p is *assumable* in a consistent theory A if the theory $A \cup \{p\}$ is also consistent. We name the dual notion by saying that p is *uncontroversial* in a theory if $\neg p$ is not assumable in the theory. The matching of the semantics of non-monotonic logic with this more standard notion of consistency will be apparent upon examining the correlation between assumability of p and the arguability of Mp in a theory, since this latter condition would seem to say there is a coherent interpretation of the axioms in which p is consistent. Our logic is weak, however, and so this correlation is weak. (The correlation is much stronger in the stronger logics mentioned later.) As an approximation, we note that Mp is arguable if p is arguable, and so instead attempt to correlate arguability of p with assumability of p. This correlation is as follows. By Theorem 9 the assumable formulas includes the arguable formulas, but not *vice versa* since C is assumable but not arguable in PC. The assumable formulas are incomparable with the conceivable formulas, since C is conceivable but not assumable in

(23) $T9 = PC \cup \{ C \supset [D \wedge [MD \supset \neg D]] \}$,

and $\neg C$ is assumable but not conceivable in the theory T3 of

(17). Also, the assumable formulas are incomparable with the uncontroversial formulas, since C is assumable but not uncontroversial in PC, and C is uncontroversial but not assumable in

(24) T10 = PC ∪ { C⊃[D∧[MD⊃¬D]], ¬C⊃[E∧[ME⊃¬E]] }.

We specify another classification by saying that a formula p is *safe* in a consistent theory A if and only if p ∈ TH(A') for all consistent A' such that A ⊆ A', and that p is *forseeable* if and only if ¬p is not safe. Let Safe(A) = {p: p is safe in A}. We then can characterize the set Safe(A) as follows.

Theorem 12. If A is consistent, then Safe(A) is the least set such that the following three conditions hold:

(i) A ⊆ Safe(A)

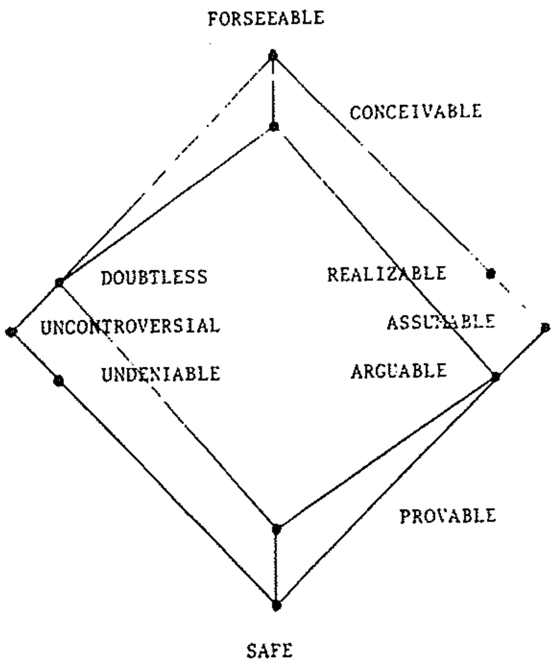
(ii) Th(Safe(A)) = Safe(A)

(iii) If p ∈ Safe(A), then Mp ∈ Safe(A).

It is clear that all safe formulas are both assumable and uncontroversial, and that these inclusions are proper. Elementary considerations show further that the forseeable formulas include the assumable and uncontroversial formulas, but again, not *vice versa*. Also, the provable formulas properly include the safe formulas with theory T3 in (17) as the example, and the forseeable formulas properly include the conceivable formulas via the same example.

A weakened version of assumability is produced by saying that p is *realizable* in a consistent theory A if there is some consistent theory A' such that A ⊆ A' and p ∈ A'. We also say that p is *undeniable* if and only if ¬p is not realizable. Clearly, the realizable formulas include the assumable formulas, but the converse does not hold as MC⊃¬C is not assumable in PC but is an axiom of the consistent theory T6 in (20). The forseeable formulas obviously include the realizable formulas, but not *vice versa* since C is forseeable but not realizable in the theory T9 of (23). Also, the realizable formulas are incomparable with the conceivable formulas, since C is conceivable but not realizable in T9 of (23), and ¬C is realizable but not conceivable in T3 of (17). The example of T10 in (24) provides an example of what following Kripke might be called the *paradoxical* formulas of a theory, formulas (in this case C) such that neither they nor their negations are realizable. The example of T9 in (23) provides an example of what might be called the *intrinsic* formulas of a theory, formulas (in this case ¬C) which are realizable and undeniable.

Putting all these observations together, we arrive at the following diagram of inclusions.



A Proof Procedure for Non-Monotonic Statement Theories

In this section, we demonstrate a proof procedure for the non-monotonic statement logic. This procedure is based on the semantic tableau method for the ordinary sentential calculus. [Beth 1958] In this method, a systematic attempt is made to find a falsifying interpretation for a formula under test. The formula is labeled "false" or "0", and semantic rules guide further labeling in an obvious way. For example, to show

[C ⊃ D] ⊃ [¬C ∨ D],

start by labeling the formula false:

```
[C ⊃ D] ⊃ [¬C ∨ D]
        0
```

For it to be false, its antecedent must be true and its consequent false:

```
[C ⊃ D] ⊃ [¬C ∨ D]
   1    0     0
```

and similarly for disjunction and negation. In order to proceed further, the tableau must split into two cases to handle the embedded implication:

```
I.  [C ⊃ D] ⊃ [¬C ∨ D]
     0 1    0  01 0
II. [C ⊃ D] ⊃ [¬C ∨ D]
     1 1 1  0  01 0 0
```

In case I., C is labeled both 1 and 0. In case II., D is labeled both 1 and 0. Thus there is no falsifying model, and the formula is valid.

Consider the tableau for [C ∨ D] ⊃ [C ∧ D]:

```
(25)    [C v D]  >  [C ^ D]
           1    0     0
          [C v D]  >  [C ^ D]
           1 1     0     0
                     [C v D]  >  [C ^ D]      CLOSED
                      1 1     0   0 0
                     [C v D]  >  [C ^ D]      OPEN
                      1 1 0   0   1 0 0
          [C v D]  >  [C ^ D]
             1 1   0     0
                     [C v D]  >  [C ^ D]      OPEN
                      0 1 1   0   0 0 1
                     [C v D]  >  [C ^ D]      CLOSED
                      1 1 1   0   1 0 0
```

This tableau has been split twice, for a total of four *branches*. Two branches are *closed* as before, that is, some formula is labeled both true (1) and false (0). But two are *open*, that is, there is an exhaustive consistent labeling of formulas. This means that there are two falsifying models, so the formula is not valid. (Notice that we could have been more clever in labeling the lines of this tableau. In the second line, for instance, we could have labeled both C's at once, forcing the D's to be labeled 0, and arriving at an open branch immediately.)

We will extend this procedure to handle non-monotonic statement theories. Without going into details, we assume an implementation of the algorithm just alluded to, which takes a *goal* and generates the complete tableau for it. (E.g., the goal of (25) is [C∨D]⊃[C∧D].) A tableau has several branches, each a consistent labeling of subformulas if one exists (when the branch is open), else a partial labeling (when it is closed). The tableau is the result of applying all rules to the goal. Two tableaux are equal if and only if they have the same goal. The tableau of a formula is obviously computable, since the number of branches is no greater than 2^N, where N is the number of subformulas of its goal.

For non-monotonic logic, we need to generalize to tableau structures. If A is a statement theory, and p is a formula whose provability is to be tested, then ⟨A, p, t, X⟩ is an A-tableau structure if and only if t is the tableau with goal A⊃p; and X is the smallest set such that t ∈ X, and if t' ∈ X, then if Mq appears labeled 0 in some branch b of t', then t'' ∈ X, where t'' is the tableau with goal A⊃¬q. In this last situation, we say that t' *mentions* t'' in branch b.

In the classical procedure, a tableau is closed if all its branches are, and this can be determined unambiguously. In the case of a tableau structure, we can't tell whether a tableau is closed until we have determined the status of the tableaux it mentions, and there may be loops to contend with.

Therefore we introduce the notion of an *admissible labeling* of a tableau structure, an assignment of one label, either OPEN or CLOSED, to each tableau in the structure, such that:

(a) If the tableau with goal A⊃¬q is labeled OPEN, then every occurrence of Mq is labeled 1 in every tableau, and
(b) A branch is labeled CLOSED if and only if some formula is labeled both 0 and 1 in that branch.

The proof procedure creates tableau structures and labels them, as follows. Given A and p, the first step is to construct the tableau with goal A⊃p. All other tableaux needed are then constructed. That is, if some constructed tableau has a formula Mq labeled 0 in an open branch, then construct the tableau with goal A⊃¬q if that tableau was not previously constructed. The tableau structure is then checked for admissible labelings by examining all possible labelings of the tableaux for labelings satisfying the admissibility test. This test consists of first labeling with 1 each occurrence of Mq in the tableau structure provided that the structure contains the tableau with goal A⊃¬q labeled OPEN. Then the labeling is admissible if all tableaux labeled OPEN have some open branch, and all tableaux labeled CLOSED have every branch closed. If in all admissible labelings the initial tableau with goal A⊃p is labeled CLOSED, then p is provable, and otherwise is unprovable. We will shortly prove the correctness of this algorithm.

We first present some examples. In the theory

(26) T11 = SC ∪ { MC⊃¬D, MD⊃¬E, ME⊃¬F }

(see [Sandewall 1972]) the T11-tableau structure for ¬F has only one admissible labeling:

```
T11 = MC>¬D |  t    ¬F | t'   ¬E | t''   ¬D | t'''   ¬C
        1   |       01 |      01 |       01 |        01
     MD>¬E  |       ME |      MD |       MC |
        1   |       0  |      0  |       0  |
     ME>¬F  |          |         |          |
        1   |  CLOSED  |   OPEN  |  CLOSED  |      OPEN
```

Notice that we don't bother to copy the axioms in each tableau, but only those parts that become relevant. The tableau structure shows that ¬F ∈ TH(T11), but ¬C ∉ TH(T11).

Another example is the T12-tableau structure for ¬C, where

(27) T12 = SC ∪ { MC⊃¬D, MD⊃¬C }.

```
T12 = MC>¬D  | t    ¬C  | t'    ¬D
        1    |      01  |       01
     MD>¬C   |      MD  |       MC
        1    |      0   |       0
```

This tableau structure has two admissible labelings. If t' is labeled OPEN, t is labeled CLOSED, and *vice versa*. So there is an admissible labeling in which t is labeled OPEN, and ¬C is not provable.

On the other hand, the T12-tableau structure for MC∨MD looks like this:

```
T12 = MC>¬D  | t   MCvMD  | t'    ¬C  | t''    ¬D
        1    |     0 00   |       01  |        01
     MD>¬C   |            |       MD  |        MC
        1    |            |       0   |        0
```

Again, there are two admissible labelings, but in both of them t is labeled CLOSED, so MC∨MD is a theorem of T12.

(The tableau structures just given are not really complete. It is left as an exercise for the reader to show that using the axioms to split each tableau into branches will not change the outcome.)

Theorem 13. The proof procedure always halts and finds all admissible labelings of the tableau structure for its goal.

The next two lemmas guarantee the correctness of the approach.

Lemma 14. If S is a fixed point of NM_A, there is an admissible labeling of the tableau structure for $A \supset p$ such that $p \in S$ if and only if the tableau is labeled CLOSED in that labeling.

Lemma 15. If there is an admissible labeling for the tableau structure for $A \supset p$, there is a fixed point S of NM_A such that, for every tableau with goal $A \supset q$, the tableau is labeled CLOSED if and only if $q \in S$.

Theorem 16. If A is a statement theory (a finite extension of the sentential calculus), then non-monotonic provability in A is decidable.

The proof procedure extends a previous procedure due to Hewitt [1972], and embodied in Micro-PLANNER [Sussman, Winograd and Charniak 1971], a computer programming language for (among other things) mechanical theorem proving. A practical implementation of this procedure would interleave the building and labeling of tableaux, and would avoid building a complete tableau structure when unnecessary. We invite you to compare this procedure with, for instance, the tableau-structure method for S5. [Hughes and Cresswell 1972] One difference between these procedures is that the present procedure splits tableaux into branches before generating alternatives, while the S5 procedure splits the whole set of alternatives into branches.

References

E. W. Beth, "On machines which prove theorems," *Simon Stevin (Wis-en Natuurkundig Tijdschrift) 32*, p. 49.

J. Doyle, "Truth Maintenance Systems for Problem Solving," MIT AI Lab, TR-419, January 1978.

P. J. Hayes, "Robotologic," in B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, New York: American Elsevier, 1970, pp. 533-554.

P. J. Hayes, "A Logic of Actions," in B. Meltzer and D. Michie, editors, *Machine Intelligence 6*, New York: American Elsevier, 1971, pp. 495-520.

P. J. Hayes, "The Frame Problem and Related Problems in Artificial Intelligence," in A. Elithorn and D. Jones, editors, *Artificial and Human Thinking*, San Francisco: Josey-Bass, 1973.

C. E. Hewitt, "Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot," MIT AI Laboratory TR-258, 1972.

G. E. Hughes and M. J. Cresswell, *An Introduction to Modal Logic*, London: Methuen and Co. Ltd. 1972.

A. K. Joshi, "Some extensions of a system for inferencing on partial information," in *Pattern Directed Inference Systems*, F. Hayes-Roth and D. Waterman, eds., New York: Academic Press, 1978.

A. K. Joshi and S. J. Rosenschein, "A formalism for relating lexical and pragmatic information: its relevance to recognition and generation," *Proc. Workshop on Theoretical Issues in Natural Language Processing*, Cambridge, Massachusetts, 1975, pp. 79-83.

I. Kramosil, "A Note on Deduction Rules with Negative Premises," *Proc. Fourth IJCAI*, pp. 53-56, 1975.

P. E. London, "A Dependency-Based Modelling Mechanism for Problem Solving," University of Maryland, Computer Science Department TR-589, Nov. 1977.

D. A. McAllester, "A Three-Valued Truth Maintenance System," MIT AI Lab, Memo 473, May 1978.

J. McCarthy and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in B. Meltzer and D. Michie, *Machine Intelligence 4*, New York: American Elsevier 1969, pp. 463-502.

J. McCarthy, "Epistemological Problems of Artificial Intelligence," *Proc. Fifth IJCAI*, pp. 1038-1044, 1977.

D. McDermott, "Assimilation of New Information by a Natural Language-Understanding System," MIT AI Lab, AI-TR-291, February 1974.

E. Mendelson, *Introduction to Mathematical Logic*, New York: Van Nostrand Reinhold, 1964.

M. Minsky, "A Framework for Representing Knowledge," MIT AI Lab, AI Memo 306, June 1974, reprinted without appendix in P. Winston, editor, *The Psychology of Computer Vision*, New York: McGraw-Hill, 1975.

R. C. Moore, "Reasoning From Incomplete Knowledge in a Procedural Deduction System," MIT AI Lab, AI-TR-347, December 1975.

A. J. Nevins, "A Human-Oriented Logic for Automatic Theorem Proving," *J. Association for Computing Machinery, 21*, pp. 606-621.

V. R. Pratt, "The Competence/Performance Dichotomy in Programming," MIT AI Lab Memo 400, Jan. 1977.

W. V. Quine and J. S. Ullian, *The Web of Belief*, second edition, New York: Random House, 1978.

R. Reiter, "On Closed World Data Bases," Department of Computer Science, University of British Columbia, TR 77-14, October 1977.

R. Reiter, "On Reasoning by Default," *Proc. Second Symp. on Theoretical Issues in Natural Language Processing*, Urbana, Illinois, August 1978.

N. Rescher, *Hypothetical Reasoning*, Amsterdam: North Holland 1964.

J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *J. Association for Computing Machinery, 12*, pp. 23-41.

E. Sandewall, "An Approach to the Frame Problem, and its Implementation," in B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, New York: John Wiley and Sons, 1972, pp. 195-204.

R. M. Stallman and G. J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," *Artificial Intelligence*, Vol. 9, No. 2, pp. 135-196.

G. J. Sussman, T. Winograd and E. Charniak, "MICRO-PLANNER Reference Manual," MIT AI Lab, AI Memo 203a, December 1971.

Completeness of Conditional Reductions

Daniel Brand
John A. Darringer
William H. Joyner, Jr.

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract

Many systems for symbolic computation, program verification, and automatic theorem proving use reductions (rewrite rules) to simplify mathematical expressions and prove equalities between them. There has also been work on proving properties of sets of reductions for theories axiomatized by equalities. However, these theoretical results deal with unconditional reductions, while most systems allow conditional reductions. These are rules that require conditions to hold before they are applied. In this paper we examine conditional reductions and the theoretical and practical problems of obtaining complete sets of such reductions.

1. Introduction

Reduction systems have played an important role in the design of many symbolic computation systems. Systems such as REDUCE [Hearn 1971] and SCRATCHPAD [Griesmer 1971] allow a user to provide his own set of reduction rules for simplifying algebraic expressions. Other examples of practical interest are provided by theorem provers based on reductions [Bledsoe 1971, Carter et.al. 1977]. The aim in these applications is to transform one symbolic expression into another simpler or more readable one by the repeated use of replacement rules until no rule applies.

Reduction systems have also been modeled and studied theoretically to determine their properties in general [Rosen 1973, Slagle 1974]. An important property that has been investigated is the notion of *completeness* [Knuth and Bendix 1970, Lankford 1975]. This work has produced methods for deciding, in certain circumstances, whether a set of reductions together with its interpreter captures the theory given by the reductions treated as equations. In addition to its theoretical interest, this question is important to developers of systems which make use of reductions, for only then can claims be made about the coverage of an application based on a system of reductions.

However, there is a serious difference between the reductions studied theoretically and those used in practice. Systems of *unconditional* reductions have received the most study, while most systems allow reductions to have *conditions* that are required to hold before the reduction can be applied. This paper attempts to develop a theory of conditional reductions.

After a brief review of unconditional reductions, we point out, through examples, the need for conditional reductions and some of the problems in their application. A notion of completeness is defined for conditional reductions that is independent of the way in which the reductions are interpreted. We provide three possible interpretations for conditional reductions, and discuss the problems with each on typical examples, and their effect on the question of deciding completeness.

2. Unconditional Reductions

In recent papers on reduction systems (for example, [Lankford 1975, Musser 1978]), the concern is with theories which are axiomatized by equalities:

$$L_1 = R_1$$
$$\dots$$
$$L_n = R_n.$$

A set of reductions for this theory is produced by replacing the equalities with a preferred direction for substitution:

$$L_1 \rightarrow R_1$$
$$\dots$$
$$L_n \rightarrow R_n.$$

The relation $\rightarrow$ is defined on expressions of the theory. If α and β are expressions, then $\alpha \rightarrow \beta$ iff there is some subexpression of α which is an instance of some L_i under a substitution θ, and β results from α by replacing this subexpression with the corresponding instance of R_i. That is, if $\alpha = \alpha[L_i\theta]$, then $\beta = \alpha[R_i\theta]$.

Since normally an expression will be reduced repeatedly, as long as a rule applies, a desirable property of a set of reductions is the *finite termination property*: that there exist no infinite sequences $\alpha_0 \rightarrow \alpha_1 \rightarrow \ldots$. As usual, $\rightarrow^*$ will be here the reflexive and transitive closure of $\rightarrow$, and an expression α will be *irreducible* if there is no expression β such that $\alpha \rightarrow \beta$. If $\alpha \rightarrow^* \beta$ and β is irreducible, then α *simplifies* to β. If for a set or reductions having the finite termination property, there is no expression α such that $\alpha \rightarrow^* \beta$ and $\alpha \rightarrow^* \gamma$ for distinct irreducible β and γ, then that set has the *unique termination property*, and is called *complete*. Completeness is a useful concept because a complete set of reductions for a theory provides a decision procedure for it. That is, to decide an equality in the theory giving rise to the reductions, it is sufficient to reduce both sides of the equality to irreducible expressions and check them for identity.

Knuth and Bendix[1970] and Lankford[1975] provide formal definitions of these properties and give algorithms that will decide whether a set of reductions with the finite termination property has the unique termination property. The algorithms also attempt to "complete" the set of reductions by adding additional rules and testing again. Termination of this phase is not guaranteed, however. They have applied the results to group theory, semigroups, and rings. Extensions to the reduction idea have been made to theories with associative and commutative axioms, which do no have finite termination [Lankford and Ballantyne 1977, Stickel and Peterson 1977], and complete reduction systems have been applied to data type verification [Musser 1978].

3. Conditional Reductions

As mentioned earlier, unconditional reductions alone are rarely used in practice. For example, the SCRATCHPAD and REDUCE systems use reductions of the form $L \rightarrow R$ **if** A , where A is a boolean expression, presumably containing variables from L. The intention is that the reduction is applied only if condition A holds. The principal questions arising about sets of such reductions depend on the nature of the conditions A, the definition of completeness, and the way in which the reductions are applied to expressions -- the definition of the relation $\rightarrow$.

We could consider two possibilities for conditions associated with reductions. A condition may be an expression to which reductions are applied in order to determine whether it evaluates to true or false, or it may be a formula of a theory separate from the one being defined by the reductions. For both practical and theoretical reasons it is advantageous to take the latter approach. We call the theory serving as a source of conditions the *underlying theory*, and assume that we have a decision procedure for it. We show in this paper that even with such simplifying assumptions deciding completeness of conditional reductions is difficult. In practice this assumption is satisfied by developing a hierarchy of theories, in which each theory is defined by a set of reductions with conditions from the theories lower in the hierarchy. Perhaps a conditional reduction can be best understood as a large (possibly infinite) set of unconditional reductions obtained by replacing all the variables of the underlying theory by all possible combinations of values satisfying the associated condition.

We assume that the matching of the left hand side of a rule to a subexpression takes place as in unconditional reductions (see [Lankford 1975]). For example, an expression does not match a left hand side F(x,x) unless both its arguments are identical; no other test is allowed to determine the equality of the arguments. Thus we restrict all non-syntactic tests to the conditions and do not interpret argument patterns as *de facto* tests in the underlying theory. Systems with associative and commutative reductions do allow a generalization of the identity test, but in a well-defined syntactic way [Stickel and Peterson 1977].

Even with an oracle for the conditions, defining the "correct" way of interpreting the rules is not straightforward. Consider the algorithms used in REDUCE and SCRATCHPAD, for example. If REDUCE is attempting to apply a reduction to an expression and the condition does not hold, the situation is as if no match for the left hand side of the rule had been found, and the search for an applicable rule continues. This same scheme is used in the theorem prover of Carter [1977]. However, in SCRATCHPAD the above scheme is followed only if the condition is shown false. When symbols are involved an oracle may not be able to resolve a condition to either true or false and in this case SCRATCHPAD will make no reduction to that expression.

This problem of unresolvable conditions can be illustrated by the following example of expressions involving one dimensional arrays.

Example 1: Let us represent the access of the i-th element of an array a by ACC(a,i), and the assignment of x to the i-th element by a := CH(a,i,x). Thus the result of several array assignments can be represented by a nested expression built from ACC, CH, names of arrays, names of indices, and names of values storable in the arrays. Assume that we have the task of writing a set of reductions that would reduce any such expression to a canonical form, that is, two expressions describing the same array value will reduce to the same form. We might include the following two reductions:

$$\mathrm{ACC}(\mathrm{CH}(a,i,x),j) \rightarrow x \quad \textbf{if } i=j$$
$$\mathrm{ACC}(\mathrm{CH}(a,i,x),j) \rightarrow \mathrm{ACC}(a,j) \ \textbf{if } i\neq j$$

The underlying theory is arithmetic, i and j are variables ranging over its domain, = and ≠ are operators of the underlying theory. In contrast, ACC and CH are operators of the theory being defined by reductions, and a is a variable of this theory. In particular, ACC, CH, and a have no interpretation. The range of x is not important to our discussion; it could be the same as i, the same as a, or range over a completely different domain.

Suppose these reductions are to be applied to the expression ACC(CH(b,m,ACC(b,n)),n). Assuming that nothing is known about b, m, and n, neither condition m=n nor m≠n can be proved. But in either case the two reductions give the same result, namely ACC(b,n). The difficulty is that the interpretations of either the SCRATCHPAD or REDUCE will not produce this result and can be considered defective or incomplete. Note that this is a different form of incompleteness than the one familiar in unconditional reductions. There, incompleteness results when an expression can be simplified in more than one way. In the conditional case we have the additional problem that an expression, under some reasonable definitions of rule application, may not reduce at all, although the rules imply that a reduced form exists.

An additional difference in the notion of completeness is due to the presence of an underlying theory for the conditions. To illustrate this point consider the completeness of the reductions in the following example.

Example 2: NOT(T) → F
NOT(F) → T.

Should this set of two reductions be considered complete in view of the fact that the expression NOT(NOT(x)) does not reduce to x? As a set of unconditional reductions this set is complete because there is no reason to suppose and no way to assert that x can take on only the two values T and F. However if we have an underlying theory with domain {T, F}, then we can formalize the incompleteness of this set.

To formalize conditional reductions, we need to define formulas of the underlying theory and expressions in the theory to which reductions are applied. Although the underlying theory of the conditions is separated from the theory being defined, the latter theory will contain terms of the underlying theory. These terms are tested by the conditions when the reductions are applied. The variables of the underlying theory will be called *term variables*. The usual *expression variables* of the theory being defined are, as in unconditional reductions, uninterpreted place holders.

Definition. Assume a theory (the *underlying theory*) with syntax determined by a set of function symbols, predicate symbols, variables, connectives, and quantifiers. A *term* is constructed in the usual way from variables and function symbols. A *formula* is constructed in the usual way from terms, predicate symbols, connectives and quantifiers.

We assume a given interpretation for the underlying theory. This includes a domain over which variables range, a function for each function symbol and a predicate for each predicate symbol. Moreover, we assume that that the language of the underlying theory contains a constant symbol for each element of the domain.

We also introduce *operators*. They are distinct from the function symbols of the underlying theory because they have no interpretation.

Definition. An *expression* is constructed from operators, expression variables and terms, similarly as terms are constructed.

Definition. A *reduction* has the form L → R if A, where L and R are expressions and A is a formula, referred to as the *condition*. The only free variables of A may be the term variables appearing in L, and the only variables of R may be those of L.

Nothing has been said about how such a reduction is applied. After defining completeness, we shall present several alternate definitions of reduction application.

4. Completeness

In this section we define a notion of completeness for conditional reductions that is independent of a particular interpretation of reduction application. Conditional reductjons are normally applied in the presence of a constraint. In practice, a constraint is a formula restricting the values of expression components. For example, symbolic computation along program paths may use constraints to record branch conditions. Below we formally define a constraint as a set of environments, and define completeness relative to such a constraint. The predicate $x<0$ would denote the constraint consisting of all environments with negative x.

Definition. An *environment* is an assignment of values to all term variables. E(x) is the evaluation of the term or formula x in environment E using the given interpretation of the underlying theory. For a formula A and an environment E we say that A *is true in* E, written E ⊨ A, if E(A) = TRUE. A *constraint* is a set of environments. A formula A *is true in* a constraint C, written C ⊨ A, iff A is true in every environment contained in C.

We extend the notion of evaluation to expressions:

Definition. For an expression α and an environment E, E(α) is an expression obtained from α by replacing terms with constants. The corresponding constant for a term t is obtained by evaluating t after its variables are replaced by the values assigned to them by E.

Our definition of completeness assumes that we have a set of reductions and objects to which reductions are applied (expressions or conditional expressions depending on interpretation). Each of the three interpretations below defines how the set of reductions determines a relation $\rightarrow$ on the objects relative to a constraint. We write $C \supset \alpha \rightarrow \beta$ if α reduces to β under constraint C, and abbreviate this to $\alpha \rightarrow \beta$ if C is the set of all environments (i.e., there is no constraint). We will assume in the following that $\rightarrow$ in each interpretation makes the set of objects into a well founded set for any constraint -- that is, we assume the *finite termination property*. As usual, we say that an object α is *irreducible* iff there is no object β so that $C \supset \alpha \rightarrow \beta$, and that the relation $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$.

Besides the relation $\rightarrow$ each interpretation will also define two binary relations $=_e$ and $=_s$ on the set of objects, both relative to a constraint. The relation $=_e$ is syntactic equality between objects and ideally should be strict identity. We cannot make it strict identity in cases where the objects contain terms of the underlying theory. (Reductions modulo an equivalence $=_e$ are treated in general by Huet[1977].) The relation $=_s$ represents semantic equality -- two objects are semantically equal iff they can be proved equal from the set of reductions when considered as equations. Of course, objects related under $=_e$ are also related under $=_s$. The goal in applying reductions is to obtain objects which are equivalent under $=_e$ from any objects related by $=_s$.

Definition. A given set of reductions is *complete* with respect to a constraint C iff for any two objects α_0 and β_0 satisfying $C \supset \alpha_0 =_s \beta_0$, there exist objects α and β such that
a) $C \supset \alpha =_e \beta$,
b) $C \supset \alpha_0 \rightarrow^* \alpha$, and
c) $C \supset \beta_0 \rightarrow^* \beta$.

In the interpretations below, we will define $\rightarrow$, $=_s$, and $=_e$ so that *all* irreducible α and β obtained from α_0 and β_0 are equivalent under $=_e$.

5. Three Interpretations

We now present three different methods of applying reductions and examine how they effect the process of determining completeness. The first interpretation is probably the most common in practice; REDUCE [Hearn 1971], SCRATCHPAD [Griesmer 1971], and MCS [Carter 1977] all use forms of it. It requires the condition of a reduction to be true before the reduction is applied. The second interpretation requires only that the condition not be false; when a reduction is applied its condition is carried along with the result in a conditional expression. This approach, which changes conditional reductions into an **if-then-else** form, has been used by Musser [1978]. The third interpretation combines features of the other two; it uses conditional expressions only in an intermediate stage, and attempts to combine them to yield unconditional expressions.

Interpretation 1: In this interpretation the objects being reduced are expressions. A reduction is applied only if the associated condition is true. This can be illustrated by the effect of this interpretation on example 1. The expression ACC(CH(a,i,x),i+1) can be reduced to ACC(a,i+1) using the second reduction and the fact that $i \neq i+1$. However, ACC(CH(a,i,ACC(a,j)),j) cannot be reduced using either reduction because for some values of i and j we have i=j, and for others $i \neq j$. This also illustrates that the set of the two reductions is incomplete since the above expression is equal to ACC(a,j) in both cases.

We can define this interpretation formally.

Definition. For two expressions α and β and for a constraint C we define $C \supset \alpha =_e \beta$ iff $E(\alpha) = E(\beta)$ for all environments E of C.

Definition. For a constraint C and expressions γ and δ we say that γ *reduces to* δ *under* C, written $C \supset \gamma \rightarrow \delta$, iff there exists a reduction $L \rightarrow R$ if A and a substitution θ so that the following three conditions are satisfied.
a) $\gamma = \gamma[L\theta]$,
b) $C \models A\theta$, and
c) $\delta = \gamma[R\theta]$.
In this situation we say that the reduction $L \rightarrow R$ if A is applied to γ to produce δ.

Definition. For an environment E and two expressions α and β, $E \supset \alpha =_s \beta$ is the reflexive, symmetric and transitive closure of $\{E\} \supset \alpha \rightarrow \beta$ and $\{E\} \supset \alpha =_e \beta$. For a constraint C and expressions α and β, $C \supset \alpha =_s \beta$ iff $E \supset \alpha =_s \beta$ for every $E \in C$.

For interpretation 1 completeness is undecidable, as proved in [Brand 1978]. The proof uses the standard approach of reducing the halting problem. Given a Turing machine, we construct a set of reductions, which is complete iff the turing machine does not terminate on blank tape. The incompleteness of the set of reductions is based on

$S(v) \rightarrow e$ if $v=0$

$S(v) \rightarrow H$ if $v \neq 0$

For nonzero v, S reduces to the irreducible H and the reductions are designed so that any expression containing H reduces to H. If v is zero then S reduces to e, representing the initial configuration. The reductions are such that an expression containing e will reduce to H iff the expression represents a halting computation of the turing machine starting from a blank tape. Thus we have a set of reductions with the finite termination property that is complete iff there is no expression representing a halting computation of the given Turing machine.

Interpretation 2: The reductions in example 1 are incomplete under Interpretation 1 because it was not possible to apply reductions under the assumption that i=j and i≠j, and then compare the results. This splitting into cases can be done in this interpretation, where the objects are conditional expressions and a reduction is applied as soon as its left hand side matches and its condition is not identically false. When applied to example 1 we get:

```
    ACC(CH(a,i,ACC(a,j)),j)
 → if i=j then ACC(a,j)
      else ACC(CH(a,i,ACC(a,j)),j)
 → if i=j then ACC(a,j) else ACC(a,j)
 → ACC(a,j)
```

In the formal treatment below we omit these **if-then-else** trees and look only at the leaves and the conditions on the paths to the leaves. Our conditional expressions will be of sets of pairs, where each pair specifies an expression and its associated condition. For example, the expression

if i=j **then** ACC(a,j) **else** ACC(a,j)

would be written

{(ACC(a,j), i=j), (ACC(a,j), i≠j)}.

We will not further reduce this expression (as was done above using **if-then-else**), but instead define $=_e$ so that

{(ACC(a,j), i=j), (ACC(a,j), i≠j)}

$=_e$ {(ACC(a,j), TRUE)}.

Definition. A *conditional expression* is a set of pairs $\{...(\alpha_i, A_i)...\}$, where α_i are expressions and A_i are formulas of the underlying theory. For any $i \neq j$ we require that $A_i \wedge A_j$ be false under the given constraint.

Definition. For two conditional expressions ξ and ω and a constraint C we say that ξ *reduces to* ω under C, written $C \supset \xi \rightarrow \omega$, iff

a) there is a reduction $L \rightarrow R$ **if** A,

b) ξ has the form $\xi_0 \cup \{(\gamma[L\theta], B)\}$ (for some substitution θ),

c) $B \wedge A\theta$ is not false under C, and

d) ω is $\xi_0 \cup \{(\gamma[R\theta], B \wedge A\theta), (\gamma[L\theta], B \wedge \neg A\theta)\}$.

Definition. For two conditional expressions $\xi = \{...(\alpha_i, A_i)...\}$ and $\omega = \{...(\beta_j, B_j)...\}$ and a constraint C we say that $C \supset \xi =_e \omega$ iff for all i, j, and all environments E in C satisfying $A_i \wedge B_j$, $E(\alpha_i) = E(\beta_j)$.

Definition. The relation $=_s$ between conditional expressions is the least relation containing $\rightarrow$, $=_e$ and closed under reflexivity, symmetry and transitivity.

Test for completeness. A given set of reductions is tested for completeness with respect to a constraint C as follows:

a) Select all pairs of reductions $L_1 \rightarrow R_1$ **if** A_1, $L_2 \rightarrow R_2$ **if** A_2, such that:

i) L_2 is of form $L_2[L_1']$,

ii) L_1' is not a variable, and

iii) L_1 and L_1' are unifiable with most general unifier θ.

b) For each such pair, find some irreducible expressions π and ρ such that

$C \supset \{(L_2[R_1]\theta, A_1\theta)\} \rightarrow^* \pi$ and

$C \supset \{(R_2\theta, A_2\theta)\} \rightarrow^* \rho$.

For completeness we require that $C \supset \pi =_e \rho$.

The test and proof of its correctness are essentially the same as in Knuth and Bendix. In our case it is only complicated by the underlying theory, and the proof can be found in [Brand 1978].

Interpretation 3: In this interpretation the objects to which reductions are applied are expressions. To define reductions in this scheme, we will consider pairs (γ, F), where γ is an expression and F is a formula, and define reductions on pairs and then reductions on expressions.

This pair calculus approach has some advantages over the previous interpretations. A major drawback of interpretation 1 (not present in interpretation 2) is that if neither alternate of reductions of the form $L \rightarrow R_1$ **if** A and $L \rightarrow R_2$ **if** $\neg A$ can be followed, no transformation is made, even in cases where R_1 and R_2 have identical instances, or instances which can be made the same by further reductions. However, in most cases following both paths in the hope of obtaining equal expressions is futile. In the **if-then-else** formulation, this often produces a large conditional expression, which might not be regarded as simpler than the original expression. The pair calculus approach retains only those results of branching which lead to simpler expressions without added conditions, and thus may be preferred if the **if-then-else** construction is unwanted or proves cumbersome.

Definition. The pair (γ, F) *reduces to* (δ, G), written $(\gamma, F) \to (\delta, G)$, iff

a) there exists a reduction $L \to R$ **if** A and a substitution θ such that δ results from γ as it would if the unconditional reduction were applied, and G is the formula $F \;\&\; A\theta$; or

b) $(\gamma, F) \to^* (\delta, G_1)$, $(\gamma, F) \to^* (\delta, G_2)$, and G is the formula $G_1 \vee G_2$; or

c) $(\gamma, F) \to^* (\delta, G)$.

We now use the definition of reduction between pairs to define a notion of reduction between expressions based on this pair calculus.

Definition. γ reduces to δ under a constraint C, written $C \supset \gamma \to \delta$, iff $(\gamma, \text{TRUE}) \to (\delta, P)$ and $C \models P$.

$=_e$ and $=_s$ are defined on expressions as in interpretation 1.

It should be clear from the definition that the calculus of pairs is at least as powerful as interpretation 1; that is, if $C \supset \gamma \to^* \delta$ under interpretation 1 then $C \supset \gamma \to^* \delta$ under interpretation 2. To see the increased power of this method, note that it succeeds on example 1: the pair (ACC(CH(a,i,ACC(a,j)),j), TRUE) reduces to (ACC(a,i), $i=j \vee i\neq j$).

However, the pairs approach does not offer a solution to some problems that interpretation 2 does solve. Consider the example below:

Example 3: $F(x) \to G(x)$ **if** A
$F(x) \to H(x)$ **if** $\neg A$
$K(x) \to G(x)$ **if** A
$K(x) \to H(x)$ **if** $\neg A$

Under interpretation 2, F(a) and K(a) would reduce to identical conditional expressions. Under interpretation 3, they would not reduce at all, since no merging could occur.

Completeness under this definition is undecidable. The proof is identical to the proof for interpretation 1, except that it is based on example 3.

6. Conditional Reductions and Equality

Because of the substitutivity property of equality, an expression in the presence of a constraint implying equality between terms might represent an equivalence class of expressions, and deciding on a representative member of the class is often difficult. An example will illustrate some of the problems:

Example 4: $F(i,j,k) \to G(i,i)$ **if** $i=j$
$F(i,j,k) \to G(j,k)$ **if** $i\neq j$

If we consider the expression F(a,b,a), interpretation 1 makes no transformation, since neither $a=b$ nor $a\neq b$ can be proved. The **if-then-else** approach, interpretation2, produces

if a=b **then** G(a,a) **else** G(b,a),

and might replace G(a,a) by G(b,b), depending on any ordering assumed and on what rules for conditionals are present. The pairs approach, interpretation 3, could produce the pairs (G(a,a), $a=b$) and (G(b,a), $a\neq b$). Neither of these two methods could continue because G(a,a) and G(b,a) are not the same.

The problem is that given these reductions, F(a,b,a) should reduce to G(b,a), with no attached conditions. Since G(a,a) is obtained on the $a=b$ branch, it can just as well be written G(b,a), and since this is the same as the expression obtained on the $a\neq b$ branch, the conditional or the two pairs should yield G(b,a) as the result.

One solution to this problem is to carry around equivalence classes of expressions when equalities induce them, and allow reductions such as the above to occur whenever two classes have a nonempty intersection rather than insist upon identity. This would give the classes {G(a,a), G(a,b), G(b,a), G(b,b)} and {G(b,a)} in the above example.

The problem does not lie with equalities in the conditions. For example the expression

if $a > 5$ **then** F(a) **else** F(|a|)

should simplify to F(|a|). But in order to do that it is necessary to realize that it is valid to replace F(a) in the **then** part by the more complicated expression F(|a|).

7. Summary

We have defined conditional reductions and examined three different ways of applying them, representing their use in practice. We also have defined a notion of completeness that is independent of the particular interpretation, and considered the question of deciding completeness in each of these interpretations. The three approaches differ in their handling of unresolved conditions. The first approach does not keep track of all potential reductions and therefore is the least demanding of space and time. For this reason it is the most common interpretation in practice. However, it is also the weakest: more sets of reductions are incomplete under this interpretation than either of the others. It has the additional disadvantage that it is undecidable whether an arbitrary set of reductions is complete or not, even if we assume an oracle for the underlying theory and that the given reductions are guaranteed to terminate. The result is that it is difficult for users of systems that employ this approach to be sure that a set of reductions captures the intended theory.

The second interpretation overcomes this problem by maintaining a record of all possible outcomes when conditions are unresolved. Thus there are sets of reductions that would be incomplete using the first interpretation but that are complete using the second. Moreover, it is decidable whether a given set of reductions is complete. The decision procedure is an extension of the test described by Knuth and Bendix [1970]. The disadvantage of this approach is the exponential space and time required to maintain this set of possible alternatives.

The third interpretation is an attempt at a compromise. That is, because only a single expression is ever retained, it requires fewer resources than the second interpretation. Further, more reductions sets are complete with this interpretation than with the first.

While it is clear that conditional reductions are necessary and useful, they are more difficult to deal with formally and practically, compared with unconditional reductions. Our study of three different interpretations suggests that in order to make completeness decidable we must use a powerful notion of reduction application, which in turn requires greater computational resources.

References

Bledsoe, W.W. (1971), Splitting and Reduction Heuristics in Automatic Theorem Proving, *Artificial Intelligence* 2, 1 (Spring 1971), 55-77.

Brand, D., Darringer, J. A., and Joyner, W. H. (1978), Conditional Reductions, IBM Watson Research Center, RC7404 (December 1978).

Carter, W.C., Ellozy, H.A., Joyner, W.H., Jr., Leeman, G.B., Jr. (1977), Techniques for Microprogram Validation, IBM Watson Research Center, RC6361.

Griesmer, J. H., and Jenks, R. D. (1971) SCRATCHPAD/1 - An Interactive Facility for Symbolic Mathematics, in Proc. 2nd Symp. on Symbolic and Algebraic Manipulation, Petrick, S. (Ed.), March, 1971, ACM, New York

Hearn, A.C. (1971), REDUCE 2 - A System and Language for Algebraic Manipulation, Proc. 2nd Symp. on Symbolic and Algebraic Manipulation.

Huet, G. (1977), Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems, *Proc. 18th Symp. Foundations of Computer Science,* (November 1977), 30-45.

Knuth, D.E. and Bendix, P.B.(1970), Simple Word Problems in Universal Algebras, *Computational Problem in Abstract Algebra,* ed. J. Leech, Pergamon Press, (1970), 263-297.

Lankford, D.S.(1975), Complete Sets of Reductions for Computational Logic. Memo ATP 21, Mathematics Department, University of Texas, Austin, (Jan. 1975).

Lankford, D. S. and Ballantyne, A. M. (1977), Decision Procedures for Simple Equational Theories with a Commutative Axiom, circulated by the authors.

Musser, D. R. (1978), A Data Type Verification System Based on Rewrite Rules, USC-ISI, March 1978.

Rosen B.K. (1973), Tree Manipulating Systems and Church-Rosser Property, *J. ACM* 20, 1 (January 1973), 160-187.

Slagle, J.R. (1974), Automated Theorem Proving for Theories with Simplifiers, Commutativity, and Associativity. *J. ACM* 21, 4 (October 1974), 662-642.

Stickel, M. E. and Peterson, G. E. (1977), Complete Sets of Reductions for Equational Theories with Complete Unification Algorithm circulated by the authors.

RESOLUTION BY UNIFICATION AND EQUALITY *

Vincent J. Digricoli
Courant Institute of Mathematical Sciences, New York University, New York, N.Y. 10012
Hofstra University, Hempstead, New York 11550

Abstract

This paper presents two rules of inference: RUE (Resolution by Unification and Equality) and NRF (the Negative Reflexive Function Rule) which together are shown to be sound and complete to prove the E-unsatisfiability of a clause set S containing neither the equality nor the functionally reflexive axioms. It furthermore specifies a hyperresolution restriction strategy using RUE and NRF which is complete. A viability criterion and an equality restriction are defined which can be used to reduce the production of useless inferences.

1. Introduction

The pre-eminent use of the equality predicate and the importance of the equality axioms in logic make it imperative that deductive systems treat of equality effectively.

Important research in respect to the equality relationship has been carried out by many distinguished authors including Darlington [2], Robinson and Wos [3], Sibert [4], Morris [5], Anderson [6], Slagle [7], Knuth and Bendix [8] and Brand [9]. More recently Harrison and Rubin [10] have introduced the concept of generalized resolution as a method to more effectively handle equality in resolution.

It is the line of research proposed by Morris in E-resolution and by Harrison and Rubin in generalized resolution that we pursue in this paper. We introduce two rules of inference:

RUE: Resolution by Unification and Equality

NRF: the Negative Reflexive Function Rule

which together are shown to be sound and complete to prove the E-unsatisfiability of a clause set S containing neither the equality nor the functionally reflexive axioms. We furthermore specify a hyperresolution restriction strategy using RUE and NRF which is complete. Morris used paramodulation in conjunction with the inference rules of E-resolution. In this paper we rely on a disagreement set analysis in place of paramodulation.

A clause set S using the equality predicate but not containing the equality axioms is E-unsatisfiable if there is no interpretation which satisfies the conjunction of S with the equality axioms which are stated below:

(1) Reflexivity: $x = x$

(2) Symmetry: $x = y \rightarrow y = x$

(3) Transitivity: $x = y \wedge y = z \rightarrow x=z$

(4) Substitution in Functions:

$$x_0 = x_k \rightarrow f(\ldots x_{k-1}, x_k, x_{k+1}, \ldots) = f(\ldots x_{k-1}, x_0, x_{k+1}, \ldots)$$

(5) Substitution in Predicates:

$$x_0 = x_k \wedge P(\ldots x_{k-1}, x_k, x_{k+1}, \ldots) \rightarrow P(\ldots x_{k-1}, x_0, x_{k+1}, \ldots)$$

2. Disagreement Sets

We define a *disagreement set of a pair of terms* as:

If t_1 and t_2 are identical only one disagreement set exists, the empty set.

If t_1 and t_2 represent different functions (a constant is a function of no arguments), there is only one disagreement set, namely, $D = \{[t_1, t_2]\}$.

If t_1 and t_2 differ with $t_1 = f(a_1, \ldots, a_n)$, $t_2 = f(b_1, \ldots, b_n)$, then D is a disagreement set of t_1 and t_2 if it is the set of pairs of corresponding arguments (a_i, b_i) which do not match (the topmost disagreement set) or the union of disagreement sets, one from each non-matching pair (a_i, b_i).

In the example:

$$t_1 = f(\ a,\ g(a),\ h(g(b))\ )$$
$$t_2 = f(\ a,\ g(c),\ h(g(a))\ )$$

there are a total of six different disagreement sets:

the topmost disagreement set:

$D_1 = \{[g(a), g(c)],\ [h(g(b)),\ h(g(a))]\}$

$D_2 = \{[a, c],\ [g(b),\ g(a)]\}$

the bottom-most disagreement set:

$D_3 = \{[a, c],\ [b, a]\}$

and also

$D_4 = \{[g(a), g(c)],\ [g(b), g(a)]\}$

$D_5 = \{[g(a), g(c)],\ [b, a]\}$

* This work was supported by Grant No. N00014-75-C-0571 with the Office of Naval Research; and by the Department of Energy, Contract No. EY-76-C-02-3077.

$D_6 = \{[a,c], [h(g(b)), h(g(a))]\}$

We can prove that $t_1 = t_2$ if we can prove the equality of all the pairs in any one disagreement set. This follows from the equality substitution axiom for for functions. We are in fact enumerating the ways in which $t_1 = t_2$ can possibly be proven by using this substitution axiom.

We now define the *disagreement set of a pair of complementary literals*: $P(s_1,\ldots,s_n)$, $\bar{P}(t_1,\ldots,t_n)$ as the union $D = \bigcup_{i=1}^{n} D_i$ where D_i is a disagreement set of (s_i,t_i).

Using the substitution axiom for predicates we can state that

$$P(s_1,\ldots,s_n) \wedge \bar{P}(t_1,\ldots,t_n) \to D$$

where D now represents the disjunction of the inequalities specified by any disagreement set of P and $\bar{P}$. In resolution by unification and equality we can resolve P and $\bar{P}$ immediately to D.

In our discussion D will represent either a disagreement set $\{(a_i,b_i)\ i=1,k\}$ or a disjunction of the inequalities specified by the disagreement set, namely $\Sigma_1^n\ a_i \neq b_i$. It will be clear from the context which D denotes.

3. The MGPU Substitution

We may wish to minimize and if possible eliminate the disagreement between a pair of terms by using substitution. In the case where t_1,t_2 are unifiable, we may apply the standard algorithm to compute the most general unifier (MGU) of t_1,t_2. Below we have modified this algorithm so that when t_1,t_2 are not completely unifiable, the algorithm instead of terminating at the first irreducible disagreement continues to perform further unification. We call the substitution produced by this algorithm the left-to-right most general partial unifier of t_1 and t_2 or more simply an MGPU of t_1,t_2.

Let W be a non-empty set of expressions. We compute the first difference set of W by locating the first symbol (counting from the left) at which not all the expressions in W have exactly the same symbol and then extracting from each expression the subexpression that begins with the symbol occupying that position. The set of these respective differences is the first difference set of W.

If we resume the comparison in each expression of W at the first symbol after the subexpression used to define the first difference set, find the next point of disagreement and again extract the corresponding subexpressions, we obtain the set of expressions which comprise the second difference set of W. If the elements of W are not identical, we can in this fashion construct k difference sets: $d_1,d_2,\ldots,d_k$, $k \geq 1$.

Now let us state the algorithm to compute an MGPU substitution for W.

MGPU Algorithm:

1. Set $j = 1$, $k = 0$, $W_k = W$, $\sigma_k = \{\ \}$.
2. Find d_j for W_k as previously described; if it does not exist, terminate with σ_k an MGPU of W.
3. If d_j contains as members a variable v and a term t which does not contain v then let $\sigma_{k+1} = \sigma_k \cup \{t/v\}$ and $W_{k+1} = W_k\{t/v\}$, i.e. we apply the substitution $\{t/v\}$ to W_k. Set $k = k+1$ and go to 2.
4. If d_j does not contain the above then let $j = j+1$ and go to 2. □

4. The RUE Rule of Inference

We now define the first of our two rules of inference:

"Given the clauses $A+P(s_1,\ldots,s_n)$ and $B+\bar{P}(t_1,\ldots,t_n)$ and a substitution σ, the RUE resolvent of $\sigma(A + P(s_1,\ldots s_n))$ and $\sigma(B + \bar{P}(t_1,\ldots,t_n))$ is $\sigma A + \sigma B + D$ where D is the disjunction of the inequalities specified by a disagreement set of the complementary literals σP and $\sigma\bar{P}$."

σ may be the null-substitution, the MGPU of $\{(s_i,t_i)\ \ i=1,n\}$ or any substitition. The advantage of using the MGPU is that it produces the shortest resolvent. The standard resolution by unification is simply RUE resolution using as σ the most general unifier of $\{(s_i,t_i)\ \ i=1,n\}$.

We have the theorem:

(4.1) THEOREM. Resolution by unification and equality is sound.

Proof. The proof follows from the generalized form of the substitution axiom for predicates, namely:

$$t_{k_1} = t'_{k_1} \wedge t_{k_2} = t'_{k_2} \wedge \ldots \wedge t_{k_n} = t'_{k_n}$$
$$\wedge\ P(\ldots t_{k_1}\ldots t_{k_2}\ldots t_{k_n}\ldots) \to P(\ldots t'_{k_1}\ldots t'_{k_2}\ldots t'_{k_n}\ldots)$$

The arguments of the P literals are identical except at t_{k_i},t'_{k_i}. This substitution axiom states that the truth value of a predicate P does not change if we substitute for one or more arguments of P an equivalent term.

We have the conjunction

$$(1)\ (\sigma A + \sigma P(s_1,\ldots,s_n)) \wedge (\sigma B + \sigma\bar{P}(t_1,\ldots,t_n))$$

and the disagreement set $D = \bigcup_1^n D_i$ here D_i is a disagreement set of $(\sigma s_i, \sigma t_i)$.

D is simply a set of pairs, $D = \{(d_i, d_i') \quad i=1,\ell\}$. Now either the terms in each pair of D are equal, in which case the corresponding arguments of σP and $\sigma\bar{P}$ are equal and we can deduce $\sigma A + \sigma B$ from (1) by applying the substitution axiom for predicates, or one or more pairs in D contain unequal terms and the disjunction $\Sigma_1^\ell\, d_i \neq d_i'$ is true.

This proves that

$$(\sigma A + P(s_1,\ldots,s_n)) \wedge (\sigma B + \sigma\bar{P}(t_1,\ldots,t_n)) \to \sigma A + \sigma B + D$$

where D is $\Sigma_1^\ell\, d_i \neq d_i'$. □

5. The NRF Rule of Inference

To build-in equality and so eliminate the need to explicitly use the equality axioms, we need a second rule of inference, the Negative Reflexive Function Rule:

"Given $A + t_1 \neq t_2$ and a substitution σ, the NRF resolvent of $\sigma A + \sigma t_1 \neq \sigma t_2$ is $\sigma A + D$ where D is the disjunction of the inequalities specified by a disagreement set of $(\sigma t_1, \sigma t_2)$."

Since the most important application of the NRF rule is the case where $t_1 \neq t_2$ is $f(a_1,\ldots,a_n) \neq f(b_1,\ldots,b_n)$, we have called the rule the negative reflexive function rule.

(5.1) Theorem. The Negative Reflexive Function Rule is sound.

Proof. In the case where $t_1 \neq t_2$ has the form $f(a_1,\ldots,a_n) \neq f(b_1,\ldots,b_n)$ the NRF rule is derived from the equality substitution axiom:

$$x_0 = x_k \to f(\ldots x_{k-1}, x_k, x_{k+1}, \ldots) = f(\ldots x_{k-1}, x_0, x_{k+1}, \ldots)$$

which states that a function does not change its value if one (or more) of its arguments is replaced by an equal.

If in $\sigma A + f(\sigma a_1,\ldots,\sigma a_n) \neq f(\sigma b_1,\ldots,\sigma b_n)$ all pairs in the disagreement set D of the inequality can be proven equal then by the substitution axiom $f(\sigma a_1,\ldots,\sigma a_n) = f(\sigma b_1,\ldots,\sigma b_n)$ since corresponding arguments are equal. This erases the inequality in $\sigma A + f(\sigma a_1,\ldots,\sigma a_n) \neq f(\sigma b_1,\ldots,\sigma b_n)$ leaving σA. If on the other hand one or more pairs in the disagreement set cannot be proven equal, then the conclusion is D, the disjunction of the inequalities of the disagreement set. So we may validly conclude:

$$\sigma A + f(\sigma a_1,\ldots,\sigma a_n) \neq f(\sigma b_1,\ldots,\sigma b_n) \to \sigma A + D. \quad \square$$

The following is an example of an application of the NRF rule using the MGPU as σ:

$$f(x, g(b), h(a)) \neq f(a, g(x), h(b)) \xrightarrow[\text{MGPU}]{\text{NRF}} a \neq b$$

And similarly by using RUE, $P(f(x,g(b),h(a)))$ and $\bar{P}(f(a,g(x),h(b)))$ resolve to $a \neq b$.

6. Completeness of RUE and NRF

We define an *RUE-NRF deduction* as:

"Given a set of clauses S, an RUE-NRF deduction of C from S is a finite sequence $C_1, C_2, \ldots, C_k$ such that each C_i either is a clause in S or an RUE-NRF resolvent of clauses preceding C_i and where $C_k \equiv C$."

A deduction of the empty clause from S is called an RUE-NRF refutation of S. In [11] we prove the following completeness theorem:

(6.1) Theorem. A set of clauses S is E-unsatisfiable if and only if there is an RUE-NRF deduction of the empty clause from S.

The proof of completeness is obtained by showing that any refutation using the equality axioms with resolution by unification can be restated as an RUE-NRF refutation without the equality axioms. These axioms are removed by relatively simple transformation rules.

7.

As an example, let us show the RUE-NRF refutation of the clause set:

S: 1. $a = b$
2. $c = b$
3. $g_2(a) = g_1(e)$
4. $\bar{P}(g_2(c))$
5. $\bar{P}(g_1(e)) + g(f(h(a),x,h(x))) \neq g(f(h(b),c,h(a)))$.

The reader is invited to prove the E-unsatisfiability of S and to count the number of times particular equality axioms are used in the proof. Below we have the refutation by RUE-NRF:

$P(g_2(c))$ $\quad \bar{P}(g_1(e)) + g(f[\ldots]) \neq g(f[\ldots])$

rue

$g_2(c) \neq g_1(e) + g(f[\ldots]) \neq g(f[\ldots])$

rue $\quad g_2(a) = g_1(e)$

$c \neq a + g(f[h(a),x,h(x)]) \neq g(f[h(b),c,h(a)])$

nrf with $\sigma = \{c/x\}$

$c \neq a + a \neq b$ (merging into $c \neq a$)

$c = b$

$a \neq b$ (merging into $a \neq b$)

$a = b$

□

In the proof we uniformly used the MGPU and the bottom-most disagreement set since it was advantageous. Merging was also performed. The equality axioms do not appear in the refutation since they are incorporated into the RUE and NRF rules of inference.

To appreciate the simplicity of this proof of 5 steps, one should examine the corresponding proof using the equality axioms with resolution by unification. The proof contains 16 steps. An equivalent paramodulation proof has 10 steps. The RUE proof is not only shorter but more transparent.

8.

In respect to the definitions we have given there are some important questions to be considered.

In both RUE and NRF what substitution should be chosen? Choosing the MGPU will produce the shortest resolvent since it minimizes the disagreement set produced. However, the following example shows that the exclusive use of the MGPU will sometimes prevent the generation of a refutation. The clause set S:

S: 1. $P(f(x),g(y)) + P(h(x),i(y))$
2. $\bar{P}(f(a),g(b))$
3. $\bar{P}(h(b),i(b))$
4. $f(a) = f(c)$
5. $h(c) = h(b)$

is refutable only if we use the substitution $\sigma = \{c/x, b/y\}$ and here we cannot use the MGPU in resolving P and $\bar{P}$ if we are to obtain a refutation. The RUE refutation of S is:

$P(h(x),i(y)) + P(f(x),g(y))$

$\sigma = \{b/y\}$

$\bar{P}(h(b),i(b))$

$h(x) \neq h(b) + P(f(x),g(b))$

$\sigma = \{c/x\}$

$h(c) \neq h(b)$

$P(f(c),g(b))$

$\bar{P}(f(a),g(b))$

$f(c) \neq f(a)$

$f(a) = f(c)$

□

In the first application of RUE the required substitution is $\sigma = \{b/y\}$ and not the MGPU $\sigma = \{b/y, b/x\}$. Hence, for completeness we must do more than use the MGPU. The algorithms we state in (12.1) and (13.2) resolve this issue to successfully preserve completeness. In algorithm (12.1) the null substitution is used until the algorithm determines the substitution required for a refutation.

Irrespective of what substitution is chosen for RUE and NRF, there is the further problem of choosing a disagreement set. We would like each time we use RUE or NRF to be able to choose only one disagreement set and yet preserve completeness. If a resolvent is to participate in a refutation, then the disagreement inequalities must be erased by positive equality literals appearing in S or derivable from S. Later we will use this as the basis for a method to select D.

What appears to be attractive, i.e., to select the bottom-most disagreement set, will not always work as is evident in the following simple example:

S: 1. $P(f(a))$ 2. $\bar{P}(f(b))$ 3. $f(a) = f(b)$.

Resolving $P(f(a))$ with $\bar{P}(f(b))$ to obtain $a \neq b$ is useless. Here we must use the topmost disagreement set in order to obtain a refutation.

9. Selection of a Disagreement Set

First consider the lemma:

(9.1) Lemma. A necessary condition that $t_1 = t_2$ can be proven in S, a clause set, is that:

(1) the literals appear in S
$\{r_1 = r_1', r_2 = r_2', \ldots, r_k = r_k'\}$, $k \geq 1$
and there is a substitution σ such that in
$\{\sigma r_1 = \sigma r_1', \sigma r_2 = \sigma r_2', \ldots, \sigma r_k = \sigma r_k'\}$, $k \geq 1$
we have that $t_1 \equiv \sigma r_1$ or t_1 and σr_1 represent the same function whose corresponding arguments can be proven equal in S. The same is true for $(\sigma r_i', \sigma r_{i+1})$, i=1,k-1 and for $(\sigma r_k', t_2)$.

or

(2) $t_1 = t_2$ has the form $f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)$ where $a_i = b_i$ can be proven in S, i=1,n.

The lemma is true because to prove an equality relationship we must use the positive equality literals appearing in S, together with instantiation, transitivity and substitution of equals in functions. Actually more is involved since in the clause $A + s_1 = s_2$ to take advantage of $s_1 = s_2$ we must erase A. The above is a necessary but not sufficient condition that we can prove $t_1 = t_2$ in S.

Note that in the lemma we state that the transitivity link between $\sigma r_i = \sigma r_i'$ and $\sigma r_{i+1} = \sigma r_{i+1}'$ must be that $\sigma r_i'$ and σr_{i+1} are either identical or match on function symbol. It is this property that we intend to use to define a restriction for the application of RUE and NRF.

We now apply this lemma to a refutation using RUE and NRF to establish the definition of a *viable disagreement set* or a *viable resolvent* and also to define a restriction for RUE when applied to a complementary pair of equality literals.

An RUE or NRF resolvent is *viable*, i.e. *can possibly participate in a refutation*, only if it satisfies the condition that:

(1) For each $s_i \neq t_i$ in the disagreement set of the resolvent we have that for s_i there is a term *a* which is the argument of a positive equality literal in S such that s_i either unifies with *a* or matches *a* on leading function symbol with (s_i, a) having a viable disagreement set. The same is also true for t_i and some other term *b* which is the argument of a positive equality literal in S.

(2) In the case that there are some $s_i \neq t_i$ which do not satisfy the above, then the set of these inequalities has a MGU which converts their logical sum to the empty clause.

Furthermore, the substitutions used to satisfy the above condition must be compatible so as to form a single composite substitution. The empty disagreement set is always considered viable.

The above is a necessary but not sufficient condition that we can prove $s_i = t_i$, $i = 1,k$ in S and thus erase these inequalities in the resolvent.

Consider the example:

S:	1. $f(x) = b$	5. $P(f(a))$
	2. $b = c$	6. $\bar{P}(g(d))$
	3. $c = d$	7. $\bar{P}(g(a))$
	4. $d = g(b)$	

If we resolve $P(f(a))$ with $\bar{P}(g(d))$ we obtain $f(a) \neq g(d)$ which is viable since $f(a)$ unifies with $f(x)$ in (1) and $g(d)$ matches-on-function-symbol with $g(b)$ in (4) with $d \neq b$ being viable. We can in fact refute S by resolving $P(f(a))$ with $\bar{P}(g(d))$. On the other hand, if we resolve $P(f(a))$ with $\bar{P}(g(a))$ we obtain $f(a) \neq g(a)$ which is not viable. We cannot refute S by resolving these clauses.

We may now state our

(9.2) Rule for Selecting the Disagreement Set in RUE and NRF:

"In resolving $P(s_1, \ldots, s_n)$ and $\bar{P}(t_1, \ldots, t_n)$ by RUE or in reducing $t_1 \neq t_2$ by NRF choose as D the topmost viable disagreement set."

We choose the viable disagreement set nearest the topmost as D because by using the NRF rule we can derive from D any lower level viable disagreement set in case it is needed for a refutation. When none of the disagreement sets in RUE and NRF are viable, then it is useless to form the resolvent because it cannot appear in a refutation. Our rule of selection will yield either one or no resolvent when we apply RUE and NRF. Furthermore, the rule retains completeness.

The weakness of the viability test lies in the fact that when S contains the literal x = y or the literals x = t and y = t' then all inequalities become viable and the filtering effect is lost.

10. The Equality Restriction for RUE

We now formulate the *equality restriction* which applies to RUE acting on complementary equality literals:

"The RUE resolution of $A + s_1 = s_2$ and $B + t_1 \neq t_2$ is permitted only if at least one pair in the set

$\{(s_1,t_1) \quad (s_1,t_2) \quad (s_2,t_1) \quad (s_2,t_2)\}$

unifies or matches-on-function-symbol. In the latter case the pair must have a viable disagreement set."

Hence, to resolve equality literals we must satisfy both the above equality restriction and be able to select a topmost viable disagreement set. These two conditions appear similar but they are not the same. In finding a topmost viable disagreement set the matching by unification or matching-on-function-symbol need not occur as described above but may take place with other literals of S.

Consider the E-unsatisfiable set:

$$S:\quad \begin{array}{llllll} 1. & a \neq b & 3. & a = e & 5. & b = f \\ 2. & c = d & 4. & e = c & 6. & f = d \end{array}$$

$a \neq b$ should not be resolved with $c = d$ because the equality restriction is not satisfied even though the resolvent has a viable disagreement set. Out of the five candidates which resolve with $a \neq b$ only two satisfy the equality restriction, namely $a = e$ and $b = f$. The RUE refutation of S can be achieved within the constraints of the equality restriction.

The equality restriction is compatible with completeness because if $t_1 \neq t_2$ is to be erased in a refutation then by Lemma (9.1) we need only take advantage of the transitivity chain $r_1 = r_1'$, $r_2 = r_2'$, $\ldots, r_k = r_k'$ which exists to prove $t_1 = t_2$ or first apply the NRF rule to $t_1 \neq t_2$ and afterwards use transitivity.

The equality restriction as stated for RUE will permit us to resolve $t_1 \neq t_2$ with $r_1 = r_1'$ using a substitution σ to obtain the disagreement $t_2 \neq \sigma r_1'$, which is permitted to resolve against $\sigma r_2 = \sigma r_2'$, leading to the inequality $t_2 \neq \sigma r_2'$, until finally we get $t_2 \neq \sigma r_k'$. This last inequality is handled by the NRF rule if t_2 and $\sigma r_k'$ are not identical. Also the possible inequalities $t_1 \neq \sigma r_1$, $\sigma r_1' \neq \sigma r_2$, $\ldots, \sigma r_{k-1}' \neq \sigma r_k$ are handled by the NRF rule. We are saying in effect that the equality restriction permits traversal of any transitivity chain which may be required in a refutation.

This restriction on RUE resolution of complementary equality literals should prevent a good number of unneeded resolvents. Since both RUE and NRF generate inequalities, the above restriction is important.

11.

Let us now return to our previous example and apply both the viability and equality restrictions. We have

$$S:\quad \begin{array}{l} 1.\ a = b \\ 2.\ b = c \\ 3.\ g_1(e) = g_2(a) \\ 4.\ P(g_2(c)) \\ 5.\ \bar{P}(g_1(e)) + \\ \quad g(f[h(a),x,h(x)]) \neq g(f[h(b),c,h(a)]) \end{array}$$

$P(g_2(c))$ can resolve against $\bar{P}(g_1(e))$ because the single existing disagreement set $\{[g_2(c),g_1(e)]\}$ is viable. However

(1) $g(f[h(a),x,h(x)]) \neq g(f[h(b),c,h(a)])$

cannot resolve against clauses 1, 2 or 3 because the equality restriction is not satisfied. But we can apply the NRF rule for which we must select a topmost viable disagreement set.

Suppose we first apply the MGPU substitution $\{c/x\}$ to (1) and then find D. We have the topmost disagreement set:

$$D_1 = \{(f[h(a),c,h(c)],\ f[h(b),c,h(a)])\}$$

which is not viable; then proceeding to a lower level:

$$D_2 = \{[h(a),\ h(b)],\ [h(c),\ h(a)]\}$$

which is not viable. Finally

$$D_3 = \{[a,b],\ [c,a]\}$$

the bottom-most disagreement set alone is viable. Hence

$$(1) \rightarrow a \neq b + c \neq a$$

We have thus far deduced:

$\bar{P}(g_1(e)) + g(f[h(a),x,h(x)]$
$\qquad\qquad \neq g(f[h(b),c,h(a)])$
$\quad$ rue $\quad P(g_2(c))$

$g_1(e) \neq g_2(c) + g(f[h(a),x,h(x)])$
$\qquad\qquad \neq g(f[h(b),c,h(a)])$
$\qquad\qquad$ nrf with $\sigma = \{c/x\}$

$g_1(e) \neq g_2(c) + a \neq b + c \neq a$

Now $g_1(e) \neq g_2(c)$ by the equality restriction is allowed only to resolve with $g_1(e) = g_2(a)$ yielding $c \neq a$, so that we now have after merging:

$$c \neq a + a \neq b$$

which is easily reduced to the empty clause. In this example a level saturation algorithm using the equality and viability restrictions would converge rapidly to a refutation.

12. A Level-Saturation Algorithm for Applying RUE-NRF

The proof of completeness of RUE-NRF given in [11] is based on the existence of a composite substitution which is obtained from a refutation using resolution by unification on the conjunction of S with the equality axioms. Whatever algorithm we use to apply RUE-NRF must be capable of generating this substitution or one which is logically equivalent in the sense that it too leads to a refutation.

When we apply RUE or NRF a substitution is used (possibly the null substitution). Using the MGPU at each step of RUE or NRF leads to the shortest resolvents but unfortunately as the example in (8) shows, the substitution required for a refutation need not be the MGPU.

The following algorithm preserves completeness by postponing substitutions until the full effect of equality relationships can be realized.

(12.1) Level Saturation Algorithm:

(1) Given S, set $S^{-1} = \{\ \}$, $S^0 = S$, $i = 0$.

(2) Using only the null substitution form all possible RUE-NRF resolvents from a pair of parent clauses taken from S^i where at least one parent is in $S^i - S^{i-1}$ and where:

  (a) the disagreement set of the resolvent is the topmost viable disagreement set

  (b) when RUE is applied to a complementary pair of equality literals, the equality restriction is satis-satisfied. The symmetry of equality must be exploited.

(3) If a resolvent in (2) is made up of negative equality literals, test for an MGU substitution which reduces the resolvent to the empty clause, i.e., the resolvent has the form $\sum_1^k s_i \neq t_i$ and σ exists such that in $\sum_1^k \sigma s_i \neq \sigma t_i$, each pair $(\sigma s_i, \sigma t_i)$ matches identically.
If this occurs, terminate with the refutation based on the substitution σ. If not, proceed to (4).

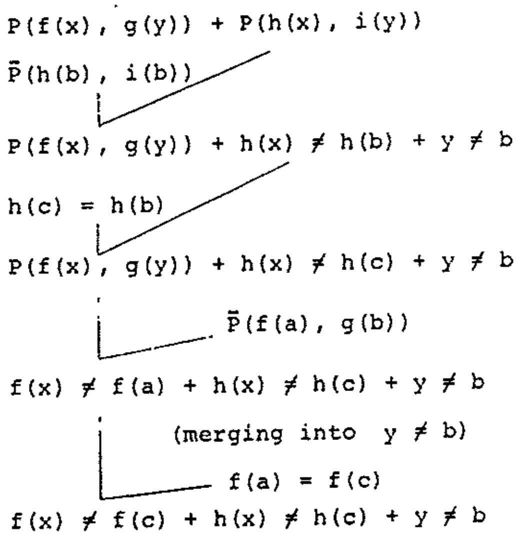
(4) Let $S^{i+1} = S^i \cup$ {resolvents produced in (2)}; let $i = i + 1$ and go to (2). □

Consider the E-unsatisfiable input clause set:

S: 1. $P(f(x), g(y)) + P(h(x), i(y))$
2. $\bar{P}(f(a), g(b))$
3. $\bar{P}(h(b), i(b))$
4. $f(a) = f(c)$
5. $h(c) = h(b)$

Note that the substitution required for a refutation is $\{c/x, b/y\}$ and that it is not the MGPU in either of the P, $\bar{P}$ resolutions.

The proof generated by the level saturation algorithm for S is:



P(f(x), g(y)) + P(h(x), i(y))
P̄(h(b), i(b))
P(f(x), g(y)) + h(x) ≠ h(b) + y ≠ b
h(c) = h(b)
P(f(x), g(y)) + h(x) ≠ h(c) + y ≠ b
P̄(f(a), g(b))
f(x) ≠ f(a) + h(x) ≠ h(c) + y ≠ b
(merging into y ≠ b)
f(a) = f(c)
f(x) ≠ f(c) + h(x) ≠ h(c) + y ≠ b

which vanishes when $\sigma = \{c/x, b/y\}$ is applied.

13. RUE-NRF Hyperresolution

Suppose we are given S to prove E-unsatisfiable and that it contains neither the equality nor functionally reflexive axioms. We partition S into two subsets: E, the electron set, consisting of positive clauses, and N, the nucleus set, containing the nonpositive clauses of S. A clause in N must have at least one negative literal. Let P be any ordering of the predicate symbols in S.

We define a *clash* denoted by

$$\{E_1, E_2, \ldots, E_q, N_j\}$$

where $E_i \in E$ and $N_j \in N$ as representing the succession of RUE-NRF resolvents:

$$(E_1, N_j) \rightarrow R_1$$
$$(E_2, R_1) \rightarrow R_2$$
$$\ldots$$
$$(E_q, R_{q-1}) \rightarrow R_q \quad \textit{(the clash resolvent)}$$

where each R_i is obtained either

(a) by RUE of E_i against R_{i-1} where we must resolve on the largest literal in E_i in accord with the P-ordering

or

(b) by applying the NRF rule to a negative equality literal in R_{i-1}.

R_q must be a positive clause and R_i, $i < q$, will be nonpositive. Note that q is greater than or equal to the number of negative literals in N_j as both RUE and NRF may add negative equality literals to a resolvent. We understand that in the notation $\{E_1, \ldots, E_q, N_j\}$ one or more of the E_i may be replaced by the NRF rule. The substitution σ used in RUE and NRF is unrestricted and may be null.

We define an RUE-NRF hyperresolution deduction as:

"Given a set of clauses S, an RUE-NRF hyperresolution deduction of C from S is a finite sequence $C_1, C_2, \ldots, C_k$ such that each C_i either is a clause of S or an RUE-NRF clash resolvent of clauses preceding C_i and where $C_k \equiv C$."

A hyperresolution deduction of the empty clause from S is called a refutation of S by hyperresolution. In [11] we prove the following completeness theorem:

(13.1) Theorem. A set of clauses S is E-unsatisfiable if and only if there is an RUE-NRF hyperresolution deduction of the empty clause from S.

The proof of completeness is obtained by showing that any hyperresolution refutation using the equality axioms with resolution by unification can be restated as an RUE-NRF hyperresolution refutation without the equality axioms.

We now state an algorithm for RUE-NRF hyperresolution:

(13.2) RUE-NRF Hyperresolution Algorithm:

(1) Partition the input set S into $E^0 + N^0$, set $i = 0$, $\Delta E = \{\ \}$, $\Delta N = \{\ \}$. Let P be any ordering of the predicate symbols in S.

(2) For each nucleus $N_j \in N^i$ form all possible clashes of N_j against E^i as follows:

Using the null substitution, resolve by RUE-NRF on the k negative literals of N_j:

$$(E_1, N_j) \rightarrow R_1 + D_1$$
$$(E_2, R_1 + D_1) \rightarrow R_2 + \sum_1^2 D_i$$
$$\cdot \quad \cdot \quad \cdot$$
$$(E_k, R_{k-1} + \sum_1^{k-1} D_i) \rightarrow R_k + \sum_1^k D_i$$

In RUE we must resolve on the highest literal of E_i as specified by the P-ordering. D_i represents a disagreement set and R_k is a positive clause. σ *the null substitution is used at each step.* When RUE or NRF is applied we choose the topmost viable disagreement set and when we perform RUE on a pair of complementary equality literals, the equality restriction applies.

If $R_k + \sum_1^k D_i$ is not positive, we add it to ΔN which in effect continues the clash later on.

We compute the MGPU of $\sum_1^k D_i$, call it σ. (i.e., $\sum_1^k D_i$ corresponds to the disagreement set $\cup_1^k D_i$ which is a set of pairs $\{(d_i, d_i') \quad i=1,\ell\}$; σ is the simultaneous MGPU of each of these pairs; it corresponds to the MGPU we would obtain for the pair of terms: $f(d_1, \ldots, d_\ell)$ and $f(d_1', \ldots, d_\ell')$.)

If $\sigma \sum_1^k D_i$ vanishes, place σR_k in ΔE and if this clause is empty we terminate with a refutation.

If $\sigma \sum_1^k D_i$ does not vanish, we place $\sigma R_k + \sigma \sum_1^k D_i$ into ΔN.

Using all possible combinations of electrons from E^i against N_j, we generate all possible clashes with N_j as nucleus. We do this for all N_j in N^i.

(3) If both ΔE and ΔN are empty, the algorithm terminates without a refutation.

(4) Set $i = i+1$, $E^i = E^{i-1} \cup \Delta E$, $N^i = N^{i-1} \cup \Delta N$, $\Delta E = \{\ \}$, $\Delta N = \{\ \}$. Go to step (2). □

In our algorithm we have used in parallel both the null substitution and the MGPU. The null substitution is necessary to preserve completeness and the MGPU enhances efficiency by producing the shortest resolvents. It is not necessary to apply both substitutions in parallel and thereby produce two resolvents in each application of RUE and NRF. It is possible to interleave the use of the null substitution say every *kth* cycle of iteration. Exactly what is the best mode of interleaving can be better determined after experimentation with a theorem prover using the above algorithm. This work is currently in progress.

14. Conclusion

The essence of this presentation has been to accept equality as being as important as unification in resolution and to define resolution in terms of both equality and unification.

A theorem prover using RUE-NRF hyper-resolution has been implemented and is currently being used on the experiments proposed by McCharen, Overbeek and Wos [1]. These experiments will provide a basis for comparing the effectiveness of RUE-NRF with various restriction strategies using resolution by unification and paramodulation. These results will appear in [11].

References

[1] McCharen, Overbeek and Wos, "Problems and Experiments for and with Automated Theorem Proving Programs," IEEE Transactions on Computers, Vol. C-25, No. 8, August 1976.

[2] Darlington, J. L., "Automatic Theorem Proving with Equality Substitutions and Mathematical Induction", Machine Intelligence, Vol. 3, (B. Meltzer and D. Michie, eds.), American Elsevier, 1968.

[3] Robinson, G. and Wos, L., "Paramodulation and Theorem Proving in First Order Theories with Equality," Machine Intelligence, Vol. 4, 1969.

[4] Sibert, E. E., Jr., "A Machine Oriented Logic Incorporating the Equality Relation", Machine Intelligence, Vol. 4, 1969.

[5] Morris, J., "E-resolution: An Extension of Resolution to Include the Equality Relation", Proc. IJCAI, 1969.

[6] Anderson, R., "Completeness Results for E-resolution", Proc. AFIPS 1970 Spring Joint Computer Conference.

[7] Slagle, James, R., "Automatic Theorem Proving with Built-in Theories Including Equality, Partial Ordering and Sets", JACM, Vol. 19, No. 1, January 1972.

[8] Knuth, D. E. and Bendix, P. B., "Simple Word Problems in Universal Algebras," Computational Problems in Abstract Algebras, Ed. Leech, J., Pergamon Press.

[9] Brand, D., "Proving Theorems with the Modification Method", SIAM Journal of Computing, Vol. 4, No. 4, December 1975.

[10] Harrison, M. and Rubin, N., "Another Generalization of Resolution," JACM, Vol. 25, No. 3, July 1978.

[11] Digricoli, Vincent J., "Resolution by Unification and Equality," Technical Report, Dept. of Computer Science, Courant Inst. Math. Sci., June 1979.

Appendix

Below we have a comparison of an RUE-NRF proof of 5 *steps* compared to the equivalent proof using the equality axioms and resolution by unification which has 16 *steps*. We are speaking of the clause set S and refutation presented in (7) as compared to the refutation given below.

For this proof S is expanded from S to 16 clauses to include the equality axioms:

S:
1. $a = b$
2. $c = b$
3. $g_2(a) = g_1(e)$
4. $P(g_2(c))$
5. $\bar{P}(g_1(e)) + g(f[h(a),x,h(x)]) \neq g(f[h(b),c,h(a)])$
6. $x = x$
7. $x \neq y + y = x$
8. $x \neq y + y \neq z + x = z$
9. $x \neq y + \bar{P}(x) + P(y)$
10. $x \neq y + g(x) = g(y)$
11. $x \neq y + g_1(x) = g_1(y)$
12. $x \neq y + g_2(x) = g_2(y)$
13. $x \neq y + h(x) = h(y)$
14. $x \neq x' + f(x,y,z) = f(x',y,z)$
15. $y \neq y' + f(x,y,z) = f(x,y',z)$
16. $z \neq z' + f(x,y,z) = f(x,y,z')$

Refutation:

(4) $P(g_2(c))$

├— (9) $\bar{P}(x) + P(y) + x \neq y$

(R1) $P(y) + g_2(c) \neq y$

├— (5) $\bar{P}(g_1(e)) + g(f[\ldots]) \neq g(f[\ldots])$

(R2) $g_2(c) \neq g_1(e) + g(f[\ldots]) \neq g(f[\ldots])$

├— (8) $x \neq y + y \neq z + x = z$

(R3) $g_1(e) \neq z + g_2(c) \neq z + g(f[\ldots]) \neq g(f[\ldots])$

├— (3) $g_1(e) = g_2(a)$

(R4) $g_2(c) \neq g_2(a) + g(f[\ldots]) \neq g(f[\ldots])$

│ (8) $x \neq y + y \neq z + x = z$

│ ├— (1) $a = b$

│ (R5) $b \neq z + a = z$

│ ├— (2) $b = c$

│ (R6) $a = c$

│ ├— (12) $x \neq y + g_2(x) = g_2(y)$

├— (R7) $g_2(a) = g_2(c)$

(R8) $g(f[h(a),x,h(x)]) \neq g(f[h(b),c,h(a)])$

├— (10) $x \neq y + g(x) = g(y)$

(R9) $f[h(a),x,h(x)] \neq f[h(b),c,h(a)]$

│ (13) $x \neq y + h(x) = h(y)$

│ ├— (1) $a = b$

│ (R10) $h(a) = h(b)$

│ ├— (14) $x \neq x' + f(x,y,z) = f(x',y,z)$

│ (R11) $f[h(a),y,z] = f[h(b),y,z]$

│ ├— (8) $x \neq y + y \neq z' + x = z'$

│ (R12) $f[h(b),y,z] \neq z' + f[h(a),y,z] = z'$

│ │ (13) $x \neq y + h(x) = h(y)$

│ │ ├— (R6) $a = c$

│ │ (R13) $h(a) = h(c)$

│ │ ├— (16) $z = z' + f(x,y,z) = f(x,y,z')$

│ ├— (R14) $f[x,y,h(c)] = f(x,y,h(a))$

├— (R15) $f[h(a),y,h(c)] \neq f[h(b),y,h(a)]$

(R16) □ The empty clause.

Not counting the symmetry axiom the above proof has 16 steps.

Now consider the RUE proof we used in (7):

(4) $P(g_2(c))$

├— (5) $\bar{P}(g_1(e)) + g(f[\ldots]) \neq g(f[\ldots])$

(R1) $g_2(c) \neq g_1(e) + g(f[\ldots]) \neq g(f[\ldots])$

├— (3) $g_2(a) = g_1(e)$

(R2) $c \neq a + g(f[h(a),x,h(x)]) \neq g(f[h(b),c,h(a)])$

│ NRF with $\sigma = \{c/x\}$

(R3) $c \neq a + a \neq b$ (merging into $c \neq a$)

├— (2) $c = b$

(R4) $a \neq b$ (merging into $a \neq b$)

├— (1) $a = b$

(R5) □ The empty clause.

The RUE proof is both shorter and more transparent. One can read it and much more easily see the basis for the refutation. These attributes of brevity and transparency will probably lead to more effective heuristics for RUE-NRF theorem provers.

REDUCTION-SYSTEMS AND SMALL CANCELLATION THEORY

Hans Bücken

RWTH Aachen, Germany

SUMMARY: We obtain a (finitary) sufficient condition for the solvability of the wordproblem using systems of reductions. As corollaries we get some classical results on small cancellation groups.

Let $\mathrm{TERM} = \langle\langle x_1, x_2, \dots, f_1, \dots f_n\rangle, \langle f_1, \dots, f_n, \dots, f_m\rangle\rangle$ be the absolutely free algebra of type $\langle 0,.,0,d_{n+1}, \dots, d_m\rangle$, generated by $\langle x_1, x_2, \dots, f_1, \dots, f_n\rangle$. The x_i are called variables, the f_j functions of degree d_j. $f_1, \dots, f_n$ are the nullary functions, also called constants. The elements of TERM are called terms; a subterm of the term T is any term, that is contained in T. If T contains no variables, then T is a ground term. A finite system $\mathcal{R}$, $\mathcal{R} \subseteq \mathrm{TERM} \times \mathrm{TERM}$ is called a reduction-system.

For a mapping θ of the variables into TERM, its homomorphic extension $\mathrm{rep}_\theta : \mathrm{TERM} \to \mathrm{TERM}$ is called a substitution. When two terms T_1 and T_2 are unifiable, the most general unifier can be found by the unification algorithm [RO 65]. T_2 is called an immediate reduction of T_1 with respect to $\mathcal{R}$, denoted $T_1 \to T_2(\mathcal{R})$, in case T_2 is the result of replacing one occurence of $\mathrm{rep}_\theta(L)$ as a subterm in T_1 by $\mathrm{rep}_\theta(R)$ for some reduction $(L,R) \in \mathcal{R}$. "$\xrightarrow{+}$" is the transitive closure of "$\to$".

A set of reductions has the finite termination-property (FTP) if there is no infinite sequence $T_1 \to T_2 \to T_3 \to \dots$ of immediate reductions. T is called irreducible, if there is no immediate reduction of T. $\Delta^*(W,\mathcal{R}) := \{\overline{W} \mid W \xrightarrow{+} \overline{W}(\mathcal{R})\} \cup \{W\}$

$\mathrm{Irred}(W,\mathcal{R}) := \{\overline{W} \mid \overline{W} \in \Delta^*(W,\mathcal{R}),\ \overline{W} \text{ irreducible}\}$

Let $\mathcal{R}$ have FTP, then there exists an algorithm which computes an element $\overline{W} \in \mathrm{Irred}(W,\mathcal{R})$ and which always terminates. If $\mathfrak{A}_{\mathcal{R}}$ is such an algorithm, then $\mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(W) := \overline{W}$.

$\mathcal{R}$ has the Church-Rosser-property, if for any T and $T_1, T_2 \in \Delta^*(T,\mathcal{R}) \Rightarrow \Delta^*(T_1,\mathcal{R}) \cap \Delta^*(T_2,\mathcal{R}) \neq \phi$. (Sometimes this property is called confluence [HU 77]). $\mathcal{R}$ is complete if $\mathcal{R}$ has FTP and CR (Church-Rosser-prop.)

$E(\mathcal{R}) := \{L \equiv R \mid (L,R) \in \mathcal{R}\}$ is the equational system belonging to $\mathcal{R}$. $W_1 \equiv W_2(E)$ iff W_1 and W_2 belong to the same congruence class relative to the equational system E. The wordproblem with respect to E is the problem of deciding $W_1 \equiv W_2(E)$ for two given terms W_1, W_2.

For most of our theorems proofs are omitted. Their proofs can be found e.g. in: [KN-BE 70], [HU 77], [RI 78].

Theorem 1:

$\mathcal{R}$ is complete $\Rightarrow$ The wordproblem with respect to $E(\mathcal{R})$ is solvable.

One is interested in orderings $\succ$ of TERM which are compatible with substitutions, t.s. $T \succ U$ implies $\mathrm{rep}_\theta(T) \succ \mathrm{rep}_\theta(U)$ for all θ. An important class of such orderings was introduced by Knuth and Bendix. Although this ordering can be modified in various ways we use the original definition because it is sufficient for our purposes here.

First one takes the function $w : \{f_1, \dots, f_m\} \to \mathbb{N} \cup \{0\}$ s.t. i) $w(f_j) > 0$ if $j \leq n$

ii) $w(f_k) > 0$ if $k < m$ and $d_k = 1$.

w is called a weight-function.

Let w_0 be the minimum weight of a constant. We define the weight of an arbitrary term as

$w(T) := w_0 \cdot \sum_{k \geq 1} n(x_k,T) + \sum_{i \leq m} w(f_i)\, n(f_i,T)$,

where $n(x,T)$ is the number of occurences of x in T

Let us say $T \succ U$ iff

(1) $w(T) > w(U)$ and $n(x_k,T) \geq n(x_k,U)$ for $k \geq 1$

or (2) $w(T) = w(U)$ and $n(x_k,T) = n(x_k,U)$ for $k \geq 1$

and either $T = f_m(f_m(\dots(f_m(x_k)\dots), U = x_k$

or $T = f_j(T_1, \dots, T_{d_j}), U = f_k(U_1, \dots, U_{d_k})$

and either (2a) $j > k$ or

(2b) $j = k$, $T_1 = U_1, \dots, T_p = U_p$, $T_{p+1} \succ U_{p+1}$ for some p, $1 \leq p < d_j$

If $E = \{L \equiv R \mid L, R \in \mathrm{TERM}\}$ is a finite equational system, $\mathcal{R}_{KB}(E) := \{(L,R) \mid L \succ R,\ L \equiv R \in E \text{ or } R \equiv L \in E\}$.

Theorem 2:

i) '$\succ$' restricted to ground terms is a (total) well-ordering.

ii) $\mathcal{R}_{KB}(E)$ has the finite termination property.

For two terms L_1, L_2 we define

$\mathrm{superpos}(L_1,L_2) := \{W \mid W = \mathrm{rep}_\theta(L_1)$, rep_θ is most general unifier of L_2 and some subterm T of L_1)

For $W \in \text{superpos}(L_1, L_2)$ with $(L_i, R_i) \in \mathfrak{R}$, W_i immediate reductions of W by (L_i, R_i), $i \in \{1,2\}$, (W_1, W_2) and (W_2, W_1) are called <u>critical pairs belonging to W</u>.

Example:

$(L_1, R_1) = (f(a,g(x,b)),\ h(a,b))$ $\quad \theta: \begin{array}{l} x \to f(a,b) \\ y \to b \end{array}$

$(L_2, R_2) = (g(f(a,b),y),\ g(y,a))$

$$\begin{tikzcd}[column sep=small] & W = f(a,g(f(a,b),b)) \arrow[dl] \arrow[dr] & \\ W_1 = h(a,b) & & f(a,g(b,a)) = W_2 \end{tikzcd}$$

$\text{superpos}(\mathfrak{M}_1, \mathfrak{M}_2) = \bigcup (\text{superpos}(L_1, L_2) \mid L_i \in \mathfrak{M}_i)$

$L\mathfrak{R} = \{L \mid (L,R) \in \mathfrak{R}$ for some $\mathfrak{R}\}$

Theorem 3:

If $\text{Irred}_{\alpha_{\mathfrak{R}}}(W_1) = \text{Irred}_{\alpha_{\mathfrak{R}}}(W_2)$ for every critical pair (W_1, W_2) belonging to $W \in \text{superpos}(L\mathfrak{R}, L\mathfrak{R})$, then the wordproblem with respect to $E(\mathfrak{R})$ is solvable.

In case a reduction-system $\mathfrak{R}$ is incomplete, this theorem shows how one can try to "complete" the system: If $Y_1 = \text{Irred}_{\alpha_{\mathfrak{R}}}(W_1)$ and $Y_2 = \text{Irred}_{\alpha_{\mathfrak{R}}}(W_2)$, $Y_1 \neq Y_2$ and Y_1 and Y_2 are comparable, then add (Y_1, Y_2) to $\mathfrak{R}$ for $Y_1 \succ Y_2$ (resp. (Y_2, Y_1) for $Y_2 \succ Y_1$). (Knuth-Bendix-extension-algorithm [KN-BE 70])

Knuth and Bendix take as an example the algebra $\text{TERM} = \langle \{x,y,z,a_1,\dots,a_n,e\}, \langle e,a_1,\dots,a_n,"\cdot","^{-1}"\rangle\rangle$ of type $\langle 0,\dots,0,2,1\rangle$, choosing the following weights: $w(e) = 1$, $w(a_i) = 1$, $w(\cdot) = 0$, $w(^{-1}) = 0$. The equational system is $E = \{ex \equiv x,\ x^{-1}\cdot x \equiv e,$

$(x \cdot y) \cdot z \equiv x \cdot (y \cdot z)\}$

They succeed in completing $\mathfrak{R}_{KB}(E)$ (after omitting unneccessary reductions) to

$$\begin{aligned} \mathfrak{R}_F = \{&(e\cdot x, x),\ (x^{-1}\cdot x, e),\ ((x\cdot y)\cdot z, x\cdot(y\cdot z)),\ (e^{-1}, e)\\ &(x^{-1}\cdot(x\cdot y), y),\ ((x^{-1})^{-1}, x),\ (x\cdot(x^{-1}\cdot y), y),\\ &(x\cdot e, x),\ (x\cdot x^{-1}, e),\ ((x\cdot y)^{-1}, y^{-1}\cdot x^{-1})\}, \end{aligned}$$

thus solving the wordproblem for free groups.

Because of this, $T^{\sim 1} := \text{Irred}_{\alpha_{\mathfrak{R}_F}}(T^{-1})$ is uniquely defined, independent of the algorithm $\alpha_{\mathfrak{R}_F}$.

In the following we will always deal with this algebra TERM and the above described ordering.

Remark 1:

The following terms are not subterms of T:

i) $((T_1 \cdot T_2) \cdot T_3)$ for arbitrary terms T_1, T_2, T_3

ii) S^{-1}, except for $S \in \{a_1,\dots,a_n\}$

iii) e, except $T = e$

iv) $S \cdot S^{-1}$, $S^{-1} \cdot S$ for arbitrary S.

Therefore T can be written without brackets. A term written in this way will be called a <u>word</u>.

The weight $w(T)$ will be denoted by $|T|$. Because of the above chosen weight, for $T = b_1 \cdot \dots \cdot b_m$, $b_j \in \{a_1,\dots,a_n,a_1^{-1},\dots,a_n^{-1}\}$, $|T| = m$.

$T = b_1 \cdot \dots \cdot b_m$ is called <u>cyclically reduced</u> if b_m is not inverse to b_1. A subset Υ of words is called <u>symmetrized</u> if all elements of Υ are cyclically reduced and for each $T \in \Upsilon$ all cyclic permutations of T and $T^{\sim 1}$ also belong to Υ.

Suppose T_1 and T_2 are two distinct irreducible (with respect to $\mathfrak{R}_F$) elements of the form $T_1 = BC_1$, $T_2 = BC_2$. Then B is called a <u>piece of Υ</u>.

Let λ be a positive real number, Υ cyclically reduced. We define:

<u>Υ satisfies $C'(\lambda)$</u>: If $S \in \Upsilon$, $S = BC$ where B is a piece of Υ then $|B| < \lambda |S|$.

Υ satisfies $T(h)$: Suppose $S_1,\dots,S_h$ is a sequence of elements of Υ with no successive elements S_i, S_{i+1} an inverse pair. Then at least one of the products $S_1 \cdot S_2$, $S_2 \cdot S_3, \dots, S_h \cdot S_1$ is irreducible with respect to $\mathfrak{R}_F$.

Theorem 4: (Dehn, Schiek, Tartakovskij, etc.)

$G = \langle a_1,\dots,a_n \mid \Upsilon \rangle$, Υ a finite symmetrized set of defining relations and Υ satisfies $C'(1/6)$ or $C'(1/4)$ and $T(4)$. If $W \in G$ and $W \equiv e(\Upsilon)$, $W \neq e$, W irreducible with respect to $\mathfrak{R}_F$ then W contains more than half of some element of Υ.

Theorem 4 presents sufficient conditions for the correctness of the classical algorithm of Dehn (If $U \cdot V \in \Upsilon$ for $|U| > |V|$ then Dehn's algorithm allows to replace U by $V^{\sim 1}$). See e.g. [LY-SCHU 77]

Our aim is to prove this result by means of reduction-systems. Let $G = \langle a_1,\dots,a_n \mid \Upsilon \rangle$ be a finitely presented group s.t. $\Upsilon \neq \emptyset$. Υ shall satisfy $C'(1/2)$. Put $M_\Upsilon := \{Sx \equiv x \mid S \in \Upsilon\}$. We want to complete $\mathcal{R}_{KB}(M\ )$ relative to $\mathcal{R}_F$.

We define inductively a sequence $\mathcal{R}_i$ of reduction-systems:

$\mathcal{R}_o := \mathcal{R}_{KB}(M_\gamma)$. For $i \geq 0$ we take

$\mathcal{P}_i := \{(T_1,T_2) \mid (T_1,T_2)$ critical pair belonging to some $M \in \text{superpos}(L\mathcal{R}_F, L\mathcal{R}_i)\}$

$\text{Irred}(\mathcal{P}_i) := \{(\text{Irred}_{\alpha_{\mathcal{R}_F \cup \mathcal{R}_i}}(T_1), \text{Irred}_{\alpha_{\mathcal{R}_F \cup \mathcal{R}_i}}(T_2)) \mid (T_1,T_2) \in \mathcal{P}_i\}$

$N(\mathcal{P}_i) := \{(T_1,T_2) \mid T_1 \neq T_2,\ T_1 \succ T_2,\ (T_1,T_2)$ or (T_2,T_1) element of $\text{Irred}(\mathcal{P}_i)\}$

$\mathcal{R}_{i+1} := \{(\overline{T}_1,\overline{T}_2) \mid \overline{T}_1 \succ \overline{T}_2,\ \overline{T}_j = \text{Irred}_{\alpha_{\overline{\mathcal{R}}}}(T_j),\ j \in \{1,2\}$
$(T_1,T_2) \in \mathcal{R}_i \cup N\mathcal{P}_i$ or $(T_2,T_1) \in \mathcal{R}_i \cup N\mathcal{P}_i$
where $\overline{\mathcal{R}} = \mathcal{R}_F \cup \mathcal{R}_i \cup N\mathcal{P}_i \setminus \{(T_1,T_2),(T_2,T_1)\}\}$

Theorem 5:

Suppose $G = \langle a_1,\dots a_n \mid \gamma \rangle$, γ symmetrized and satisfies $C'(\frac{1}{2})$. Then:

For some n_o we get

$\mathcal{R}_{n_o} = \{(Lx,Rx),\ (L,R) \mid L \cdot R^{\sim 1} = S \in \gamma\}$; it satisfies furthermore:

i) $\mathcal{R}_{n_o+1} = \mathcal{R}_{n_o}$

ii) a) Is $(L,R) \in \mathcal{R}_{n_o}$, $L = L'b \Rightarrow (Rb^{\sim 1},L') \in \mathcal{R}_{n_o}$

b) Is $(L,R) \in \mathcal{R}_{n_o}$, $L = bL' \Rightarrow (b^{\sim 1} \cdot R,L') \in \mathcal{R}_{n_o}$

Remark 2:

Theorem 5 gives an example where the effect of some relatively complicated applications of reduction-systems is of simple group-theoretic nature. Another example is "symmetrizing": Just as one can "symmetrize a set γ of words", the above algorithm "symmetrizes" a set of reductions.

If one analyses the role of reductions of the form (Lx,Rx), one sees that in the groundcase they are unnecessary if one omits the brackets. For reason of simplicity we will omit the brackets as well as these reductions and call the remaining system $\mathcal{R}_\gamma$ and put $\mathcal{R} = \mathcal{R}_F \cup \mathcal{R}_\gamma$.

Remark 3:

Any algorithm $\alpha_\mathcal{R}$ is an extension of Dehn's algorithm; it is a proper extension, because reductions (L,R) with $|L| = |R|$ are also allowed.

Remark 4:

As the word $S \cdot e$ $(e \cdot S)$ will always be reduced to $\text{Irred}_{\alpha_\mathcal{R}}(S)$, we will not distinguish between S, $e \cdot S$ and $S \cdot e$; for example $A \cdot e \cdot B$ will be called irreducible if $A \cdot B$ is irreducible.

Lemma 1:

Suppose $(\tilde{L}_i,\tilde{R}_i)$, $(\tilde{L}_j,\tilde{R}_j) \in \mathcal{R}_\gamma$, $W \in \text{superpos}(\tilde{L}_i,\tilde{L}_j)$, and (W_1,W_2) critical pair belonging to W.

Then: $\tilde{L}_i = L_iA$, $\tilde{L}_j = AL_j$, $A \neq e$,
$W = L_iAL_j$, $W_1,W_2 \in \{\tilde{R}_iL_j,\ L_i\tilde{R}_j\}$

$(\tilde{L}_i,\tilde{R}_i)$, $(\tilde{L}_j,\tilde{R}_j)$ are called reductions belonging to W.

The proof is by inspection.

Definition:

Let (W_1,W_2) be a critical pair belonging to $W \in \text{superpos}(L\mathcal{R}_\gamma, L\mathcal{R}_\gamma)$. A sequence $(X_n,Y_n,\dots,X_o,Y_o,W)$ is called a superpos-deduction-chain of length n if:

i) If $n = 0$, then $X_o = W_1$, $Y_o = \text{Irred}_{\alpha_\mathcal{R}}(W_1)$

ii) If $n > 0$, then each X_l, $0 < l \leq n$ is of the form $X_l = L_{i_k} \cdot Z_{i_{k-1}}^{\sim 1} \cdot \ldots \cdot L_{i_1} \cdot Z_{i_o}^{\sim 1} \cdot Y_o \cdot Z_{j_o}^{\sim 1} \cdot L_{j_1} \cdot \ldots \cdot Z_{j_{m-1}}^{\sim 1} \cdot L_{j_m}$, $k+m = l$, for some words Z_{i_s}, Z_{j_t} possibly equal to e and some words L_{i_s}, L_{j_t} not equal to e, subject to the following conditions:

either i) $X_l = L_{i_k} \cdot Z_{i_{k-1}}^{\sim 1} \cdot X_{l-1}$, and

ii) $L_{i_k} \cdot Z_{i_{k-1}}^{\sim 1} \cdot \ldots \cdot L_{i_1} \cdot Z_{i_o}^{\sim 1}$ is irreducible

iii) $Z_{i_{k-1}}^{\sim 1} \cdot Y_{l-1}$ reducible with respect to $\mathcal{R}_F$ or $Z_{i_{k-1}} = e$,

iv) For $Y_l^F = \text{Irred}_{\alpha_{\mathcal{R}_F}}(Z_{i_{k-1}}^{\sim 1} \cdot Y_{l-1})$ we have $L_{i_k} \cdot Y_l^F$ is reducible with respect to $(\tilde{L}_{i_k},\tilde{R}_{i_k}) \in \mathcal{R}_\gamma$

v) $Y_l = \text{Irred}_{\alpha_\mathcal{R}}(L_{i_k} \cdot Y_l^F)$

or i') $X_l = X_{l-1} \cdot Z_{j_{m-1}}^{\sim 1} \cdot L_{j_m}$, and

ii') $Z_{j_o}^{\sim 1} \cdot L_{j_1} \cdot \ldots \cdot Z_{j_{m-1}}^{\sim 1} \cdot L_{j_m}$ is irreducible iii') $Y_{l-1} \cdot Z_{j_{m-1}}^{\sim 1}$ reducible with respect to $\mathcal{R}_F$

or $Z_{j_{m-1}} = e$

iv') For $Y_l^F = \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}_F}}(Y_{l-1} \cdot Z_{j_{m-1}}^{\sim 1})$ we have

$Y_l^F L_{j_m}$ is reducible with respect to

$(\tilde{L}_{j_m}, \tilde{R}_{j_m}) \in \mathcal{R}_\gamma$

v') $Y_l = \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(Y_l^F L_{j_m})$

$(\tilde{L}_{i_k}, \tilde{R}_{i_k})$ (resp. $(\tilde{L}_{j_m}, \tilde{R}_{j_m})$) are called reductions belonging to X_l.

If $X_n = U_1 Y_o U_2$ we put $X_n(W_2) := U_1 W_2 U_2$.

Example:

$(L_k \cdot A_k, R_k) \in \mathcal{R}_\gamma$ for $k \in \{0,1,2\}$, $(A_i L_i, R_i) \in \mathcal{R}_\gamma$ for $i \in \{4,5\}$
$(A_o L_3, R_3) \in \mathcal{R}_\gamma$

$$\begin{array}{lll}
W = L_o \cdot A_o \cdot L_3 & L_2 L_1 \underline{L_o A_o} L_3 L_4 L_5 & W_2 = L_o R_3 \\
 & \swarrow \quad \longrightarrow & \\
W_1 = R_o L_3 & L_2 L_1 \underline{R_o L_3} L_4 L_5 & L_2 L_1 L_o R_3 L_4 L_5 = X_4(W_2) \\
 & \downarrow^{*} & \\
Y_o = A_1 U_o & L_2 \underline{L_1 A_1} U_o L_4 L_5 & \\
 & \downarrow & \\
X_1 = L_1 A_1 U_o & L_2 \underline{R_1 U_o} L_4 L_5 & X_1(W_2) = L_1 L_o R_3 \\
 & \downarrow^{*} & \\
Y_1 = U_1 A_4 & L_2 U_1 \underline{A_4 L_4} L_5 & \\
 & \downarrow & \\
X_2 = L_1 A_1 U_o L_4 & L_2 \underline{U_1 R_4} L_5 & X_2(W_2) = L_1 L_o R_3 L_4 \\
 & \downarrow^{*} & \\
Y_2 = U_2 A_5 & L_2 U_2 \underline{A_5 L_5} & \\
 & \downarrow & \\
X_3 = L_1 A_1 U_o L_4 L_5 & L_2 \underline{U_2 R_5} & X_3(W_3) = L_1 L_o R_3 L_4 L_5 \\
 & \downarrow^{*} & \\
Y_3 = A_2 U_3 & \underline{L_2 A_2} U_3 & \\
 & \downarrow & \\
X_4 = L_2 L_1 A_1 U_o L_4 L_5 & \underline{R_2 U_3} & X_4(W_3) = L_2 L_1 L_o R_3 L_4 L_5 \\
 & \downarrow^{*} & \\
Y_4 = U_4 & U_4 &
\end{array}$$

Remark 5:

The $Z_{i_k}^{\sim 1}$ (or the $Z_{j_m}^{\sim 1}$) just express that a part of Y_{l-1} is possibly cancelled by the reductions of $\{(x \cdot x^{-1}, e), (x^{-1} x, e)\}$. So $Y_{l-1} = Z_{i_k} \cdot Y_l^F$, or $Y_{l-1} = Y_l^F Z_{j_m}$.

Theorem 6:

Suppose $G = \langle a_1, \ldots, a_n \mid \gamma \rangle$, γ satisfies $C'(1/2)$ and for all $W \in \mathrm{superpos}(L\mathfrak{S}_\gamma, L\mathcal{R}_\gamma)$ and all superpos-deduction- chains $(X_m, Y_m, \ldots, X_o, Y_o, W)$, $m \geq 0$, we have $\mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(Y_m^{\sim 1} \cdot X_m(W_2)) = e$.
Then: $U \equiv e(\mathcal{R})$ implies $\mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(U) = e$.

Proof:

Let $U \equiv e(\mathcal{R})$ be irreducible with respect to $\mathcal{R}_F$.
There is a word $U' \equiv U(E(\mathcal{R}_F))$, s.t.
$U' = T_1^{\sim 1} S_{i_1} \cdot T_1 \cdot \ldots \cdot T_k^{\sim 1} S_{i_k} \cdot T_k$, T_j words of G, $S_{i_j} \in \mathcal{R}$
Obviously U' can be reduced to e by $\mathcal{R}$.
Assumption:
There is an algorithm $\mathfrak{A}_{\mathcal{R}}$ with $\mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(U') \neq e$.

Let $\overline{W} = \min_{<}\{W \mid W \in \Delta^*(U', \mathcal{R}),\ W \to \overline{W}_1(\mathcal{R}),\ W \to \overline{W}_2(\mathcal{R})$
$\mathrm{Irred}(\overline{W}_1, \mathcal{R}) = \{e\}$ and
$e \notin \mathrm{Irred}(\overline{W}_2, \mathcal{R})\}$.

$\overline{W}$ exists because $\mathcal{R}$ has FTP and because of the assumption.
Then $\overline{W} = U_1 W U_2$, U_1, U_2 irreducible (because of the minimality of $\overline{W}$), $\overline{W}_i = U_1 W_i U_2$, $i \in \{1,2\}$, (W_1, W_2) critical pair belonging to W.
As $e \in \mathrm{Irred}(\overline{W}_1, \mathcal{R})$ there is a superpos-deduction-chain $(X_m, Y_m, \ldots, X_o, Y_o, W)$ describing the reduction of $\overline{W}_1$ to e, with
$X_m = U_1 \cdot \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(W_1) \cdot U_2$, $Y_m = e$
Then $e = \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(Y_m^{\sim 1} \cdot X_m(W_2)) = \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(e^{\sim 1} \cdot X_m(W_2))$
$= \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(X_m(W_2)) = \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(W_2)$.
This contradiction proves the theorem. □

Definition:

$\mathcal{R}$ satisfies K1 :⟺ If $\mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(W_i) \neq \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}_F}}(W_i)$
for some critical pair (W_1, W_2)
then $\mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(W_1) = \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(W_2)$

Lemma 2 :

Suppose $\mathcal{R}$ satisfies K1, $W \in \mathrm{superpos}(\tilde{L}_i, \tilde{L}_j)$, (W_1, W_2) critical pair, $W \to W_i(\tilde{L}_i, \tilde{R}_i)$ and $\mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}}}(W_i) = \mathrm{Irred}_{\mathfrak{A}_{\mathcal{R}_F}}(W_i)$, $i \in \{1,2\}$
Then:
$\tilde{L}_i = L_i B_i^{\sim 1} A \qquad \tilde{R}_i = R_i B_j$

$\tilde{L}_j = A \cdot B_j^{\sim 1} \cdot L_j$, $\qquad \tilde{R}_j = B_i \cdot R_j$

$\mathrm{Irred}_{\alpha_{\mathfrak{R}}}(W_1) = R_i \cdot L_j$, $\qquad \mathrm{Irred}_{\alpha_{\mathfrak{R}}}(W_2) = L_i \cdot R_j$

The proof is by inspection.

Definition:

The notation is as in lemma 1: Assume $\mathfrak{R}$ satisfies K1:

A) $\bar{\bar{R}}_{i_o}$ is called block in $(\tilde{L}_{i_1}, \tilde{L}_{i_o}, \tilde{L}_{j_o})$ $:\Longleftrightarrow$

i) $R_{i_o} = Z_1 \cdot \bar{\bar{R}}_{i_o}$

ii) There is a superpos-deduction-chain (X_1, Y_1, X_o, Y_o, W) s.t.

a) $X_1 = L_{i_1} \cdot X_{i_o}^{\sim 1} \cdot Z_1 \cdot \bar{\bar{R}}_{i_o} \cdot L_{j_o}$

b) $\mathrm{Irred}_{\alpha_{\mathfrak{R}}}(X_1) \neq \mathrm{Irred}_{\alpha_{\mathfrak{R}}}(X_1(W_2))$

c) $Y_1 = \mathrm{Irred}_{\alpha_{\mathfrak{R}}}(L_{i_1} X_{i_o}^{\sim 1} Z_1) \cdot \bar{\bar{R}}_{i_o} \cdot L_{j_o}$

B) $\bar{\bar{L}}_{j_o}$ is called a block in $(\tilde{L}_{i_o}, \tilde{L}_{j_o}, \tilde{L}_{j_1})$ $:\Longleftrightarrow$

i) $L_{j_o} = \bar{\bar{L}}_{j_o} \cdot Z_2$

ii) There is a superpos-deduction-chain (X_1, Y_1, X_o, Y_o, W) s.t.

a) $X_1 = R_{i_o} \cdot \bar{\bar{L}}_{j_o} \cdot Z_2 X_{j_o}^{\sim 1} L_{j_1}$

b) $\mathrm{Irred}_{\alpha_{\mathfrak{R}}}(X_1) = \mathrm{Irred}_{\alpha_{\mathfrak{R}}}(X_1(W_2))$

c) $Y_1 = R_{i_o} \cdot \bar{\bar{L}}_{j_o} \mathrm{Irred}_{\alpha_{\mathfrak{R}}}(Z_2 X_{j_o}^{\sim 1} L_{j_1})$

C) $\mathfrak{R}$ satisfies K2 $:\Longleftrightarrow$

Let $(X_1^i, Y_1^i, X_o, Y_o, W)$ be superpos-deduction-chains of $W \in \mathrm{superpos}(\tilde{L}_{i_o}, \tilde{L}_{j_o})$, $\mathrm{Irred}_{\alpha_{\mathfrak{R}}}(X_1^i) \neq \mathrm{Irred}_{\alpha_{\mathfrak{R}}}(X_1^i(W_2))$, $i \in \{1,2\}$.

Then there is a block $\bar{\bar{R}}_{i_o}$ in $(\tilde{L}_{i_1}, \tilde{L}_{i_o}, \tilde{L}_{j_o})$ and a block $\bar{\bar{L}}_{j_o}$ in $(\tilde{L}_{i_o}, \tilde{L}_{j_o}, \tilde{L}_{j_1})$.

D) Let $\bar{\bar{R}}_{i_o}$ be a block in $(\tilde{L}_{i_1}, \tilde{L}_{i_o}, \tilde{L}_{j_o})$,

$\bar{\bar{L}}_{j_o}$ be a block in $(\tilde{L}_{i_o}, \tilde{L}_{j_o}, \tilde{L}_{j_1})$.

$\bar{\bar{R}}_{i_o} \cdot \bar{\bar{L}}_{j_o}$ is called invincible if

i) $\bar{\bar{R}}_{i_o} \cdot \bar{\bar{L}}_{j_o}$ is irreducible.

ii) $\bar{\bar{R}}_{i_o} = \bar{\bar{R}}'_{i_o} \cdot b \Longrightarrow b \cdot \bar{\bar{L}}_{j_o}$ is not subword of some $L \in L\mathfrak{R}_{\Upsilon}$

iii) $\bar{\bar{L}}_{j_o} = b \cdot \bar{\bar{L}}'_{j_o} \Longrightarrow \bar{\bar{R}}_{i_o} \cdot b$ is not subword of some $L \in L\mathfrak{R}_{\Upsilon}$

E) $\mathfrak{R}$ satisfies K3 $:\Longleftrightarrow$

i) $\mathfrak{R}$ satisfies K1 and K2

ii) If $\bar{\bar{R}}_{i_o}$ is block in $(\tilde{L}_{i_1}, \tilde{L}_{i_o}, \tilde{L}_{j_o})$, $\bar{\bar{L}}_{j_o}$ is block in $(\tilde{L}_{i_o}, \tilde{L}_{j_o}, \tilde{L}_{j_1})$,

then $\bar{\bar{R}}_{i_o} \cdot \bar{\bar{L}}_{j_o}$ is invincible.

Lemma 3:

Assume $\mathfrak{R}$ satisfies K3 and Υ satisfies $C'(1/4)$, $W \in \mathrm{superpos}(\tilde{L}_{i_o}, \tilde{L}_{j_o})$, $(X_n, Y_n, \ldots, X_o, Y_o, W)$ superpos-deduction-chain, $W \longrightarrow X_o(\tilde{L}_{i_o}, \tilde{R}_{i_o})$,

$X_n = L_{i_k} Z_{i_{k-1}}^{\sim 1} \cdot \ldots \cdot Z_{i_o}^{\sim 1} \cdot Y_o \cdot Z_{j_o}^{\sim 1} \cdot \ldots \cdot Z_{j_{m-1}}^{\sim 1} \cdot L_{j_m}$,

and $\mathrm{Irred}_{\alpha_{\mathfrak{R}}}(X_k) \neq \mathrm{Irred}_{\alpha_{\mathfrak{R}}}(X_k(W_2))$ for all $k \in \{1, \ldots, n\}$. Let $(M_t, N_t) \in \mathfrak{R}_{\{\tilde{L}_t \cdot \tilde{R}_t^{\sim 1}\}}$.

Then:

$Y_n = \bar{R}_{i_k} \cdot \bar{\bar{R}}_{i_{k-1}} \cdot \ldots \cdot \bar{\bar{R}}_{i_1} \cdot \bar{\bar{R}}_{i_o} \cdot \bar{\bar{L}}_{j_o} \cdot \bar{\bar{L}}_{j_1} \cdot \ldots \cdot \bar{\bar{L}}_{j_{m-1}} \cdot \bar{L}_{j_m}$

$\bar{\bar{R}}_{i_l}$ is block in $(\tilde{L}_{i_{l+1}}, \tilde{L}_{i_l}, M_{i_{l-1}})$

$\bar{\bar{R}}_{i_{l-1}}$ is block in $(\tilde{L}_{i_l}, M_{i_{l-1}}, M_{i_{l-2}})$

$\bar{\bar{L}}_{j_s}$ is block in $(M_{j_{s-1}}, \tilde{L}_{j_s}, \tilde{L}_{j_{s+1}})$

$\bar{\bar{L}}_{j_{s-1}}$ is block in $(M_{j_{s-2}}, M_{j_{s-1}}, \tilde{L}_{j_s})$

and $\bar{\bar{R}}_{i_l} \cdot \bar{\bar{R}}_{i_{l-1}}$ and $\bar{\bar{L}}_{j_{s-1}} \cdot \bar{\bar{L}}_{j_s}$ are invincible.

The proof is by induction on l and s.

Theorem 7:

Suppose $\mathfrak{R}$ satisfies K3 and $C'(1/4)$.

$W \in \mathrm{superpos}(\tilde{L}_{i_o}, \tilde{L}_{j_o})$, $(X_n, Y_n, \ldots, X_o, Y_o, W)$ superpos

-deduction-chain of length n.

Then: $\mathrm{Irred}_{\mathfrak{R}_{\gamma}}(Y_n^{\sim 1} \cdot X_n(W_2)) = e$.

Proof: Use lemma 3 and the irreducibility of

$X_n(W_2)$.

Theorem 8:

If γ satisfies $C'(1/6)$ or $C'(1/4)$ and $T(4)$ then $\mathfrak{R}$ satisfies K3.

The proof is by inspection.

Corollary:

Dehn's algorithm solves the wordproblem for $G = \langle a_1, \ldots, a_n \mid \gamma \rangle$ if γ satisfies $C'(1/6)$ or $C'(1/4)$ and $T(4)$.

Proof: Theorem 8 and theorem 6.

One can show, that the assumption $C'(1/6)$ in the corollary is sharp; take for instance $G = \langle a,b,c \mid a \cdot b \cdot c \cdot a^{-1} \cdot b^{-1} \cdot c^{-1} \rangle$:

$$\begin{aligned} \mathfrak{R}_{\gamma} = \{ & (cba, abc),\ (a^{-1}cb, bca^{-1}),\ (b^{-1}a^{-1}c, ca^{-1}b^{-1}), \\ & (c^{-1}b^{-1}a^{-1}, a^{-1}b^{-1}c^{-1}),\ (b^{-1}c^{-1}a, ac^{-1}b^{-1}), \\ & (c^{-1}ab, bac^{-1}),\ (abca^{-1}, cb),\ (bca^{-1}b^{-1}, a^{-1}c), \\ & (ca^{-1}b^{-1}c^{-1}, b^{-1}a^{-1}),\ (bac^{-1}b^{-1}, c^{-1}a) \} \end{aligned}$$

$$\begin{array}{lll} W_2 & = & abababcbca^{-1}ca^{-1}b^{-1}a^{-1}b^{-1}c^{-1}b^{-1}c^{-1}b^{-1}c^{-1} \quad \text{is irreducible} \\ & & \qquad\quad \uparrow \\ W & = & ababab\underline{ca^{-1}cb}ca^{-1}b^{-1}a^{-1}b^{-1}c^{-1}b^{-1}c^{-1}b^{-1}c^{-1} \\ & & \qquad\quad \downarrow \\ W_1 & = & ababcbcbca^{-1}b^{-1}a^{-1}b^{-1}c^{-1}b^{-1}c^{-1}b^{-1}c^{-1} \\ & & \qquad\qquad\qquad \downarrow \\ & & \qquad\qquad\qquad \vdots \\ & & \qquad\qquad\qquad \downarrow \\ & & \qquad\qquad\qquad e \end{array}$$

W reduces to e and the irreducible element W_2.

The assumption in theorem 6 is not finitary, but the assumption in theorem 7 is finitary. As one can see from the proofs, all theorems remain true, if one chooses other weight-functions $\overline{W}$ s.t. $\overline{W}(e) = 1$. The corresponding corollary gives Greendlingers theorem in [GRE 60].

Theorem 6 and theorem 7 are not just helpful for small cancellation groups, but for other groups, too. For instance, in slightly modifying the assumptions of theorem 7, $G = \langle a,b \mid a^3b, abab \rangle$ can be shown to have a reduction-system $\mathfrak{R}_G$, achieved by the Knuth-Bendix-extension-algorithm, which solves the wordproblem for G. This is also an example of a proper extension of Dehn's algorithm.

References:

[BRI 56]	Britton,J.L.: Solution of the Word Problem for Certain Types of Groups, I. Proc. Glasgow Math. Ass.3, (1956) 45-54
[DE 11]	Dehn,M.: Über unendliche diskontinuierliche Gruppen, Math.Ann.71 (1911) 116-144
[DE 12]	Dehn,M.: Transformation der Kurven auf zweiseitigen Flächen, Math.Ann.72 (1912) 413-421
[GRE 60]	Greendlinger,M.: Dehn's Algorithm for the Word Problem, Comm.Pure Appl.Math.13 (1960) 67-83
[HU 77]	Huet,G.: Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. Rapp.de Recherche No.25o, IRIA, Rocquencourt (1977)
[KN-BE 70]	Knuth,D.E.,Bendix,P.B.: Simple Word Problems in Universal Algebras, Computational Problems in Abstract Algebras, J.Leech, Ed., Pergamon Press (1970) 263-297
[LA 75]	Lankford,D.: Canonical Inference, Report ATP-25,Dec.1975, Departments of mathematics and computer sciences University of Texas at Austin
[RI 78]	M.M.Richter: Logikkalküle Teubner, Stuttgart (1978)
[LY-SCHU 77]	Lyndon, R.C., Schupp,P.E.: Combinatorial Group Theory, Springer Verlag (1977)

[RO 65]	Robinson,J.A.: A Machine-Oriented Logic Based on the Resolution Principle, J.A.C.M. Vol.12(1965) 23-41
[SCHU 73]	Schupp,P.E.: A Survey of Small Cancellation Theory, Word Problems, W.W.Boone, Ed., Studies in Logic, Vol.71, North-Holland (1973) 569-591
[TART 52]	Tartakovsky: Am.Math.Soc.Translations 60 (1952), Reprint 1, 12o-228

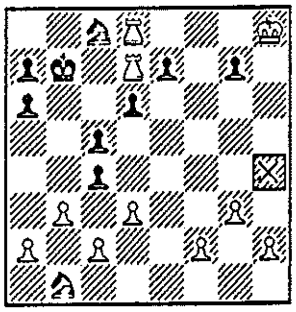
Observation and Inference Applied in a Formal Representation System

Robert Elliot Filman
Artificial Intelligence Laboratory
Stanford University
Stanford, California

1. Introduction

An intelligent computer program must have both a representation of its knowledge, and a mechanism for manipulating that knowledge in a reasoning process. This work is an attempt to formalize the expression and solution of a difficult problem within a machine manipulable form.

Our consideration centers on the following retrograde chess analysis puzzle (*figure 1*). Its solution (from basic chess principles) is certainly beyond the ability of any current computer program. Formally and completely expressing the solution of this chess puzzle is a difficult enough task; this paper reports on that expression.



A piece has fallen off of the board from the square marked X. What piece was it? This position was achieved in a legal chess game, though there is no presumption that either player was playing to win.

figure 1

This problem was chosen because its solution "requires" both deductive and observational inferences, in a context isolated from other issues of correctness and sufficiency.

The notion of deductive inference, obtaining new proof steps by the application of syntactic inference rules, ought to be familiar to the reader. We recognize, however, that human reasoning proceeds not only by deduction, but also by the immediate recognition of results, a process we identify with *observation*. We have extended our representational system to include observational inference by performance of computation in a semantic model.

Within the context of expressing the solution to retrograde chess problems, we achieve a synthesis of the two approaches. In particular, we have axiomatized the rules of chess within *first order logic* (the declarative representation), but include within this system a method for computing (when an effective procedure is known) the values of predicates and functions (the computational representation). This form of procedural representation allows not only efficient computation of known predicates and functions, but also provides a method for talking about, in the formal language, these predicates and functions. We call this set of semantic predicates and functions the *Chess Eye*.

We shall also highlight the representational decisions made in this axiomatization, discussing both the necessity for these particular choices, and their implications for designers of other representational systems.

Using a proof checker for first order logic, FOL [Weyhrauch 77], we have detailed a proof for the solution of the chess puzzle of *figure 1*, including proofs for almost all of the necessary associated lemmas [Filman 79]. In the process, we have shown the close correspondence between the *formal* solution to the problem, and an *informal, descriptive* analysis.

2. Inference

Human problem solving proceeds in many different fashions. Humans are capable of *deduction*, applying syntactic rules to previous inferences. This has been the reasoning mode most frequently employed in A.I. programs. However, generally intelligent systems will need to reason by other schemes, such as *induction* (reasoning from particular cases to a general conclusion), *analogy* (modifying reasoning from a solution of another problem to fit the current situation), and what we call *observation* (quickly noticing an apparent conclusion, performed in our system by computation in a semantic model). If we are to create such programs, we must find a pragmatic representational mechanism, one that can support these other kinds of reasoning. Effectively, we are looking for what McCarthy calls *epistemologically adequate* representation formalisms [McCarthy 77].

The work described in this paper is concerned primarily with the qualities required of formalisms capable of simultaneously representing complex deductive and observational reasoning. Deduction is obtaining conclusions by application of syntactic rules. The term *observation* is usually applied to human inference in several ways; they are unified by the notion of drawing a quick and immediate conclusion by examination. Examples of human observation are seen in the repairman noticing the failure of some mechanism, the mathematician recognizing the value of some simple function, or the game player recognizing a familiar pattern of pieces.

Our computer systems will need to be able to unify such observational action with their deductive results. We seek to accomplish this while preserving the deductive properties of the observational predicates and functions; that is, we wish to speak about them in our formal language and manipulate them in our deductive logic. We would also like to do all this in a manner that retains the mathematical validity accruing from using a formal logical system.

This examination considers the adequacy of the following scheme: that knowledge which is to manipulated in deductive form is represented in a *extended first order logic* formalism. Associated with this logical system is a computational LISP model of these axioms. We permit the results of the evaluation of constant functions on constant objects in this model to be legal inferences in the deductive, logical system. This shall serve as our notion of *observation*. This is not to imply that functional evaluation captures all of observation; we are perfectly willing to hook whatever hardware device onto this system (for example, a television camera or ADC) as is necessary. Nevertheless, we feel that this notion of functional evaluation expresses the necessary elements that computers will need for their observational ability, and that whatever other observational mechanism are later invented could easily be incorporated into such a system.

3. Epistemology and Heuristics

This work is subsumed under the idea that the general artificial intelligence problem is better attacked by separating the issues of the representation of knowledge (a notion also know as *epistemology* or *competence*) and those of selecting which knowledge is to be used in solving some particular problem (an issue of *search*, *heuristics* or *performance*) [McCarthy 77, Chomsky 65, Pratt 77].

This division has several beneficial side effects. An appropriate epistemology can be *task independent*. Similarly, one does not wish to have to design a new knowledge representation for every new problem.

Separating the knowledge out from the search process permits the employment of *multiple heuristic strategies*, so that many different algorithms can be applied in trying to reach a particular goal. A single representational formalism permits the selection of the appropriate strategy at the problem solution time, rather than tailoring a new strategy to each new representation (and each representation to a particular strategy).

There is often scientific benefit to *dividing* a problem into pieces, and then solving these pieces separately. The A.I. problem naturally divides into representational and heuristic parts.

And finally, a good question to ask the designer of any A.I. program is, "What did you tell it, and what did it figure out by itself?" The use of an explicit and separate epistemology can answer that question and remove (or instill) the appropriate doubts in the reader's mind.

4. Epistemological Adequacy

A primary question of a candidate representation is, *is it adequate*? That is, in this formalism, and for this domain, will we be able to both express the range of problem situations, and make the appropriate inferences? Is the resulting derivation short enough to be useful for our application?

Now, these are murky criteria. If we play an "epistemology game", where we invent a representation, and you find an unrepresentable "counter example" (except for the most trivial domains) we will run out of paper and patience long before you run out of exceptions. Reality is a very complex thing. Even natural language is inadequate to completely model it. The best we can hope for is to select some small piece of the world, and formalize that part.

Similarly, the only criterion we have for "appropriate" reasoning is comparison with the processes of the other known reasoning engines, people. A machine derivation that is exponentially longer than a human's is probably unsatisfactory. A machine derivation quite shorter than a human solution is a cause for suspicion.

5. Basic Formalisms

We need a basic foundation on which to build our representation. For that, we have chosen an *extended first order logic*. It is the usual predicate calculus, with the addition of *sorts*, *equality*, *functions* and the ability to do *computation in a semantic model*.

One might well ask, "Why logic?" After all, logic has a bad name in the artificial intelligence community, principally the result of the unsuccessfully attempt at applying general purpose theorem provers to sets of clauses back in the late sixties. The battles of that time seem to have been won by the proceduralists, whose rules allowed them to program whatever interactions they pleased. But, in retrospect, perhaps the moral of those times is that successful A.I. programs must not only have pure knowledge of their domain, but know how to use it. In fact, representing *knowledge about using knowledge* in a uniform epistemological fashion is an important open research question for A.I..

However, our attention is centered upon epistemological issues. From this perspective, we like formal logic for several reasons:

- The sentences of first order logic are *natural*. That is, they're easily understood.
- First order logic retains the *explicit quantification* lacking in some other representations.
- Historically, there has been much work in logic, and many *semi-decision procedures* for first order logic already exist. Any system employing logic can avail itself of this work. The various forms of resolution are examples of such procedures.
- Logic is a *general* representation. It is obviously not contrived for one particular domain. We are discovering the expression of many notions such as *concept*, *knowledge* and *observation* that can be expressed within this logic [McCarthy 79, Moore 79, Filman 79].
- Declarative representations in general, and first order logic in particular, are easily and uniformly

extensible. This beneficial extensibility is found in two different respects. When we wish to add additional information to the system, we can merely create an axiom having that knowledge; if new notions arise, we can define new predicates and functions.

These, however, are the minor reasons. There are two important justifications for the use of first order logic for A.I. formalisms.

• First, we like first order logic for providing us with a good *semantic model.* The [Tarski] semantics of formal logic imply a clear concept of meaning.

• Secondly, almost all other current representations, be they microplanner, semantic nets, KRL, or whatnot, are variations on the rules of formal logic; essentially, *the others are logic, anyway.* That a frame might provide some heuristic advantages, or a semantic net contain some useful implicit axioms, we don't deny; however, they are all declarative systems, and equivalent in terms of epistemology. Reliance on pure logic has two positive attributes: we are isolating the epistemological issues from the implementation, and we are effectively speaking the *lingua franca* of representation languages. Any particular network or schematic formalism would have required considerable explanation on our part. Now one may wish to argue that one's "frame" or "semantic net" provides heuristic (organizational) advantages over a pure "list of WFF's". That's fine, but it is not the issue we are concerned with in this work.

There's a caveat to this claim, the presence of the notions such as THNOT in microplanner. THNOT X is true if the machine is unable to deduce X; as such, its a comment about the state of the heuristic system, rather than the representational formalism. Now, certainly this is a useful heuristic tool. However, systems which are limited to identifying THNOT with NOT will inevitably have trouble expressing certain relations and ideas.

6. Proof Generation and Proof Checking

A fully intelligent computer program will need not only an epistemologically adequate world view, but also an appropriate set of heuristic search procedures to manipulate this knowledge. This reasoning program is somewhat beyond our current abilities. In any case, we don't need the full ability of this reasoning program to judge candidate epistemological formalisms. Rather, we need to have our deductions certified, not automatically generated. Clearly, the ability to accept a valid deduction is a prerequisite to the actual generation of that deduction. And also clearly, an epistemologically adequate formalism can be tested by the use of a proof checker. This is essentially the *Missouri program* of McCarthy-Hayes [McCarthy 69], and is in the tradition of the *Advice Taker* [McCarthy 58].

We are fortunate to have here at the Stanford A.I. Lab the program FOL [Weyhrauch 77, Filman 76], an extended natural deduction proof checker for first order logic. FOL will act as our Missouri program, checking the validity of our candidate inferences.

We mentioned that FOL was a proof checker for an *extended* sorted first order logic. There are two particular extensions to FOL of interest to us. FOL has been extended to incorporate a *tautology decider* for propositional logic with equality. And, more important to us, with our interest in observational interaction, FOL has the ability to create a *semantic model* of one's world, and to evaluate the values of functions and predicates in that model, returning the result to the deduction level. Thus, if one has the function + in one's logical language, and the LISP function *PLUS* in one's model, and had associated, or *ATTACH*ed (using the Semantic Attachment mechanism) these two together, we would be able to make a legal inference such as:

```
*****SIMPLIFY 3 + 2;
```

to which FOL would respond:

```
1) 3 + 2 = 5
```

The simplify mechanism is able to evaluate the values of predicates and functions on constant objects, and to check quantified statements for all members of finite sets. This kind of simplification is fairly satisfactory for simulating our notion of observation in chess.

7. Chess Puzzles

At the beginning of this paper, we mentioned a certain chess puzzle (*figure 1*). While space limitations preclude describing the solution of that puzzle, the reader should notice that, not only is the solution very complex, but that it also requires steps of two different forms. Any solution will exhibit both deduction, inference of the form "Both sides can't be in check at the same time, black is in check, therefore white is not in check", and *observation,* inference of the form "I see black is in check", or "There was no black piece that could have moved from there." We perceive our task as finding the appropriate incorporation of this observational reasoning into the deductive framework. Incidently, the fallen piece in the puzzle of *figure 1* was the white queen's bishop. The reader is referred to [Filman 79] for the detailed solution.

8. Objects of the Chess World

If the reader solves the puzzle for himself, he will surely perceive how complicated chess reasoning can be. Besides complexity, chess (as a problem domain) has another appealing attribute: we can *unequivocally* solve this problem; no reader ought to raise any correctness objections. Chess is a completely specified domain. This serves to help us to isolate the interesting features.

We proceed to outline our formalization of the chess world. Space constraints limit how much we can say about our axiomatization; we touch here only the more important notions.

Axioms usually talk about objects, and we begin by presenting the different kinds of objects present in the chess world. First of all, we have the notion of *Boards*. Most chess problems are presented in terms of a chess board. That is, the eight squares by eight squares, with the white king over here, and a black pawn down there. There are sixty four *Squares* on any chessboard.

Of course, if we are to speak of white queens and black knights, we need a name for this type of object, too. These are the *Values*.

These values are, however, to be distinguished from the chesspieces themselves; after all, the white queen's knight is not the same as the white king's knight; a promoted pawn has not always been a queen. Our reasoning will need to make these distinctions. Hence, we name each of the 32 chessmen with a unique, individual name, obtaining the set of *Pieces*.

All of the above, while somewhat sketchy, ought to appear fairly natural. Now, we present a kicker. We need to speak of the various things that must have happened to reach this point in the game, the capture that must have occurred, and so forth. Even the chess rules, in delimiting castling moves and draw conditions, refer to the entire game history. Hence, we declare an explicit set of historical states, the *Positions*, which can be though of as the history (set of moves) employed in reaching this particular board. Notice that for any given board, there will be many different games that could have been played to achieve it; hence, we will have to associate a "variable" position with any board, and deduce properties of all the positions (games that could have reached it) associated with that board.

There is also the natural notion of the *Moves*, the explicit, discrete transitions between the positional states, and the *Colors*, black and white, for the two chess armies.

9. Predicates and Functions

Of course, monadic predicates do not make an exciting universe. We will mention only a few of the more interesting predicates and functions.

The fundamental predicate in the system is that of *SUCCESSOR(p1,p2)* defined on two positions. If one position, *p2*, is the successor of another, *p1*, then *p2* can be reached by a legal move from *p1*.

If *SUCCESSOR* is to be the succession function of our system, then like arithmetic, we want a notion of "less than", which is called *PREDEGAME*. *PREDEGAME(p1, p2)* is true when *p1* occurred in the game that reached *p2*. It can (of course) be "determined" by "examining" *p2* if *p1* happened in its play. Naturally, for every legal game, the initial position, *P0*, has the predegame relation to that game. In our axiomatization, this becomes: $\forall p.\ PREDEGAME(P0, p)$.

Any given position will have associated with it the chessboard resulting from playing that game. We call this the *Tboard* of that position (for total-board). As the various pieces have their own individual identities, we can refer to the piece occupying a given square in a particular position. Thus, if *px* is a position that was played to reach the problem board, then *Pos(px, BKR1)* = *WK* (on *BKR1* (black king's rook one) in *px* is the *WK* (white king)).

Recall that we made a distinction between the values on a given chessboard, and the piece names themselves. This is partially because chessboards don't mention which piece is on a given square, but, more importantly, because some pieces (pawns) change their value in the course of a game. Thus, we see two other natural functions; we state that *Valueon(b, sq)* = *v* if the value on the board *b* on the square *sq* is *v*, and *Val(p, x)* = *v* if the value in position *p* of the piece *x* is *v*. Note that, for all non-pawn pieces, $\forall p\ x. \neg PAWNS\ x \supset Val(p, x) = Val(P0, x)$. That is, in any position, the value of any non-pawn piece is the same as its value was in the initial position. This is a theorem, not an axiom, and was proven within the axiomatization.

10. Board Relations

While we're talking about values, perhaps this is an appropriate time to mention *UD*, the undefined value. Essentially, we need a way of specifying the values of squares of boards when we don't know what is on that square. For example, on our problem board (*figure 1*), the X's square has an undefined value.

This creates a natural ordering on boards by "greater" definition. We call this ordering *SUBBOARD*. A board *b1* is a sub-board of a board *b2* (*SUBBOARD(b1, b2)*), if they are equal everywhere *b1* is defined.

We state that the *BOARD(p, b)* relation is true between a position and a board, when the board *b* is a *SUBOARD* of the total board of the position *p*. Hence, in reasoning about the given chess puzzle, we will first define the board *GIVEN*, the problem board, and then presume to be talking about the archetypical position, *px*, which has the *BOARD* relation with *GIVEN*.

11. Chess Movement

Of course, all of the entities we have defined so far have been syntactic ones. One of our major demonstrations is the use of a semantic model combined with these syntactic entities. We will wish to compute actual moves; for that reason, we will need to define moves on concrete chessboards, rather than the conceptual chess positions. Thus, we have the predicate *MOVETO(b, v, sq1, sq2)*, which states that a piece of value *v*, could move on board *b*, from square *sq1* to square *sq2*.

There is an axiom that breaks *MOVETO* down into the various different value movements, each with its own predicate. There is also an attachment in the chess eye that can take a constant board, value and squares, and evaluate the truth of this predicate on those arguments, and attachments to each of the submovement predicates. Both are useful, for the computation is simple and quick, and the axiom can be applied to non-constant arguments.

Even though the actual computation of moves is often done on explicit boards, we implicitly have associated with any move various functions. For example, we can speak of the square the move originated on, the *From* square of the move, or the piece that made any particular move the *Mover*.

Please recall that these are just a sample of the notions of the chess world. This is hardly the complete set of predicates and functions. The reader is referenced to [Filman 79] for a more complete listing.

12. Axioms

Our discussion of the predicates and functions of the chess world is necessarily very incomplete. So too must our examples of axioms from this system. The reader can, of course, consult [Filman 79] for the complete axiomatization.

The most important axioms in this system are those that delimit the transition between successor states; that is, those that define the legal moves. For example, consider the following representative axiom, MCONSEQA (*move consequences A*):

```
axiom MCONSEQA:   ∀r q.(SUCCESSOR(r, q)⊃
   ((¬WHITETURN r ≡ WHITETURN q) ∧
   Prevpos q = r ∧
   ¬POSITIONINCHECK(q, Color r) ∧
   (WHITEPIECE Mover Move q ≡ WHITETURN r)∧
   Pos(r, From Move q) = Mover Move q ∧
   Pos(q, To Move q) = Mover Move q ∧
   Pos(q, From Move q) = EMPTY ∧
   (CAPTURE Move q ⊃
      Pos(r, To Move q) = Taken Move q) ∧
   (CASTLING(r, q) ∨ EN_PASSANT(r, q) ∨
      SIMPLELEGALMOVE(r, q))));;
```

It states some of the conditions on moves; what piece must occupy which square, whether a move can leave the moving side in check, which pieces can be moved, and the beginnings of a taxonomy of movement.

It is also important to talk both about what is never true after a move transition, and what remains true between moves. For example, if a piece x is captured on the move that reach position r, then x is not on any square sq in r:

```
axiom MCONSEQF:
   ∀r sq x.(Taken Move r=x⊃¬Pos(r, sq)=x);;
```

13. The Chess Model

In our system, observation is performed by making the appropriate attachments to the FOL constants, and calling the semantic simplification routine. For example, chess boards are represented as eight long lists of eight atomic values.

The axioms have associated LISP functions which compute notions ranging from the simple "*the value on that square on this board*" on through "*can a piece of value v move, on b, from sq1 to sq2*" all the way to "*is black in check on board b*." The LISP functions themselves are not particularly noteworthy; they are usually the obvious transformation of the axioms onto the data structures. Thus, a single observational step in our proof produces the observation that black is in check on the board GIVEN (the problem board).

```
*****simplify BLACKINCHECK (GIVEN);
1 BLACKINCHECK (GIVEN)
```

Deducing the corresponding fact would require a very long proof. As a point of comparison, the theorem ALLSTART: $\forall x.\ \exists sq.\ Pos(P0,\ sq)=x$ which states that *every chesspiece is on some square in the initial position* required 165 proof steps to infer deductively, but only a single simplification using the chess eye in the semantic model.

14. Chess Induction

One other prominent feature of the chess axiomatization deserves mention, that of *Chess Induction*. Chess induction is an axiom that states that if a property P is true in a position p, and it remains true over the successor relation, then it will be true in all descendants of p (all games that can be reached by playing from p). We usually use the initial position, $P0$, for p, and prove properties true of all legal positions.

Chess induction, combined with the historical state vector (our position) is a powerful deductive technique. For example, using chess induction, we prove theorems such as: *A piece not on its original square has moved in this game (or been the moving rook of a castle)*:

```
∀ r sq x. ((Pos(r, sq)=x∧ ¬(Pos(P0, sq)=x))⊃
   ∃ q.((PREDEGAME(q, r)∨ q=r)∧
         ((Mover Move q=x∧To Move q=sq)∨
      (CASTLE Move q∧Alsomover Move q=x∧
         Alsoto Move q=sq))))
```

And that bishops stay on a single color square (here *ybi* is a variable ranging over bishops):

```
∀r sq1 sq2 ybi.((Pos(P0, sq1)=ybi∧Pos(r, sq2)=ybi)⊃
   (WHITESQUARES(sq1)≡WHITESQUARES(sq2)))
```

Note that here we have a powerful technique for dealing with the frame problem. If we have control over the types of interactions available in state transitions, induction schemas such as this one allow us to prove constant properties of distant states.

15. The Proof

This has been just a small sprinkling of examples from the chess axiomatization. We hope it has been useful in conveying their flavor and form.

From this axiomatization, we preceded to generate a proof of the solution to the chess puzzle. This is essentially the path that a general reasoning program seeking to solve the chess puzzle might follow.

The details of that proof are too gory to go into, but we can present some of the highlights and statistics. The form of the proof matched, fairly closely, an English explanation of the detailed solution to the same problem. The human proof required 68 "steps"; the FOL "proof", 405. In the process of deriving the FOL proof, 159 general chess lemmas and theorems were proven. These required 1702 steps. Additionally, six lemmas specific to this problem were also demonstrated, requiring another 136 steps. About half of the proof steps in the main proof were instantiations of axioms and theorems, and another quarter were requests for the confirmation of the chess eye.

It is clear to us that this proof of the solution to the problem from basic chess principles is well beyond the ability of any current theorem prover.

16. Conclusions

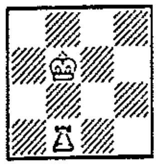
We believe that there are lessons to be learned here in the design of epistemological structures for artificial intelligence, and we will attempt to convey some of these precepts. This is, of course, a brief description; a more thorough examination may be found in the last chapter of [Filman 79].

The proof paralleled closely the reasoning inherent in the human solution of this problem, and, in that respect, FOL was acting as a good candidate Missouri program. As with any advice taker, be it machine or human, there were of course times when it would not accept "obvious" conclusions. This is in part a function of both the minor failings in the proof checker, and faults of the axiomatization.

The axiomatization adequately modeled the human proof, despite the disparity of size, with two major exceptions. Our notion of observation, tied as it was to computation on well defined objects, was inadequate for handling incompletely defined objects. These "objects" may be thought of as "objects with variables in them", or, perhaps, "partially defined objects". The board and sub-board dichotomy is an example of such partial objects. The creation of the value UD (undefined) (and the concomitant creation of partial boards) served as a partial remedy in this particular case. However, the general problem of reasoning about composite objects, some of whose primitives are unknown, was not uniformly approached. Introducing an "undefined" primitive value "solved" this problem for those predicates using values; this solution required modifying all chess eye functions that employed values. It did not deal with stating in the model restrictions and properties of these undefined objects. A similar scheme could perhaps have been employed for the other objects of the chess world (pieces, boards, positions). Of course, such a modification requires modifying all axioms and functions that employ these primitives to handle them. Even such an approach has certain inadequacies. A more general solution might involved a system that can talk about its own possible worlds, models, and proofs as objects in another, meta world. Nevertheless, it seems clear that a general reasoning system that wishes to employ such procedural attachment needs to deal with incomplete composite objects.

The second disparity involved mathematical notions. One thing that was avoided in this axiomatization was characterizing the elementary notions of mathematics. Arithmetic was easily avoided by reliance upon computation in the LISP model. However, this failure to allow a general concept of set or list within the FOL logical structure turned human arguments of the *pigeon-hole principle* form into case arguments in FOL. Thus, counting arguments became case checking problems. While not a completely satisfactory solution, avoiding set theory proved to be a workable decision for a problem of this size.

The particular semantic model selected for representing the chess world in LISP was adequate for dealing with "ordinary" retrograde chess puzzles. One can, however, stretch the domain demanded of any system to its limits. Certain things, such as the dimensions of the chess board, the notion of legal game, and the board's orientation relative to black and white were fixed in the internal representation. It is of course possible to generate chess puzzles on these themes. If this seems somewhat obscure, perhaps a couple of examples will clarify the situation. Consider this simple observation:

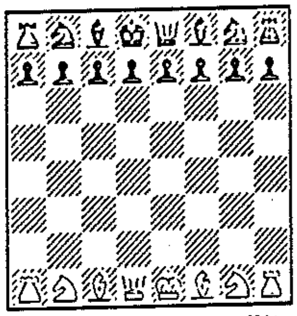


Is white in check on this fragment?

figure 2

The semantic model, as currently formulated, would not recognize this "fragment" as a piece of a "board". However, this is a natural extension and simple observation for the human problem solver. The axiomatization can be criticized for adhering too closely to the rules of chess, and ignoring its primitives. Although this sounds like a drastic error, correcting the axiomatization to reflect this more primitive level would not be that drastic a revision.

Similarly, consider the following problem, from [Gardner 59]:



White to play and mate in four. This is a legal position.

figure 9

This is not so much an issue of mating in four, so much as recognizing that (as the black king and queen are on the wrong color squares) the pieces have exchanged sides. While the current axiomatization could be used to prove that *if white started at the bottom of this board, then this board could not have been reached in a legal game*, in some sense, the puzzle needs to be solved before it can be put in the acceptable form for the axioms. What we have here is an issue of language. There are certainly many questions one would like to ask in, for example, English, for which natural language is inadequate. Imagine, if you will, the circumlocutions involved in giving a written spelling test.

If we were redo this axiomatization, we would recognize the need for both a flexible semantic model (a board being a list of squares and values, or perhaps even squares and limitations on values).

We also perceive the necessity for dealing, in a more uniform manner, with objects that are congregations of other, more primitive terms. Thus, in our proof, a position is a list of moves. This notion of historical position, retaining (and thus being able to comment upon) all that had happened to reach it, proved a very successful mechanism. The notion of *set of pieces captured* would have been the appropriate way to axiomatize the set theoretic arguments. Similarly, other natural chess notions, such as *path*, *block* and *strategy* can be seen as extensions of this idea. We have had some success expressing *strategies* as *accomplishable* predicates on positions.

Speaking of positions, they rank as a major success of this formalism. The ability to talk about what must be true of all objects of a certain kind, by examining them, and to prove, inductively, their properties, proved to be a very powerful notion. While the discrete nature of chess' time sequence eliminated some of the difficulties involved in transferring this arrangement to the "real" world, we do feel that this idea is touching on a powerful and often useful notion. More generally, practice with this axiomatization and others leads us to the belief that there are essentially two different kinds of situational variables, those expressing the *statics* of a particular reality (the chess example being *boards*), what is where, and others expressing *historical development*, such as our *positions*. The former are most useful for forward analysis, answering questions of the form, *Where can we go from here?*; the latter, for dealing with *How did we get here?*. Generally, the static elements forms the *slices* of the historical situations. As we have stated, inductive schema on historical state vectors seems to be a promising approach to the frame problem.

We feel that this proof illuminates the essential dichotomy of reasoning paradigms employed by A.I. systems; the rule base syntactic forms, and the "block box" semantic computations.

To summarize, evaluation in the semantic model seems an appropriate method for incorporating a form of observational behavior into a formal inference scheme. Care must be taken, however, to select a system flexible enough to handle future eventualities, and to employ mechanisms which will support complicated reasoning. Incorporation of the procedural functions as a semantic model will retain ability to talk freely about these functions on the logical language level, and to manipulate them with all of the previously obtained mathematical results. More powerful systems will be obtained when these semantics models, together with the descriptive language can be regarded as themselves objects for logical manipulation.

17. Bibliography

Chomsky 65 Chomsky, Noam; Some Aspects of the Theory of Syntax. MIT Press, Cambridge Massachusetts 1965.

Filman 76 Filman, Robert E., and R.W.Weyhrauch; An FOL Primer. Stanford A.I. Memo 288, 1976.

Filman 79 Filman, Robert E.; The Interaction of Observation and Inference. (*dissertation, in preparation.*)

Gardner 59 Gardner, Martin; *Mathematical Games* in the Scientific American May, 1959.

Hayes 77 Hayes, P.J.; In Defense of Logic. In the proceeding of the *5th International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts 1977.

McCarthy 58 McCarthy, John; *The Advice Taker*. Reprinted in Minsky, ed. Semantic Information Processing. MIT Press, Cambridge, Massachusetts 1968.

McCarthy 69 McCarthy, John, and P.J. Hayes; *Some Philosophical Problems from the Standpoint of Artificial Intelligence*. in Machine Intelligence 4, (eds. B. Meltzer and D. Michie) Edinburgh: Edinburgh University Press 1969.

McCarthy 77 McCarthy, John; Epistemological Problems of Artificial Intelligence. In the proceeding of the 5th International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts 1977.

McCarthy 79 McCarthy, John; *Concept valued functions*. To appear in Machine Intelligence 9.

Moore 79 Moore, Robert C.; Reasoning about Knowledge and Action. (*dissertation, forthcoming.*)

Pratt 77 Pratt, Vaughan R.; The Competence/Performance Dichotomy in Programming. MIT A.I. Memo 400, 1977.

Weyhrauch 77 Weyhrauch, Richard W.; A users manual for FOL. Stanford A.I. Memo 235.1, 1977.

Weyhrauch 79 Weyhrauch, Richard W.; Prolegomena to a theory of formal reasoning. (*Forthcoming.*)

Complexity of Combinations of Quantifier-Free Theories

Derek C. Oppen
Artificial Intelligence Laboratory
Computer Science Department
Stanford University

Preliminary Version

Abstract

We restrict our attention to decidable quantifier-free theories, such as the quantifier-free theory of integers under addition, the quantifier-free theory of arrays under storing and selecting, or the quantifier-free theory of list structure under ***cons***, ***car*** and ***cdr***. We describe a simple nondeterministic procedure which combines decision procedures for theories such as these into a decision procedure for their combination. We analyze the running time of this nondeterministic decision procedure and use it to show, for example, that the satisfiability problem for the quantifier-free theory of integers, arrays, list structure and uninterpreted function symbols under +, ≤, ***store***, ***select***, ***cons***, ***car*** and ***cdr*** is NP-complete. We discuss the complexity of the satisfiability problem for formulas already in disjunctive normal form (why some combinations of theories admit deterministic polynomial time decision procedures while for others the problem is NP-hard) and the essential role that case analysis plays in deciding combinations of theories.

1. Introduction

In many applications of theorem proving, particularly those involving program verification, program manipulation and program optimization, we would like to be able to very quickly decide formulas or simplify expressions involving the common data structures of programming languages: numbers, arrays, records, list structure, sets, multisets.

The first-order theories of these data structures are either undecidable or of very high complexity. For this reason, most "practical" theorem provers restrict their attention to quantifier-free formulas over these data structures. Empirically this restriction seems reasonable: it admits a large and useful class of formulas, yet theorem provers which handle this class generally do so reasonably efficiently.

This research was supported by the National Research Foundation under contract MCS 78-02835.

The purpose of this paper is to explore the complexity of reasoning in quantifier-free theories. We are particularly interested in combinations of quantifier-free theories, such as the theory which "combines" the quantifier-free theories of integers under addition, arrays under storing and selecting, and list structure under ***car***, ***cdr***, ***cons***. The reason is that the formulas which arise in practice tend to be "mixed" formulas containing symbols from various theories (formulas such as $A[I + 1] < A[I]$) rather than from just one of the base theories. Thus, we are interested in, for instance, the quantifier-free theory of integers, arrays, list structure and uninterpreted function symbols under +, ≤, ***store***, ***select***, ***cons***, ***car*** and ***cdr***. (A decision procedure for this theory is implemented as part of the simplifier in the Stanford Pascal Verifier.)

Implementers of theorem provers for a theory such as this one have generally shied away from implementing an actual decision procedure, and have instead relied on ad hoc techniques designed to catch "most" cases. There are at least two reasons for this. One is that, until recently, there has been little research done on what a decision procedure looks like for "mixed" theories such as these (or even if one can exist). The main reason, however, is the common belief that any decision procedure for such a theory must be slow and impractical, that the complexity of such an apparently rich theory must be very high. However, as we shall show, the satisfiability problem for the above theory is in fact only NP-complete.

In section 2, we describe a nondeterministic procedure for deciding combinations of quantifier-free theories which generalizes the deterministic procedures given in [Nelson and Oppen 1978], [Shostak 1977] and Suzuki and Jefferson [1977]. We analyze the running time of this procedure. In section 3, we review some existing results on decidability and complexity of various theories. In section 4, we use the results of the previous sections to analyze the complexity of several combinations of theories. In section 5, we consider quantifier-free DNF combinations of theories, that is, combinations of theories all of whose formulas are in disjunctive normal form. Some of these theories have polynomial time solutions while others are NP-hard. In section 6, we discuss the cost of the case-analysis inherent in any (deterministic) implementation of the procedure described in section 2.

2. Nondeterministic Combinations of Theories

Assume we have several quantifier-free theories with no non-logical symbols in common and that for each we have a *satisfiability program* which determines the satisfiability of a conjunction of literals in the theory. Our goal is to construct a (nondeterministic) satisfiability program for the quantifier-free theory whose set of non-logical symbols is the union of the sets of non-logical symbols of the individual theories and whose set of axioms is the union of the sets of axioms of the individual theories. We will assume that we have just two theories; the generalization to more than two is straightforward. The following definitions and three lemmas are taken from [Nelson and Oppen 1978b].

If $\mathcal{S}$ is a theory, then a term is an *$\mathcal{S}$-term* if each non-logical symbol occurring in the term is a non-logical symbol of $\mathcal{S}$. We define *$\mathcal{S}$-literal* and *$\mathcal{S}$-formula* similarly. If $\mathcal{S}$ is a theory, a *satisfiability program* for $\mathcal{S}$ is a program which determines whether a conjunction $L_1 \wedge \ldots \wedge L_k$ of $\mathcal{S}$-literals is satisfiable in $\mathcal{S}$. We will use the name of a theory to denote also its satisfiability program and the conjunction of its axioms.

A *parameter* of a formula is any non-logical atomic symbol which occurs free in the formula. Thus the parameters of $a = b \vee \forall x\ P(x, f(x)) = c$ are a, b, P, f, and c.

A *simple* formula is one whose only parameters are individual variables. For instance, $x \neq y \vee z = y$ and $\forall x\ x \neq y$ are simple, but $x < y$ and $f(x) = y$ are not. Thus an unquantified simple formula is a propositional combination of equalities between individual variables.

Lemma 1: If F is any formula, then there exists an unquantified simple formula Res(F), the *residue* of F, which is the strongest simple formula that F entails; that is, if H is any simple formula entailed by F, then Res(F) entails H. Res(F) can be written so that its only variables are free variables of F.

Here are some examples of residues.

Formula	Residue
$x = f(a) \wedge y = f(b)$	$a = b \supset x = y$
$x = \textit{store}(v, i, e)[j]$	$i = j \supset x = e$
$x = \textit{store}(v, i, e)[j] \wedge y = v[j]$	***if*** $i = j$ ***then*** $x = e$ ***else*** $x = y$
$x \neq x$	***false***

Lemma 2: If A and B are formulas whose only common parameters are individual variables, then $\mathrm{Res}(A \wedge B) = \mathrm{Res}(A) \wedge \mathrm{Res}(B)$.

A formula F is *non-convex* if there exist variables $x_1, y_1, \ldots, x_n, y_n$, $n \geq 2$, such that $F \supset x_1 = y_1 \vee \ldots \vee x_n = y_n$ but for no i between 1 and n does $F \supset x_i = y_i$. Otherwise, F is *convex*. That is, a formula is non-convex if it entails a disjunction of equalities between variables without entailing any of the equalities alone, otherwise it is convex. For instance, the formula $1 \leq x \leq 2 \wedge y = 1 \wedge z = 2$ is non-convex over the integers because it entails the disjunction $x = y \vee x = z$ without entailing either equality alone.

Lemma 3: Let $F_1, F_2, \ldots, F_n$ be simple, convex formulas and V be the set of all variables appearing in any F_i. Suppose that for all x, y in V and for all i, j from 1 to n, either both F_i and F_j entail $x = y$, or neither do. Then $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ is satisfiable if and only if each F_i is satisfiable.

2.1 The Nondeterministic Satisfiability Procedure

We assume we have two theories $\mathcal{S}$ and $\mathcal{T}$ which have no common non-logical symbols, that we have satisfiability programs for $\mathcal{S}$ and $\mathcal{T}$, and that $\mathcal{S}$ and $\mathcal{T}$ are the axioms for $\mathcal{S}$ and $\mathcal{T}$. We are given an unquantified formula F whose non-logical symbols are among those of $\mathcal{S}$ and $\mathcal{T}$, and wish to determine whether F is satisfiable in the theory $\mathcal{S} \cup \mathcal{T}$, that is, whether $\mathcal{S} \wedge \mathcal{T} \wedge F$ is satisfiable.

Consider first the disjunctive normal form of F. Each disjunct is a conjunction of literals; F is satisfiable if and only if one of these disjuncts is satisfiable. Our first step is therefore to guess which atomic formulas in F make up a satisfiable disjunct. Call the conjunction of these literals F'.

F' may contain "mixed" literals, literals which are neither $\mathcal{S}$-literals nor $\mathcal{T}$-literals. For instance, suppose that $\mathcal{S}$ is quantifier-free Presburger arithmetic, that $\mathcal{T}$ is the quantifier-free theory of list structure and that F' is the single literal $y = \textit{car}(x) + 3$. This literal is neither a $\mathcal{S}$-literal nor a $\mathcal{T}$-literal. We wish to divide F' into two formulas, one which can be handled by the satisfiability program for $\mathcal{S}$ and one which can be handled by the satisfiability program for $\mathcal{T}$. That is, we want to construct two formulas F_S and F_T so that F_S is a conjunction of $\mathcal{S}$-literals, F_T is a conjunction of $\mathcal{T}$-literals, and $F_S \wedge F_T$ is satisfiable if and only if F' is. In our example, $y = \textit{car}(x) + 3$ is equivalent to $z = \textit{car}(x) \wedge y = z + 3$ where z is a new (existential) variable; $z = \textit{car}(x)$ is a $\mathcal{T}$-literal and $y = z + 3$ is a $\mathcal{S}$-literal. We can therefore let F_S be $y = z + 3$ and F_T be $z = \textit{car}(x)$. In general, we construct F_S and F_T from arbitrary F' in similar fashion: for each literal appearing in F', if the literal is an $\mathcal{S}$-literal, we add it to F_S; if it is a $\mathcal{T}$-literal, we add it to F_T; otherwise we introduce new variables to replace terms of the wrong "type" and add equalities defining these variables.

We wish to determine if $\mathcal{S} \wedge F_S \wedge \mathcal{T} \wedge F_T$ is satisfiable. First, if $x_1, ..., x_k$ are all the variables in F_S and F_T, we guess the equalities and disequalities that hold among the x_i, and let E be a conjunction of equalities and disequalities of variables describing our guess. For instance, if there are four variables x_1, x_2, x_3 and x_4, E might be $x_1 = x_2 \wedge x_3 = x_4 \wedge x_1 \neq x_3$.

We add E to our conjunction and now wish to determine the satisfiability of $\mathcal{S} \wedge F_S \wedge \mathcal{T} \wedge F_T \wedge E$. It suffices to show that $\mathrm{Res}(\mathcal{S} \wedge F_S \wedge \mathcal{T} \wedge F_T \wedge E)$ is satisfiable (that is, not ***false***). By lemma 2, this residue is equivalent to $\mathrm{Res}(\mathcal{S} \wedge F_S \wedge E) \wedge \mathrm{Res}(\mathcal{T} \wedge F_T \wedge E)$. Since E already expresses precisely the equalities and disequalities that hold between the variables, $\mathrm{Res}(\mathcal{S} \wedge F_S \wedge E)$ and $\mathrm{Res}(\mathcal{T} \wedge F_T \wedge E)$ are simple and convex, and entail the same set of equalities among variables. Hence, by lemma 3, to verify that F' is satisfiable, it suffices to show that both $\mathrm{Res}(\mathcal{S} \wedge F_S \wedge E)$ and $\mathrm{Res}(\mathcal{T} \wedge F_T \wedge E)$ are satisfiable. This will be the case if and only if $F_S' = \mathcal{S} \wedge F_S \wedge E$ and $F_T' = \mathcal{T} \wedge F_T \wedge E$ are both satisfiable. Since F_S' is a conjunction of $\mathcal{S}$-literals and F_T' is a conjunction of $\mathcal{T}$-literals, we can use the satisfiability programs of $\mathcal{S}$ and $\mathcal{T}$ to determine their satisfiability.

The essential idea behind this nondeterministic procedure thus is to guess all the equalities that hold between the variables and then to use the individual satisfiability programs to decide whether the formula with these equalities is satisfiable.

2.2 Analysis of the Algorithm

What is the running time of this nondeterministic satisfiability procedure? Let n be the length of the incoming formula F. We can guess F' in nondeterministic polynomial time; the size of F' is linear in n. We can construct F_S and F_T in polynomial time. We can guess the equalities that hold between the variables of F' in nondeterministic polynomial time; the size of E is a polynomial in n. The sizes of F_S' and F_T' are again polynomial in n. The remaining time required is whatever time is required by the satisfiability programs for $\mathcal{S}$ and $\mathcal{T}$ to verify that F_S' and F_T' are satisfiable.

Consider now the satisfiability problem for $\mathcal{S} \cup \mathcal{T}$. It is certainly NP-hard (because of the arbitrary boolean structure allowed in formulas). The problem of constructing the formulas F_S' and F_T' is in NP. If the problems of determining the satisfiability of conjunctions of $\mathcal{S}$-literals and of $\mathcal{T}$-literals are also in NP, then the satisfiability problem for $\mathcal{S} \cup \mathcal{T}$ is in NP and hence NP-complete. Otherwise, the complexity of the satisfiability problem will be dominated by the complexity of the problem for $\mathcal{S}$ or for $\mathcal{T}$.

The results given above for two theories generalize in a straightforward fashion to more than two theories.

The following summarizes these results.

Theorem 1: Let $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_k$ be decidable, quantifier-free theories with no common non-logical symbols. Then $\mathcal{T}_1 \cup \mathcal{T}_2 \cup ... \cup \mathcal{T}_k$ is decidable; if the satisfiability problem for each of the $\mathcal{T}_i$ is in NP, then the satisfiability problem for $\mathcal{T}_1 \cup \mathcal{T}_2 \cup ... \cup \mathcal{T}_k$ is in NP and hence NP-complete.

3. Review of Existing Complexity Results

Before using the results of the previous section to analyze the complexity of various combinations of theories, we first summarize some existing results. In the following, the quantifier-free DNF theory is the theory in which every formula is already in disjunctive normal form. This restriction is of interest because the complexity of its satisfiability problem is just the complexity of determining the satisfiability of a conjunction of literals.

3.1 Theory of Integers under Addition

The first order theory was shown decidable by [Presburger 1929]. [Fischer and Rabin 1972] prove that the theory has a double-exponential lower bound on nondeterministic time. [Oppen 1978] proves that the the theory has a triple-exponential upper bound on deterministic time. [Reddy and Loveland 1978] prove that the bounded quantifier subtheory has a double-exponential upper bound. The satisfiability problems for the quantifier-free theory and the quantifier-free DNF theory are NP-complete; this follows from [Borosh and Treybig 1976].

3.2 Theory of Integers under Successor

This is the same as the above theory except that addition of variables is not allowed, only addition of a variable and a constant. (An example formula in this theory is $x \neq y \supset x + 1 \leq y \vee y + 1 \leq x$.) This theory is an interesting subtheory of Presburger arithmetic because [Pratt 1977] has shown that one can determine the satisfiability of a formula of length n in the quantifier-free DNF theory in time $O(n^3)$.

3.3 Theory of Equality with Function Symbols

A proof of the decidability of the quantifier-free theory appears in [Ackermann 1953]. (An example of a valid formula in this theory is $x = y \supset f(x,y) = f(y,x)$.) [Nelson and Oppen 1978a] give an $O(n^2)$ decision procedure for the DNF quantifier-free theory. It follows that the satisfiability problem for the quantifier-free theory is NP-complete.

3.4 Theory of List Structure under *car*, *cdr* and *cons*

There are several possible axiomatizations for this theory:

$$\begin{aligned}
&car(cons(x, y)) = x \\
&cdr(cons(x, y)) = y \qquad\qquad (1)\\
&listp(x) \supset cons(car(x), cdr(x)) = x \\
&listp(cons(x, y))
\end{aligned}$$

$$\begin{aligned}
&car(cons(X, Y)) = X \\
&cdr(cons(X, Y)) = Y \\
&X \neq nil \supset cons(car(X), cdr(X)) = X \qquad (2)\\
&cons(X, Y) \neq nil \\
&car(nil) = cdr(nil) = nil
\end{aligned}$$

$$\begin{aligned}
&car(cons(X, Y)) = X \\
&cdr(cons(X, Y)) = Y \\
&cons(car(X), cdr(X)) = X \qquad (3)\\
&car(X) \neq X \\
&cdr(X) \neq X \\
&car(car(X)) \neq X \\
&\quad \ldots
\end{aligned}$$

[Nelson and Oppen 1978a] show that the satisfiability problem for the quantifier-free DNF theory axiomatized by (1) has an $O(n^2)$ solution, but that the problem for (2) is NP-complete. [Oppen 1978a] gives a linear algorithm for (3). Therefore, for any of these axiom schemata, the quantifier-free theory is NP-complete. The first order theory was shown decidable but not elementary recursive by [Oppen 1978a].

These results generalize easily to data structures with one constructor c and k selector functions $s_1, \ldots, s_k$. Such data structures are called *recursively defined data structures*.

3.5 Theory of Arrays under Selecting and Storing

The axioms for this theory are as follows:

$$\begin{aligned}
&select(store(A, I, E), I) = E \\
&I \neq J \supset select(store(A, I, E), J) = A[J] \\
&store(A, I, select(A, I)) = A \\
&store(store(A, I, E), I, F) = store(A, I, F) \\
&I \neq J \supset store(store(A, I, E), J, F) \\
&\qquad\qquad = store(store(A, J, F), I, E)
\end{aligned}$$

select(A, I) is the Ith component of the one-dimensional array A. A[I] abbreviates ***select***(A, I). ***store***(A, I, E) is the array whose Ith component is E and whose Jth component, for J ≠ I, is A[J]. A two-dimensional array is considered a vector of vectors, so A[I,J] abbreviates A[I][J]. The last three axioms are only needed if equalities between array terms are allowed ([Kaplan 1968]).

[Downey and Sethi 1976] show that the satisfiability problem for the DNF quantifier-free theory is NP-complete.

3.6 Theory of Integers with Function Symbols

[Shostak 1978] shows that the quantifier-free theory of integers with uninterpreted functions under + and ≤ is decidable. (This also follows immediately from Theorem 1.)

3.7 Theory of Integers and Arrays

[Suzuki and Jefferson 1977] show that the quantifier-free theory of arrays and integers under +, ≤, ***store*** and ***select*** is decidable. (This also follows directly from Theorem 1.) They also extend their results to the quantifier-free theory all of whose formulas are of the form P ⊃ Q ∧ PERM(A,B) where P and Q are conjunctions of literals over the theory of arrays and integers under +, ≤, ***store*** and ***select*** and A and B are array terms. PERM(A,B) is interpreted to mean that array A is a permutation of array B.

4. Complexity of Various Combinations of Theories

The results quoted in the last section lead immediately to the following corollaries of Theorem 1.

Corollary 1: The satisfiability problem for the quantifier-free theory of integers, arrays, list structure and uninterpreted function symbols under +, ≤, ***store***, ***select***, ***cons***, ***car*** and ***cdr*** is NP-complete.

Corollary 2: The satisfiability problem for the quantifier-free theory of integers and arrays under +, ≤, ***store*** and ***select*** is NP-complete.

This is the theory considered by [Suzuki and Jefferson 1977]. It is easy to verify as well that the addition of the PERM predicate does not change the NP-completeness.

Corollary 3: The satisfiability problem for the quantifier-free theory of integers and uninterpreted function symbols under + and ≤ is NP-complete.

This is the theory considered by [Shostak 1978].

5. Convexity

Since the theories we considered in the previous section were already NP-hard (because of the arbitrary boolean structure allowed in formulas), our analysis of the running time of our nondeterministic procedure could be fairly gross: it sufficed to show that each step required at most nondeterministic polynomial

time. But what if we restrict our attention to formulas already in disjunctive normal form, that is, to quantifier-free DNF combinations of theories?

The satisfiability problem for some quantifier-free DNF theories (such as the theory of integers under addition or of arrays under storing and selecting) is already NP-hard, and any theory including such a theory must therefore be at least as hard.

However, if we further restrict our attention to quantifier-free DNF theories with deterministic polynomial time satisfiability problems, we might hope that their quantifier-free DNF combinations also admit deterministic polynomial time solutions. For instance, we might consider combinations of the quantifier-free DNF theories of integers under successor, equality with uninterpreted function symbols, and list structure under **car**, **cons** and **cdr** (with axioms (1) or (3)) since each has a deterministic polynomial satisfiability problem.

The results are mixed. For instance, [Nelson and Oppen 1978a] show that the satisfiability problem for the quantifier-free DNF theory of list structure with uninterpreted function symbols has an $O(n^2)$ solution. On the other hand, [Pratt 1977] shows that the theory of integers (under successor) with uninterpreted function symbols is NP-hard.

These results are closely related to the property of convexity. Recall that a formula is non-convex if it entails a disjunction of equalities between variables without entailing any of the equalities alone; otherwise it is convex. Define a theory $\mathcal{S}$ to be *convex* if every conjunction of $\mathcal{S}$-literals is convex; otherwise if is *non-convex*.

Some of the theories considered in this paper are convex, others non-convex. The theories of integers under addition and of integers under successor are non-convex. For instance, the formula $1 \le x \le 2 \wedge y = 1 \wedge z = 2$ entails the disjunction $x = y \vee x = z$ without entailing either equality alone. The theories of equality with uninterpreted function symbols and of list structure under **car**, **cdr** and **cons** are convex ([Nelson and Oppen 1978a]). The theory of arrays is non-convex. For instance, the formula $x = \mathbf{store}(a, i, e)[j] \wedge y = a[j]$ entails the disjunction $i = j \wedge x = e \vee i \neq j \wedge x = y$.

Consider again the complexity results given above for DNF quantifier-free combinations of theories. If at least one of the theories was non-convex, the DNF combination was NP-hard. The only combination of theories for which the DNF satisfiability problem admits a polynomial solution was a combination of two convex theories.

Suppose we have two convex theories $\mathcal{S}$ and $\mathcal{T}$, and that for each we have a deterministic polynomial time decision procedure for deciding satisfiability of conjunctions of literals. Then we can decide the satisfiability of a conjunction F in their union in polynomial time by the following procedure (see [Nelson and Oppen 1978b]).

1. Construct F_S and F_T from F as in the nondeterministic procedure in section 2.

2. If either F_S or F_T are unsatisfiable, then so is F.

3. If either F_S or F_T entail some equality between variables not entailed by the other, then add the equality as a new conjunct to the one that does not entail it and go to step 2.

4. If this step is reached, F is satisfiable.

Steps 2 and 3 can be executed at most n times, where n is the length of F, since there can be most n variables in F_S and F_T, and there can be at most n - 1 non-redundant equalities between n variables. Step 2 takes polynomial time. Step 3 also takes polynomial time: to determine if $x = y$ is entailed by F_S, say, we check whether $F_S \wedge x \neq y$ is unsatisfiable.

This procedure therefore runs in polynomial time, and leads to the following theorem (suggested by Chris Goad):

Theorem 2: Let $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$ be decidable, convex, quantifier-free theories with no common non-logical symbols and with deterministic polynomial time DNF satisfiability problems. Then $\mathcal{T}_1 \cup \mathcal{T}_2 \cup \ldots \cup \mathcal{T}_k$ has a deterministic polynomial time DNF satisfiability problem.

6. Case Splitting

If the theories being combined are non-convex, the nondeterministic procedure given in section 2 for combining satisfiability programs translates in the obvious fashion into a deterministic procedure. The incoming formula is converted into disjunctive normal form, each disjunct is massaged into one containing no literals of "mixed" type, a case split is done on all the ways that the variables in each conjunct can be equal, and the individual satisfiability programs are used to determine the satisfiability of each branch of the split.

This simplistic way of combining satisfiability programs is of course rather inefficient: a superior method is given in [Nelson and Oppen 1978b]. However, it is interesting to briefly analyze the running time of this brute force algorithm since it illustrates the importance of case splitting.

Assume that the size of the original formula is n. The size of each disjunct in disjunctive normal form is thus O(n); there may be 2^n disjuncts. Under a reasonable model of computation, time. But what if we restrict our attention to formulas already in disjunctive normal form, that is, to quantifier-free DNF combinations of theories?

The satisfiability problem for some quantifier-free DNF theories (such as the theory of integers under addition or of arrays under storing and selecting) is already NP-hard, and any theory including such a theory must therefore be at least as hard.

However, if we further restrict our attention to quantifier-free DNF theories with deterministic polynomial time satisfiability problems, we might hope that their quantifier-free DNF combinations also admit deterministic polynomial time solutions. For instance, we might consider combinations of the quantifier-free DNF theories of integers under successor, equality with uninterpreted function symbols, and list structure under **car**, **cons** and **cdr** (with axioms (1) or (3)) since each has a deterministic polynomial satisfiability problem.

The results are mixed. For instance, [Nelson and Oppen 1978a] show that the satisfiability problem for the quantifier-free DNF theory of list structure with uninterpreted function symbols has an $O(n^2)$ solution. On the other hand, [Pratt 1977] shows that the theory of integers (under successor) with uninterpreted function symbols is NP-hard.

These results are closely related to the property of convexity. Recall that a formula is non-convex if it entails a disjunction of equalities between variables without entailing any of the equalities alone; otherwise it is convex. Define a theory $\mathcal{S}$ to be *convex* if every conjunction of $\mathcal{S}$-literals is convex; otherwise if is *non-convex*.

Some of the theories considered in this paper are convex, others non-convex. The theories of integers under addition and of integers under successor are non-convex. For instance, the formula $1 \le x \le 2 \wedge y = 1 \wedge z = 2$ entails the disjunction $x = y \vee x = z$ without entailing either equality alone. The theories of equality with uninterpreted function symbols and of list structure under **car**, **cdr** and **cons** are convex ([Nelson and Oppen 1978a]). The theory of arrays is non-convex. For instance, the formula $x = \textit{store}(a, i, e)[j] \wedge y = a[j]$ entails the disjunction $i = j \wedge x = e \vee i \neq j \wedge x = y$.

Consider again the complexity results given above for DNF quantifier-free combinations of theories. If at least one of the theories was non-convex, the DNF combination was NP-hard. The only combination of theories for which the DNF satisfiability problem admits a polynomial solution was a combination of two convex theories.

Suppose we have two convex theories $\mathcal{S}$ and $\mathcal{T}$, and that for each we have a deterministic polynomial time decision procedure for deciding satisfiability of conjunctions of literals. Then we can decide the satisfiability of a conjunction F in their union in polynomial time by the following procedure (see [Nelson and Oppen 1978b]).

1. Construct F_S and F_T from F as in the nondeterministic procedure in section 2.

2. If either F_S or F_T are unsatisfiable, then so is F.

3. If either F_S or F_T entail some equality between variables not entailed by the other, then add the equality as a new conjunct to the one that does not entail it and go to step 2.

4. If this step is reached, F is satisfiable.

Steps 2 and 3 can be executed at most n times, where n is the length of F, since there can be most n variables in F_S and F_T, and there can be at most n - 1 non-redundant equalities between n variables. Step 2 takes polynomial time. Step 3 also takes polynomial time: to determine if $x = y$ is entailed by F_S, say, we check whether $F_S \wedge x \neq y$ is unsatisfiable.

This procedure therefore runs in polynomial time, and leads to the following theorem (suggested by Chris Goad):

Theorem 2: Let $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_k$ be decidable, convex, quantifier-free theories with no common non-logical symbols and with deterministic polynomial time DNF satisfiability problems. Then $\mathcal{T}_1 \cup \mathcal{T}_2 \cup \dots \cup \mathcal{T}_k$ has a deterministic polynomial time DNF satisfiability problem.

6. Case Splitting

If the theories being combined are non-convex, the nondeterministic procedure given in section 2 for combining satisfiability programs translates in the obvious fashion into a deterministic procedure. The incoming formula is converted into disjunctive normal form, each disjunct is massaged into one containing no literals of "mixed" type, a case split is done on all the ways that the variables in each conjunct can be equal, and the individual satisfiability programs are used to determine the satisfiability of each branch of the split.

This simplistic way of combining satisfiability programs is of course rather inefficient: a superior method is given in [Nelson and Oppen 1978b]. However, it is interesting to briefly analyze the running time of this brute force algorithm since it illustrates the importance of case splitting.

Assume that the size of the original formula is n. The size of each disjunct in disjunctive normal form is thus O(n); there may be 2^n disjuncts. Under a reasonable model of computation,

High Level Proof in LCF

Avra Cohn
Computer Science Department
Edinburgh University
July 1978

Abstract

We discuss a methodology, using the Edinburgh LCF system, for generating large formal proofs in a natural and structured way. Techniques include the use of (i) user-programmed strategies which encapsulate common patterns of inference, (ii) operations for building complex strategies out of simple ones, and (iii) the ability to set up and work within hierarchies of axiomatic theories. These are illustrated by the proof of correctness of a simple compiler.

Introduction

Because programs are complex objects, the proofs of their properties are typically long, tedious to carry out, and difficult to structure. Systems for performing or producing formal proofs rely on the assumption that a certain portion of most program proofs consists in steps which are in some sense routine; such systems become attractive when they can claim to handle bodies of routine detail automatically. The portion of proof thought to be routine varies from just a little, in the case of proof-checking systems, to rather a lot, in automatic theorem-provers.

Given that the user of a mechanical proof system must make some contribution to the proof process (if only to state the goal), one can classify and assess these systems according to the nature of the user-system interaction required to generate proofs. Among the questions one might ask about these systems are: How does the user drive the proof process? How natural is the interaction? How clearly does the user's sequence of contributions parallel his or her intuitive 'plan' for carrying out the proof? At what level of abstraction (from the actual inference steps) does the interaction occur? What is the nature of the finished product? -- in general, How is the proof attempt organised and structured?

In this paper we address these questions in the context of a particular proof-generation system (Edinburgh LCF, Logic for Computable Functions, [5]), and a particular proof performed within that system (a proof of correctness of a simple compiler). The study is intended as a step in developing a methodology for generating large formal proofs in a natural and structured way. The aims in generating the proof (and of LCF in general) have been to investigate the ways in which informal strategies for conducting proofs can be expressed, manipulated and generalised, and the extent to which their production can be made automatic.

Edinburgh LCF, implemented in 1975 [5], was based on Stanford LCF [10]. It provides a framework which can accomodate many styles of proof, from proof-checking to automatic theorem proving, at the discretion of the user. Features which make the system useful and flexible are

1. the availability of a logic (PPLAMBDA, for Polymorphic Predicate Lambda Calculus) in which the properties of programs and their semantics can be directly and easily stated;
2. the availability of a general- purpose programming language (ML, for Meta-Language), in which the user can write programs to manipulate PPLAMBDA objects, in particular, programs to encapsulate patterns of inference or strategies for doing proofs;
3. facilities for setting up and working within hierarchies of axiomatic theories which extend the basic logic;
4. a built-in body of basic strategies (and operations for combining them into complex ones) and
5. facilities for doing routine and user-specified simplifications (re-writes) automatically and efficiently

ML is a higher-order programming language, with a type discipline that allows user-defined (possibly polymorphic) types. PPLAMBDA is a logic based on terms from the typed lambda calculus, atomic formulae which are equivalences or inequivalences between terms, and compound formulae built up as in the predicate calculus. In ML, beside the usual types (integer, boolean, etc.), types are defined to correspond to PPLAMBDA constructs (term, formula, theorem, etc.) and functions on PPLAMBDA constructs (rules of inference which produce theorems as results).

In order to approach the questions suggested above, we have chosen to study the generation in LCF, of the proof of correctness of a compiler for a simple Algol-like language [4]. The compilation is considered in stages, each stage concerning itself with the removal of one high level construct in favour of a low-level one. It is hoped that the factoring of the compilation into stages will make the proof easier and more modifiable. One stage, for example, consists in removal of block-structuring in favour of stack-handling operations. The stage that we describe here deals with the removal of two constructs, while-loops and conditionals. These are replaced by combinations of if-not-jump statements and go-to statements, in a language whose programs are sequences of labeled statements.

The compiler described for this stage is based closely on a formulation by B. Russell [14]. In order to deal with an independently-motivated problem, we have essayed to stay as close to his statement of the problem as possible.

The compiler and its proof are formalised using denotational semantics. Both the source (high level) language and the target (low level) language are given standard semantics, that is, semantic functions mapping programs to functions which are their meanings. The compiler is specified as a function mapping source-language into target-language programs. The statement of correctness is that the meaning of a source program is equivalent to the meaning of its image program under compilation. The following diagram illustrates these relations:

$$\begin{tikzcd}
\boxed{\text{source language programs}} \arrow[r, bend left, "\text{compiler}"] \arrow[d, "\text{source language semantic function}"'] & \boxed{\text{target language programs}} \arrow[d, "\text{target language semantic function}"] \\
\boxed{\text{source language semantics}} \arrow[r, dash, bend right, "\text{equivalence}"'] & \boxed{\text{target language semantics}}
\end{tikzcd}$$

The informal proof given by Russell is in fact incorrect (see Section 3); that errors can (and are likely to) occur in proofs of a certain complexity is evidence of the need for formal, machine-checked or -generated proofs. We have supplied a correct proof .

In what follows, we will describe the problem and the informal proof, sketch the formalisation in LCF, and describe the way in which the strategies for generating the informal proof are translated into ways of generating the proof in LCF.

2. The Problem

To summarise, we present the source and target languages given by Russell, in BNF notation. We let hp, hp1 and hp2 (and sometimes p, p1 and p2) range over a domain HPROGRAM of high level programs, x over a set ID of identifiers, and ex over a set EXP of expressions.

```
              Source Language

hp ::= assign (x, ex) |
       if (ex, hp1, hp2) |
       while (ex, hp1) |
       compound (hp1, hp2)
```

For the target language syntax, x and ex are as above, while lp and lp1 range over a domain LPROGRAM of low-level programs, ls and ls1 over a domain LABELEDSTATEMENT of labeled statements, t over a domain STATEMENT of statements, and L over a domain LABEL of labels.

```
              Target Language

lp ::= <ls1, ... lsn, L>
ls ::= L: t
t  ::= assign (x, ex) |
       ifnot (ex, L) |
       goto (L) |
       program (lp1)
L  ::= L1 | L2 | L3 | L4 | L5
```

The compiling algorithm is defined for the various constructs. We use an informal notation in which, for example, "if t = assign (x, ex) then ..." tests whether t is an assignment, and if so, evaluates ... with x and ex bound to the correct components of the assignment statement. This is easily formalisable in lambda-notation. (We omit "program" in the program-statements below, writing "compiler p1" rather that "program (compiler p1)").

```
                    Compiler

compiler p =  if p = assign (x, ex)
              then     L1: assign (x, ex)
                       L2
              else if p = if (ex, p1, p2)
              then     L1: ifnot (ex, L4)
                       L2: compiler p1
                       L3: goto (L5)
                       L4: compiler p2
                       L5
               else if p = while (ex, p1)
               then    L1: ifnot (ex, L4)
                       L2: compiler p1
                       L3: goto (L1)
                       L4
               else if p = compound (p1, p2)
               then    L1: compiler p1
                       L2: compiler p2
                       L3
```

(The extra terminating labels are present for technical reasons only.) Note that the structure of expressions is not considered in further depth, so the problem of expression compilation is put aside; and that assignments are not compiled, but simply passed along. An assignment can be considered as either a source language program or a target language statement. We also note the presence of the program-statement in the target language; a whole target language program can be considered as a statement in a surrounding target language program. This block structuring in the target language has the effect of segregating the inner labels from the outer ones, so that only five labels are ever needed, and the function of each label, in the various cases, is fixed. (The problem of the generation of distinct labels is thus factored out and is considered at a later stage of the compilation. This very useful factoring is due to Russell.) For example, the label L2 always labels the program-statement which is the code for the body of a while-loop, in the while case.

We give the semantics of the two languages; they are again based closely on those given by Russell. However, for doing the inductions required in the proof, it will be necessary to be more precise about defining the semantic functions as least fixed points of certain functionals (as they are recursively defined, and in the low-level semantics case, mutually recursively).

For the source language semantics, we will need the domains ID and EXP, as mentioned, and another, VALUES, for basic values, including truth values.

We need a domain STORE and associated function eval to evaluate expressions in stores:

```
STORE = ID -> VALUES
eval: EXP x STORE -> VALUES
```

and a semantic function which we call hsem, for high level semantic function, of type:

```
hsem: HPROGRAM -> STORE -> STORE
```

hsem is the least fixed point of the functional hsemfun:

```
hsemfun = λhsem. λhp. λs.
    if hp = assign (x, ex)
    then          s [eval (ex, s) / x]
    else if hp = if (ex, hp1, hp2)
    then          if eval (ex, s)
                  then hsem hp1 s
                  else hsem hp2 s
    else if hp = while (ex, hp1)
    then          if eval (ex, s)
                  then (hsem hp) (hsem hp1 s)
                  else s
    else if hp = compound (hp1, hp2)
    then          (hsem hp2) (hsem hp1 s)
```

This is a standard direct semantics, and requires little comment.

For the low-level semantics, we need the following semantic domains:

```
CONTINUATION = STORE -> STORE
LABELENV = LABEL -> CONTINUATION
```

and a semantic function lsem (for low-level), mapping target language programs to <u>label</u> <u>environments</u>.

```
lsem: LPROGRAM -> LABELENV
```

Label environments map labels to continuations; each label in a target program is associated with the continuation (store to store function) which represents the meaning of the program from that label to the end of the program. For this, we need another semantic function (lsemstatement), of type:

```
lsemstatement: STATEMENT -> LABELENV -> CONTINUATION ->
                   (LPROGRAM -> LABELENV) -> CONTINUATION
```

lsemstatement is defined, for the various constructs:

```
lsemstatement t e c lsem =
     if t = assign (x, ex)
     then       λs. c ( s [eval (ex, s) / x])
     else if t = ifnot (ex, L)
     then       λs. if eval (ex, s)
                    then c s
                    else e L s
     else if t = goto (L)
     then       e L
     else if t = program (lp)
     then       λs. c (lsem lp L1 s)
```

This is straightforward in all but the program-statement case; in that case, note that the label environment is disregarded, and the continuation provided is applied to the meaning (found by applying lsem, the meaning function for whole target programs) of the program which constitutes the program-statement, thus isolating the inner labels as desired.

The function lsem, giving meanings to whole low level programs, is to be the least fixed point of the following functional ("compiler" is abbreviated to "comp" here and elsewhere). $\perp_{LE}$ is the undefined element in the domain LABELENV, and e {c1 / L1} . . . {cn / Ln} denotes the label environment e with the continuations ci bound to the labels Li.

```
     lsem = FIX lsemfun

     lsemfun = λlsem. λlp.
   if lp =  L1: t1
            L2: t2
             .
             .
             .
            Ln: tn
            Ln+1
   then
⊥LE{lsemstatement t1 (lsem lp)(lsem lp L2) lsem / L1}
  {lsemstatement t2 (lsem lp)(lsem lp L3) lsem / L3}
          .
          .
          .
  {lsemstatement tn (lsem lp)(lsem lp Ln+1) lsem /Ln}
  {(λs.s) / Ln+1}
```

As lsem and lsemstatement can be seen to be mutually recursive, we use the standard device of passing along a functional argument, which explains why lsemstatement needs its last argument. Finally, we state the formula which represents the correctness of the compiler:

$$\forall p.\ \text{hsem } p \;=\; \text{lsem (compiler } p)\ \text{L1}$$

That is, the meaning of any source language program is equivalent to the meaning of the compiled version of it (a label environment in which its labels are attached to continuations), applied to L1, the first label of the program. Intuitively, this will be the meaning of the first statement of the compiled program, in the label environment of the <u>whole</u> compiled program , with the continuation for the <u>rest</u> of the program. Note the recursiveness here.

This is the formula we go on to prove, first informally, and then in LCF.

3. The Informal Proof

As was mentioned earlier, Russell's proof is in fact incorrect. It is unlikely that the formula can be proved as an equivalence (see Milne [9] .) Russell tries to prove it by doing a simultaneous computation induction on the functions hsem and compiler. The proof we have found divides the formula into a pair of inequivalences:

```
      ∀ p. hsem p ⊑ lsem (compiler p) L1

      ∀p. hsem p ⊒ lsem (compiler p) L1
```

The first is proved by computation induction on the function hsem. The induction rule to which we appeal can be stated, for a formula w[f], and a definition |- f = FIX fun, as:

```
basis:           w [⊥ ]
step:    ∀ f'. w[f' ]  ⊃  w[fun f' ]
conclusion:      w [ FIX fun ]
```

Thus, the basis and step are, respectively:

```
 ∀ p.⊥ p  ⊑   lsem (compiler p) L1

 ∀ p.  hsem' p  ⊑   lsem (compiler p) L1        ⊃
       hsemfun hsem' p   ⊑  lsem (compiler p) L1
```

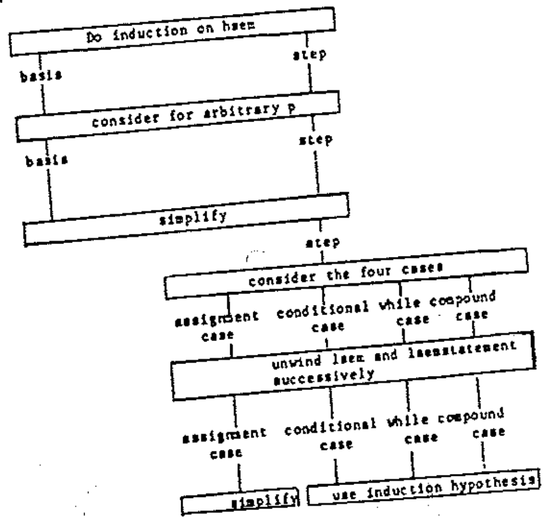
that is, for the step.

```
 ∀ p. hsem' p  ⊑   lsem (compiler p) L1    ⊃
      (λs. if p = assign (x, ex)
           then s [eval (ex, s) / x)
           else if p = if (ex, p1, p2)
           then     if (eval (ex, s)
                    then hsem' p1 s
                    else hsem' p2 s)           lsem (compiler p) L1
           else . . .)              ⊑
```

The basis is seen to be trivial. The step is done by considering the four cases (corresponding to the four high level constructs). For each case, we successively unwind occurrences of lsem and lsemstatement (on the right hand side), until either the two sides are seen to be equivalent (in the assignment case) or until the induction hypothesis can be used (in the other three cases). We can summarise the informal proof by giving a plan or strategy for performing it:

> Do induction on hsem, then prove for arbitrary p. The basis case is trivial. For the step, divide into the four cases, and for each, do successive unwinding until either the left side and the right side are equivalent, or until the induction hypothesis can be instantiated.

This plan produces a tree of successive subgoals:



What we wish to examine, in generating the proof mechanically, is the relationship between this plan and the interaction with LCF required to prove the formula.

The other direction (⊒) has to be proved in a more complicated way. We will do induction on lsem. However, the recursiveness in the label environment makes this difficult, as we will want to "unwind" calls of it that occur recursively (such as "lsem' (comp p) L2"), and we cannot unwind an induction variable. (The step is claimed to be true for <u>all</u> lsem'.) The solution is suggested by observing the way in which the computation induction rule "unwinds" the functional once for us; what we need is a rule of iterated computation induction to unwind any number of times. fun^n x means fun (fun ... (fun x)...) n times.

```
basis:    .     w [⊥ ]           ^
                w [fun ⊥ ]       ^
                   .
                   .
                   .
                 w [fun^n-1 ⊥ ]
  step: ∀ f'.(w [ f' ]           ^
                 w [fun f' ]     ^
                     .
                     .
                     .
                 w [fun^n-1 f' ]) ⊃
                 w [fun^n f' ]
  conclusion: w [FIX fun ]
```

It is not difficult to show this is derivable from ordinary computation induction, by extending the consequent of the step to a conjuction. We have, in fact, derived this rule of inference in LCF, from the built-in rule of computation induction. (Other desired rules can be similarly derived, for example, structural induction rules.)

In the longest case (the if case) we need four unwindings (n = 4). Thus, for the other direction, the basis is:

```
 ∀ p. hsem p  ⊒       (compiler p) L1              ^
 ∀ p. hsem p  ⊒   lsemfun   (compiler p) L1        ^
 ∀ p. hsem p  ⊒   lsemfun^2   (compiler p) L1      ^
 ∀ p. hsem p  ⊒   lsemfun^3   (compiler p) L1
```

and the step is:

```
   ∀ lsem'.(∀p. hsem p  ⊒   lsem' (compiler p) L1                  ^
             ∀p. hsem p  ⊒   lsemfun lsem' (compiler p) L1         ^
             ∀p. hsem p  ⊒   lsemfun^2 lsem' (compiler p) L1)      ^
             ∀p. hsem p  ⊒   lsemfun^3 lsem' (compiler p) L1       ⊃
                  ∀p. hsem p  ⊒   lsemfun^4 lsem' (compiler p) L1
```

Recall that lsem = FIX lsemfun. Aside from the proliferation of basis cases, and the presence of functionals raised to powers, the proofs of the induction steps are very similar in the two directions, so for simplicity, we will restrict the discussion to the proof in the first direction. (The bases are proved by unwinding as well, and by proving as a lemma that lsemstatement t ⊥ ⊥ = ⊥ .)

To convey the flavour of the proof, and to suggest the level of complexity involved, we will go through a small section of it informally. Let's suppose we are proving that

```
  ∀ p. hsem p   ⊑  lsem (compiler p) L1
```

so we have assumed, as hypothesis,

```
  ∀ p. hsem' p  ⊑  lsem (compiler p) L1
```

and wish to show (for arbitrary p) the step:

```
   hsemfun hsem' p  ⊑  lsem (compiler p) L1
```

Suppose, for example, we are now considering the while case. Say that p = while (ex, p1). By unwinding hsemfun, we reduce the left-hand-side to:

```
   λs. if (eval (ex, s)
       then hsem' p1 s
       else hsem' p2 s
```

3. The Informal Proof

As was mentioned earlier, Russell's proof is in fact incorrect. It is unlikely that the formula can be proved as an equivalence (see Milne [9] .) Russell tries to prove it by doing a simultaneous computation induction on the functions hsem and compiler. The proof we have found divides the formula into a pair of inequivalences:

```
∀ p. hsem p ⊑ lsem (compiler p) L1

∀ p. hsem p ⊒ lsem (compiler p) L1
```

The first is proved by computation induction on the function hsem. The induction rule to which we appeal can be stated, for a formula w[f], and a definition |- f = FIX fun, as:

```
basis:          w [⊥ ]
step:   ∀ f'. w[f' ]  ⊃  w[fun f' ]
conclusion:     w [ FIX fun ]
```

Thus, the basis and step are, respectively:

```
∀ p.⊥ p  ⊑  lsem (compiler p) L1

∀ p.  hsem' p  ⊑  lsem (compiler p) L1      ⊃
      hsemfun hsem' p  ⊑  lsem (compiler p) L1
```

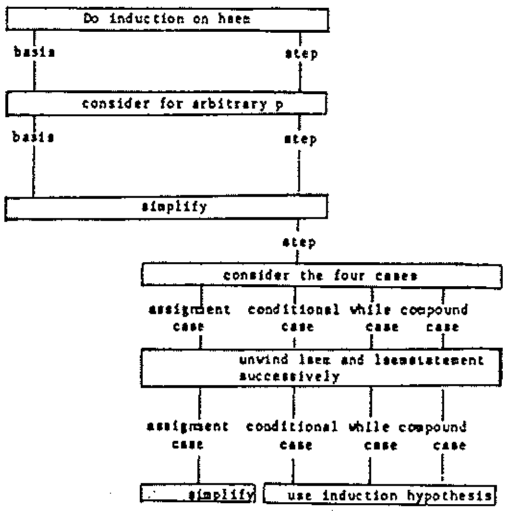
that is, for the step,

```
∀ p. hsem' p  ⊑   lsem (compiler p) L1    ⊃
    (λs. if p = assign (x, ex)
         then s [eval (ex, s) / x]
         else if p = if (ex, p1, p2)
         then    if (eval (ex, s)
                 then hsem' p1 s
                 else hsem' p2 s)
         else . . .)          ⊑      lsem (compiler p) L1
```

The basis is seen to be trivial. The step is done by considering the four cases (corresponding to the four high level constructs). For each case, we successively unwind occurrences of lsem and lsemstatement (on the right hand side), until either the two sides are seen to be equivalent (in the assignment case) or until the induction hypothesis can be used (in the other three cases). We can summarise the informal proof by giving a plan or strategy for performing it:

> Do induction on hsem, then prove for arbitrary p. The basis case is trivial. For the step, divide into the four cases, and for each, do successive unwinding until either the left side and the right side are equivalent, or until the induction hypothesis can be instantiated.

This plan produces a tree of successive subgoals:



What we wish to examine, in generating the proof mechanically, is the relationship between this plan and the interaction with LCF required to prove the formula.

The other direction (⊒) has to be proved in a more complicated way. We will do induction on lsem. However, the recursiveness in the label environment makes this difficult, as we will want to "unwind" calls of it that occur recursively (such as "lsem' (comp p) L2"), and we cannot unwind an induction variable. (The step is claimed to be true for <u>all</u> lsem'.) The solution is suggested by observing the way in which the computation induction rule "unwinds" the functional once for us; what we need is a rule of iterated computation induction to unwind any number of times. fun^n x means fun (fun ... (fun x)...), n times.

```
basis:        w [⊥ ]          ∧
              w [fun ⊥ ]      ∧
                    .
                    .
                    .
              w [fun^n-1 ⊥ ]
step: ∀ f'.(w [ f' ]         ∧
              w [fun f' ]     ∧
                    .
                    .
                    .
              w [fun^n-1 f' ]) ⊃
              w [fun^n f' ]
conclusion: w [FIX fun ]
```

It is not difficult to show this is derivable from ordinary computation induction, by extending the consequent of the step to a conjuction. We have, in fact, derived this rule of inference in LCF, from the built-in rule of computation induction. (Other desired rules can be similarly derived, for example, structural induction rules.)

In the longest case (the if case) we need four unwindings (n = 4). Thus, for the other direction, the basis is:

```
∀ p. hsem p  ⊒      (compiler p) L1          ∧
∀ p. hsem p  ⊒   lsemfun   (compiler p) L1   ∧
∀ p. hsem p  ⊒   lsemfun^2   (compiler p) L1 ∧
∀ p. hsem p  ⊒   lsemfun^3   (compiler p) L1
```

and the step is:

```
∀ lsem'.(∀p. hsem p  ⊒  lsem' (compiler p) L1                 ∧
         ∀p. hsem p  ⊒  lsemfun lsem' (compiler p) L1         ∧
         ∀p. hsem p  ⊒  lsemfun^2 lsem' (compiler p) L1       ∧
         ∀p. hsem p  ⊒  lsemfun^3 lsem' (compiler p) L1)      ⊃
             ∀p. hsem p  ⊒  lsemfun^4 lsem' (compiler p) L1
```

Recall that lsem = FIX lsemfun. Aside from the proliferation of basis cases, and the presence of functionals raised to powers, the proofs of the induction steps are very similar in the two directions, so for simplicity, we will restrict the discussion to the proof in the first direction. (The bases are proved by unwinding as well, and by proving as a lemma that lsemstatement t ⊥ ⊥ = ⊥ .)

To convey the flavour of the proof, and to suggest the level of complexity involved, we will go through a small section of it informally. Let's suppose we are proving that

```
∀p. hsem p  ⊑  lsem (compiler p) L1
```

so we have assumed, as hypothesis,

```
∀ p. hsem' p  ⊑  lsem (compiler p) L1
```

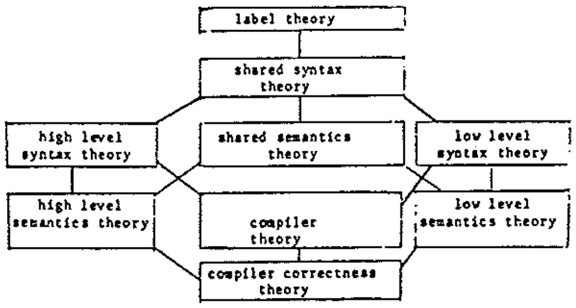
and wish to show (for arbitrary p) the step:

```
hsemfun hsem' p  ⊑  lsem (compiler p) L1
```

Suppose, for example, we are now considering the while case. Say that p = while (ex, p1). By unwinding hsemfun, we reduce the left-hand-side to:

```
λs. if (eval (ex, s)
        then hsem' p1 s
        else hsem' p2 s
```

To organise the compiler proof (as in the diagram in Section 2), we will want to build theories about the syntax of both languages, and perhaps theories of the syntax and semantics which are shared between them (just to keep things factored). For example, assignments are common to both the source language and target language syntax and stores to both semantics. We will also want theories of the semantics of both languages, each semantic theory having as a parent the appropriate syntax theory, since the semantic functions act upon syntactic entities. The compiler theory will require both syntactic theories as parents (and so will have the two syntax theories as ancestors), as the compiler is a function from source language programs to target language programs (syntactic entities). The correctness statement and proof will be in a theory having both semantic theories as parents. We can picture the structure of theories as follows:



To give a further idea of the nature of theories, the theory of syntax for the source language can be pictured as follows. EXP and ASSIGN are inherited from the shared syntax theory.

```
parent theory:  shared syntax theory
types:          HPROGRAM = ASSIGN + IF + WHILE + COMPOUND
                IF = EXP x HPROGRAM x HPROGRAM x HPROGRAM
                WHILE = EXP x HPROGRAM
                COMPOUND = HPROGRAM x HPROGRAM
constants:      mkif: IF -> HPROGRAM
                destif: HPROGRAM -> IF
                isif: HPROGRAM -> tr
                    (etc.)
```

We would also introduce axioms defining the new constants "isif", "mkif", "destif" and the rest, in terms of the injection, projection and discriminator functions for sum domains which are built into PPLAMBDA.

The theory of low level syntax is build in a similar fashion.

In the high level semantic theory, we introduce new constants "hsem" and "hsemfun", and the semantic function discussed earlier appears as an axiom:

```
parent theory:  high level syntax theory
constants:      hsem: HPROGRAM -> STORE -> STORE
                hsemfun: (HPROGRAM -> STORE -> STORE) ->
                         HPROGRAM -> STORE -> STORE

axioms:          |- hsem = FIX hsemfun
                 |- hsemfun = λhsem. λp. λs.
                    if isassign p then ...
                    else if isif p then ...
                    else if iswhile p then ...
                    else if iscompound p then ...
```

Notice the way in which the usual syntactic and semantic domain equations are reflected as new types in theories, and the semantic equations as axioms.

The last theory to be built is the compiler correctness theory. Any helpful lemmas (such as mentioned in Section 3) can be proved and recorded in this theory.

To summarise, we have sketched the structure of theories in which the problem is formalised and in which the proof takes place. Since we will work in the compiler correctness theory, all of the types and axioms and lemmas of the other theories are available. This is the first of the two main ways in which LCF gives a means of structuring and organising proofs. The second, the method of <u>tactical programming</u>, is the subject of the next section.

<u>5. A Short Tutorial on LCF: Close-up of the Proof</u>

To introduce LCF and proof <u>in</u> LCF, we will focus in on a particular section of the proof described in Section 3, showing how the proof attempt in LCF might proceed.

Suppose we have somehow got as far, in generating the proof, as the point at which we noted that the rest of the proof consisted of some uses of the induction hypothesis. Suppose that we are in the while case of the induction step. We now have a subgoal ahead of us:

```
[λs. if eval (ex, s)
    then (hsem' p) (hsem' p1 s) else s]    ⊑
[λs. if eval (ex, s)
    then (lsem (compiler p))(lsem (compiler p) L1 s) else s]
```

However, there is more to be said about it than this. For one thing, there are some assumptions that have been made, and we can use them. These are the induction hypothesis and the current case assumption.

```
∀p.  hsem' p  ⊑  lsem (compiler p) L1

  p = while (ex, p1)
```

In addition, suppose that we have proved the lemmas mentioned earlier, having anticipated that they will be helpful. (They will, in fact, already have been used in the proof, during the sequence of unfoldings that led to the current subgoal.) We might desire that these lemmas be applied automatically (as rewrites or simplifications) wherever possible. If so, we want to include them as part of the subgoal, so that they are available when needed. Thus we will define a <u>goal</u> to be a triple, consisting of a formula (to be proved), a set of useful simplification rules, and a list of formulae representing the current useable assumptions. The ability to define goals so that we can keep track of this relevant auxiliary information is another aid provided by LCF for structuring proofs. Now, our subgoal can be denoted as:

```
formula:   [if eval (ex, s) then (hsem' p)(hsem' p1 s) else s] ⊑
           [if eval (ex, s) then (lsem (comp p) L1)
                                 (lsem (comp p1) L1 s)]
simpset:   |-∀p'. p' = while (ex, p1) ⊃
               lsem (comp p') L1 = lsemstatement(ifnot(ex,L4))
                     (lsem (comp p'))(lsem (comp p') L2) lsem
                 (etc.)
assumptions: p = while (ex, p1)
             ∀p.  hsem' p  ⊑  lsem (compiler p) L1
```

Next, we will have to be precise about what it means to <u>achieve</u> a goal. We will be satisfied, in regard to our current goal, if we can produce a theorem matching the formula part of the goal and depending on no assumptions beyond those in the assumptions list part of the goal (or on which the simplifications depend).

Let us use the following notation for the goal-subgoal relation:

```
                g
     ===================================
            [g1; g2;  ... ; gn]
```

to mean that in order to achieve the goal g, one should try to achieve all of the subgoals in the <u>list</u> below.

We would now like to design a detailed plan for working on current subgoal -- but we'll be more general. In each of the four cases of the proof, we expect to arrive at some point at which either both sides of the formula (an inequivalence, we'll presume) are easily shown to be equal, or at which an instance (or several instances) of the induction hypothesis will be "embedded" in the formula. According to the plan, we should either notice that a trivial simplification is needed to finish, or if that is not the case, try "taking apart" the terms on either side of the inequivalence in an effort to find an instance of the induction hypothesis. Suppose we take into account three ways in which the induction hypothesis instance can be embedded; the terms on either side of the inequivalence might be lambda-expressions, they might be combinations, or they might be conditionals. The three subplans needed, then, can be denoted by the figures:

```
        (λs. t1) ⊑ (λs. t2),     ss,   aset
      ----------------------------------------
        [t1 ⊑ t2,    ss,    aset]


 (if t1 then t2 else t3) ⊑ (if t1 then u2 else u3),ss,aset
 -----------------------------------------------------------
      [t2 ⊑ u2,  ss u (t1 = TT),  aset u (t1 = TT);
       t3 ⊑ u3,  ss u (t1 = FF),  aset u (t1 = FF)]


         (t1 u1) ⊑ (t2 u2),   ss,   aset
         ------------------------------
              [t1 ⊑ t2,   ss,   aset;
               u1 ⊑ u2,   ss,   aset]
```

In each case, that is, to achieve the goal (on top), one should try to achieve the subgoal(s) in the list (on the bottom). This goal-subgoal relation is one feature of the subplans. The other feature is some information on exactly how to achieve the goal, once the subgoal(s) is (are) achieved successfully. When the subgoals have been achieved, we will be in possession of lists of theorems of the form (respectively):

```
[ A |- t1 ⊑ t2]

[ A |- t2 ⊑ u2;     A |- t3 ⊑ u3]

[A |- t1 ⊑ t2;     A |- u1 ⊑ u2]
```

where the assumptions on the left all belong to the assumptions lists of the corresponding subgoal. What we want now is a method for obtaining the theorems which (respectively) achieve the original goals. That is, we need functions taking theorem lists as above to theorems

```
... |- (λs. t1) ⊑ (λs. t2)

... |- (if t1 then t2 else t3) ⊑ (if t1 then u2 else u3)

... |- (t1  u1) ⊑ (t2  u2)
```

Those functions are built up from the rules of inference for PPLAMBDA provided in LCF. In the first case, we simply use the basic inference rule LAMGEN. LAMGEN is of type (term -> theorem -> theorem). We'll denote this rule of inference using the following notation:

```
    s,   A |- t1 ⊑ t2
  -------------------
    A |- (λs.t1) ⊑ (λs. t2)     where s is not free in A
```

noting the proviso. In general, inference rules take one or more theorems to a new theorem. What is vital here is that the built-in rules of inference (or ML programs constructed from them) are the only functions which can return theorems as results. The type-checking descipline in ML assures this. Thus it's impossible to create theorems not derivable in the logic.

The method of achieving an original goal will be represented as a function taking a list of theorems respectively achieving the list of subgoals to a theorem achieving the original goal. In the first case, that function will be (given the term s):

```
λthlist. LAMGEN s (head thlist)
```

Such a function is called a proof (a defined type in ML).

The proofs for the other two subplans are similar, using other PPLAMBDA inference rules for the monotonicity of function application and of conditionals.

A proof plan (henceforth called a tactic) is another defined ML type. A tactic takes a goal (to be proved) to a pair consisting of a list of goals (the subgoals) and a proof, a function from a theorem list to a theorem, providing a way of achieving the goal from achievments of its subgoals.

We now have the basic tools for discussing tactical programming. The tactic described thus far can be written as a procedure in ML. We'll call it LAMGENTAC because it is in an obvious sense the inverse of the inference rule LAMGEN. In the informal notation described earlier, the procedure would be something like:

```
LAMGENTAC (w, ss, aset) =
        if w = (λs. t1) ⊑ (λs. t2)
             then [t1 ⊑ t2 ,   ss,   aset],
                  (λthmlist. LAMGEN s (head thmlist))
```

If w were not of the form indicated, LAMGENTAC would fail (to be applicable). Tactics for the other two subplans are programmed similarly. We'll call the other two tactics CONDTAC (for conditional) and COMBTAC (for combination), respectively.

Let's return to our top-level plan (tactic); we wish to take apart the terms on either side of an inequivalence, see whether the induction hypothesis can be used, and to stop when all that's needed is a trivial simplification. As an example, suppose we have generated subgoals whose formula parts are

```
          t ⊑ t,
(or)      ⊥ ⊑ t
```

So as not to burden the user with proof at this level, a standard tactic (called SIMPTAC) is provided in LCF to do such simple bits of reasoning (among others), as well as to do any simplifications or rewrites suggested in the simplification set of a goal. Applied to a goal, SIMPTAC repeatedly performs a set of basic simplifications and all of the suggested rewrites until no more can be done. It will return a list of the remaining subgoals, if there are any, as well as a proof function justifying its simplifications. The ability to do simplifications automatically plays a large part in making the generation of long proofs feasible in LCF.

The next step in building the tactic is to tie together the three (sub)tactics that we have designed. Given a goal (that does not immediately submit to simplification), we wish to try LAMGENTAC, and failing that CONDTAC, and failing that COMBTAC (the order isn't important). For this purpose, we use tacticals ; tacticals are functions which build compound tactics from known ones. The tactical ORELSE takes two tactics, TAC1 and TAC2, and returns a tactic which applies TAC1 to a goal -- unless TAC1 isn't applicable, in which case it applies TAC2. The tactical THEN also takes a pair of tactics, TAC1 and TAC2; it returns a tactic which given a goal, applies TAC1 to the goal, and TAC2 to each of the subgoals thus produced. REPEAT takes a single tactic, TAC1, and returns a tactic which applies TAC1 to a goal, then to each of the resulting subgoals, etc., until it is no longer applicable.

We can now build up a compound tactic (call it TAKE-APART-TAC) to do more of what we want done:

```
TAKE-APART-TAC = SIMPTAC THEN
                 (LAMGENTAC ORELSE CONDTAC ORELSE COMBTAC)
```

Finally, we will write and use a tactic (called USE-ASSUMPTION- TAC) which searches the assumptions list of a goal for a formula that matches the one to be proved, and returns an appropriate subgoal:

```
   p' ⊑ q',   ss,  [...;   x1...xn. p ⊑ r; ...]
 ------------------------------------------------
   r' ⊑ q',   ss,  [...;   x1...xn. p ⊑ r; ...]
```

where r' is r with the instantiations used in matching p to p'.

Applied to the subgoal with the formula part

```
hsem' p  ⊑  lsem (compiler p) L1
```

(which will be got to eventually in the proof effort), USE-ASSUMPTION-TAC will return a subgoal with a formula part

```
lsem (compiler p) L1 ⊑ lsem (compiler p) L1
```

by matching to the induction hypothesis in the assumptions list. The above subgoal can be handled by SIMPTAC. The whole tactic we want, then, can be expressed as:

```
REPEAT ((USE-ASSUMPTION-TAC THEN SIMPTAC)
              ORELSE
        (TAKE-APART-TAC THEN SIMPTAC))
```

Applied to our subgoal, this compound tactic yields an empty goal list and a proof that will take an empty theorem list and return a theorem achieving the subgoal.

Using the general tactic (we'll call it FINISH-UP-TAC) saves us having to know the course of the proof in advance and it works for all four cases of the proof (in both directions). Of course, FINISH-UP-TAC could be made more general still. In ML, one can program tactics at whatever level of generality one wants or is capable of.

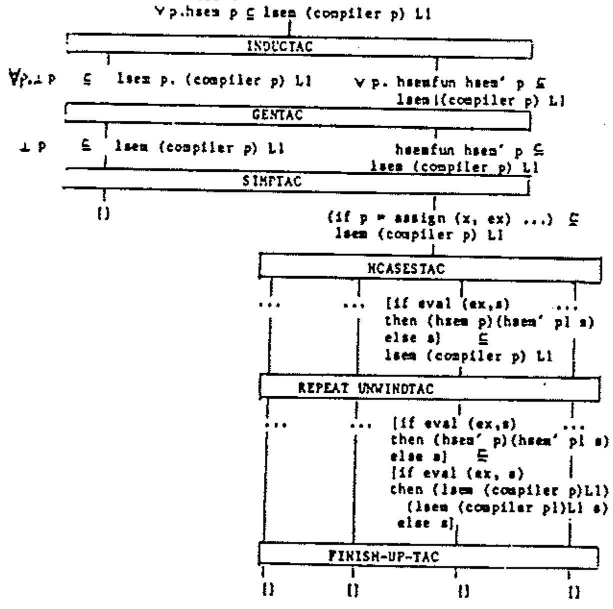
LCF provides a setting in which strategies like the one we've just developed can be expressed, varied, combined, improved and analysed; it's a tool for the study of patterns of reasoning, and the study of machine-assisted deduction. The ways in which this is so should become yet more evident in the next section, in which we discuss the performance of the whole proof.

6. The Whole Proof

Bearing in mind the plan we've sketched for carrying out the proof (Section 3), as well as the notion (and notation) of tactics in LCF, we're now prepared to dicuss the proof of the correctenss of the compiler as a whole. The following tactic, (we'll call it WHOLE-TAC), when applied to the ⊆ -direction goal, gives an empty list of subgoals, and the desired proof:

```
INDUCTAC THEN GENTAC THEN SIMPTAC THEN
THEN HCASESTAC THEN
REPEAT UNWINDTAC THEN FINISH-UP-TAC
```

The component tactics, of course, need to be explained (except FINISH-UP-TAC, which is as described in Section 5) -- but the tree of successive subgoals produced by the component tactics should give some idea of what the component tactics do. (The tree is traced out for the while case). The correspondence between this tree and the tree in Section 3 should be noted .



∀p.hsem p ⊆ lsem (compiler p) L1
INDUCTAC
∀p.⊥ p ⊆ lsem p. (compiler p) L1
∀p. hsemfun hsem' p ⊆ lsem ((compiler p) L1
GENTAC
⊥ p ⊆ lsem (compiler p) L1
hsemfun hsem' p ⊆ lsem (compiler p) L1
SIMPTAC
[]
(if p = assign (x, ex) ...) ⊆ lsem (compiler p) L1
HCASESTAC
...
...
[if eval (ex,s) then (hsem p)(hsem' p1 s) else s] ⊆ lsem (compiler p) L1
...
REPEAT UNWINDTAC
...
...
[if eval (ex,s) then (hsem' p)(hsem' p1 s) else s] ⊆ [if eval (ex, s) then (lsem (compiler p)L1) (lsem (compiler p1)L1 s) else s]
...
FINISH-UP-TAC
[]
[]
[]
[]

When WHOLE-TAC is applied to the main goal (shown at the top node of the tree), an empty goal list and a proof are returned immediately. The tree gives us the abstract <u>control structure</u> for generating the proof.

We <u>could</u>, of course, do the proof by applying the component tactics one-by-one to the various subgoals, and saving the proof functions in order to work back up again when empty subgoal lists were produced (by FINISH-UP-TAC). In practice, in fact, one might very well wish to experiment with component tactics for the more difficult subgoals before composing a large tactic which, like WHOLE-TAC, finishes the proof in one stroke.

Let's now consider the sub-tactics of WHOLE-TAC. We start with no assumptions, but we will put various simplification rules in the simplification set, reflecting the lemmas which will be useful (Section 3). The goal is:

```
formula: ∀p. hsem p ⊆ lsem (compiler p) L1
simpset:  |- ∀p'.p' = assign (x, ex) ⊃
                 lsem (comp p')  L1 =
                 lsemstatement (assign (x,ex))
                               (lsem (comp p'))
                               (lsem (comp p') L2)
                               lsem
                  (etc.)
assumptions:   none
```

Among the standard tactics provided in LCF is INDUCTAC, which takes the least-fixed-point definition of the function on which induction is to be done, and the goal to be proved. It returns a list of two subgoals, the basis case and the step, where the step has a new assumption in its list, namely, the induction hypothesis. For the definition |- f = FIX fun, and the formula w, INDUCTAC is:

```
         w,  ss,  aset
----------------------------------
        [w [⊥], ss aset;
w [fun f' ], ss, aset u (w [ f' ]) ]
```

The proof part of INDUCTAC uses the LCF inference rule INDUCT:

```
      A1 |- w [⊥]        A2 , w [f' ] |- w[fun f' ]
  ----------------------------------------------------
                A1 u A2 |- w [FIX fun ]
```

After applying INDUCTAC to the subgoal, we apply SIMPTAC, and this finishes the proof in the basis case. We now want another standard tactic, GENTAC, to give as a subgoal the goal for an arbitrary program p':

```
   ∀x. w,   ss,   aset
------------------------------
     w , ss, aset
```

where x is not free in aset.

Now we have a subgoal:

```
formula: [if p = assign (x, ex)
          then λs. s [eval (ex, s) / x]
          else if p = if (ex, p1, p2)
          then λs. if eval (ex, s)
                   then hsem' p1 s
                   else hsem' p2 s
          else if p = while (ex, p1)
          then λs. if eval (ex, s)
                   then (hsem' p)(hsem' p1 s)
                   else s
          else if p = compound (p1,p2)
          then (hsem' p2)(hsem' p1 s)]
                   ⊆
          lsem (compiler p) L1
simpset:  the initial one
assumptions: the initial ones  as well as
             ∀p. hsem' p ⊆ lsem (compiler p) L1
```

We would now like to divide the step into cases, for the various program constructs. For this purpose we could use CASESTAC the standard tactic in LCF to break a goal into three subgoals, according to whether a (given) term is TT, FF or ⊥ , adding the case assumptions to the assumptions lists and simplification sets. One application of CASESTAC would produce the cases for which p = assign (x, ex) is TT, FF and ⊥ ; simplification will take care of the undefined case. In the false case, we would have to use CASESTAC again, and so on. A useful tactic here would do this nested case analysis for us, and might be called HCASESTAC (for high-level cases). (It could be written in terms of CASESTAC.) In any case, we will end up with four subgoals, for the four cases. The while subgoal, in particular, will be:

```
formula:   λs. if eval (ex,s)
               then (hsem'p)(hsem' p1 s)
               else s    ⊆
               lsem (compiler p) L1
simpset:   initial one and the  case assumptions
assumptions:  ∀p. hsem' p ⊆ lsem (compiler p) L1
               p = while (x, ex)
```

In this (and the other cases) we wish to do the sequence of unwindings of lsem and lsemstatement that we did in the informal proof. The tactic to do the unwinding (call it UNWINDTAC) will unwind the definitions of lsem and lsemstatement once each, until the induction hypothesis occurs in the formula part of the current subgoal, and then simplify. Finally, we use FINISH-UP-TAC to complete the proof.

There are many possibilities, of course, for writing more general and automatic tactics. For example, a tactic could be designed to try to find suitable induction hypotheses, or to generate lists of the useful lemmas by examining subgoals. What should be clear is the way in which tactics can be programmed in ML, and combined using tacticals, to represent all of these strategies, at their various levels of generality.

7. Conclusion

The concepts illustrated in the discussion of the proof of Russell's compiler are those of goal-oriented proof and tactical programming. In LCF, instead of beginning with a set of axioms and rules and working laboriously towards a goal, one can begin with the goal, and continue to generate subgoals until a trivial-to-prove set of subgoals is found. The generation of subgoals (by the application of tactics which reflect proof strategies), is a natural style of proof. It corresponds to the way in which proofs are abstracted, or proof plans suggested, when people communicate these things to each other. Tactical programming allows varying degrees of automation in proof attempts. At one extreme, tactics can generate subgoals at a basic level (tactics which are 'inverses' of basic inference rules do this), requiring the user to be aware of the detailed course of the proof. Toward the opposite extreme, the user can experiment with tactics to prove the main goal in one stroke. In this style, several alternative tactics can be combined into one, using, for example, the tactical ORELSE; compound tactics can be designed which analyse the results of applying one tactic in order to find another which is more apt. Tactics can even be written to implement automatic theorem proving strategies. The user, working abstractly, need not be aware of the details of the proof. What one ends up with is something that might becalled a 'proof story' or a high-level proof, rather than a proof in the sense of a sequence of theorems which are either axioms or follow from earlier entries in the sequence by rules of inference. Although one could arrange for this sequence of primitive steps to be generated, it is part of the underlying philosophy of LCF that the high-level proof -- the structure of the proof and the tactics which generated it -- are more intelligible and more relevant to understanding the proof. The structure of the proof attempt affords a basis for making generalisations, for proving similar theorems, for making remarks about what proofs (of the sort in question) require. This is the basis for the claim that LCF provides a setting for a general study of the sorts of reasoning needed in proofs of program properties.

As mentioned, we plan to extend the work described by designing and formalising a whole sequence of languages, beginning with an Algol-like one with block structuring and procedure calls, and progressing toward an assembly-like one . We will divide the compilation process into stages, each stage focusing on one (or a cluster of related) feature(s). We will then give the semantics of the languages, respectively, and prove that each stage of the compilation preserves semantics for the pair of languages in question.

This project will obviously depend on the theory-building facilities of LCF for manageable organisation and modularity. It will depend on LCF's capacity for handling detail automatically. Most importantly, it will depend on (and test!) the concept of tactical programming. Clearly, the proof outlined for this stage of the compilation is very complex; the hope is that the same, or a similar, tactic can be used for the other stages as well. What we seek, in fact, is a coherent body of general tactics (for proving compilers), tactics which can be used for the compilers at any of the levels discussed, or for other compilers, suitably formulated. If we succeed in defining a body of tactics of this sort, we feel we'll have made some progress toward developing a methodology for machine-assisted program proof.

8. Related Work

Other work on compiler correctness proofs varies in motivation and level of formality sought. The early work (for example, by London, [7], and by McCarthy and Painter, [8]), dealt primarily with giving informal proofs for compilers for applicative languages (LISP subsets). Some work also concentrated on the use of algebraic methods in compiler proofs (Burstall and Landin, [3]).

In relation to Edinburgh LCF, the original Stanford LCF was based on a similar logic, but did not include a meta-language with which to manipulate objects in the logic, so that it had no facilities for tactical programming. It did include facilities for doing automatic simplifications, similar to the current ones. Stanford LCF was closer to a proof-checking system. R. Milner and R. Weyhrauch, 1972 [10], designed, and partially carried out, a plan for performing the proof of a compiler. The source language was similar to the one we have described in including assignments, conditionals, while statements and sequencing; expressions were also considered. The target language was much lower level, including stack handling commands. A decision was made to structure the proof by introducing concepts from universal algebra, and proving, as subgoals to the main goal, that the high-level semantic function, the compiler, the low-level semantic function, etc., were all homomorphisms. A structure of eleven subgoals was outlined, seven of which were machine checked. The main conclusion of the work was that additional structuring would be required to make the proof effort feasible and the results intelligible.

The work described here takes advantage of a more sophisticated LCF system, in several ways. The addition of ML to the system, and the consequent facilities for tactical programming, as well as the facility which allows the development of theories, make LCF a proof performing system, rather than a proof checker. The initial work described here seems to indicate that the use of theories and of tactical programming, as well as the factoring of the compilation and the proof into stages, may provide enough structure that algebraic concepts are not required. The end result is not a proof simply, accompanied by some history of its development, as it would have been in Stanford LCF, but a collection of (hopefully) general tactics for doing similar proofs.

Another compiler proof attempt was made in Stanford LCF by M. Newey, in 1975 [11]. Newey concerned himself with a rather different problem, proving the correctness of a compiler for a subset of LISP (into LAP code). An operational (interpretive) semantics was given for the source language, and a body of theories was presented for the integers, equality, and other necessary objects and relations. The proof was planned in detail, and a "high level goal structure" sketched, but the proof was deemed too long to be carried out in Stanford LCF.

A comparison to the current work is complicated by the choice of LISP as the source language. An operational semantics is, in a sense, the natural one for LISP, as it is not, for an Algol-like language. Such a semantics is closer in structure to the semantics for the target language, which probably makes the proof easier. A similar use of theories is made in the two projects, although they can be better structured in the current LCF.

Acknowledgements

This work involved the Edinburgh LCF system, developed by Mike Gordon, Robin Milner, Lockwood Morris, Malcolm Newey and Chris Wadsworth. My special thanks to my supervisor, Robin Milner, and to Chris Wadsworth and Mike Gordon.

9. Bibliography

1. R.S. Boyer and J S. Moore, "Proving theorems about LISP functions", JACM 22,1 (1975)

2. L.S. van Benthem Jutting, "Checking Landau's 'Grundlagen' in the Automath system", Thesis, Technische Hogeschool, Eindhoven (1977)

3. R.M.Burstall and P.J. Landin, "Programs and their proofs: an algebraic approach", Machine Intelligence 4, Edinburgh University Press (1969)

4. A. Cohn, forthcoming Ph.D. Thesis, Computer Science Department, Edinburgh University

5. M. Gordon, R. Milner and C. Wadsworth, "Edinburgh LCF", Computer Science Department, Edinburgh University, CSR-11-77, (1977)

6. F.W. von Henke and D.C. Luckham, "A methodology for verifying programs", Proceedings of the International Conference on Reliable Software, Los Angeles, Ca. (1975)

7. R.L. London, "Correctness of a compiler for a LISP subset", Proceedings of the Conference on Proving Assertions about Programs, New Mexico State University (1972)

8. J. McCarthy and J.A. Painter, "Correctness of a compiler for arithmetic expressions", Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science, Providence, Rhode Island: American Mathematical Society (1967)

9. R. Milne, "Verifying the Correctness of Implementations", lecture notes, (1977)

10. R. Milner and R. Weyhrauch, "Proving compiler correctness in a mechanized logic", Machine Intelligence 7, Edinburgh University Press, (1972)

11. M. Newey, "Formal semantics of LISP with applications to program correctness", Computer Science Department, STAN-CS-75-475, Stanford University (1975)

12. B. Randell and L.J. Russell, "Algol 60 Implementation", Academic Press Inc., London (1964)

13. J.A. Robinson, "A machine-oriented logic based on the resolution principle", JACM 12,1 (1965)

14. B. Russell, "Implementation correctness involving a language with goto statements", SIAM Journal of Computing, Vol.6, No. 3, (1977)

15. R.W. Weyhrauch, "A users manual for FOL", Computer Science Department, Stanford University STAN-CS-77-432, (1977)

DECIDING LINEAR INEQUALITIES BY COMPUTING LOOP RESIDUES

Robert E. Shostak
SRI International

ABSTRACT

V. Pratt [18] has shown that the real and integer feasibility of sets of linear inequalities of the form $x \leq y + c$ can be decided quickly by examining the loops in certain graphs. We generalize Pratt's method, first to real feasibility of inequalities in two variables and arbitrary coefficients, and ultimately to real feasibility of arbitrary sets of linear inequalities. The method is well suited to applications in program verification.

1. Introduction

Procedures for deciding whether a given set of linear inequalities has solutions often play an important role in deductive systems for program verification. Array bounds checks and tests on index variables are but two of the many common programming constructs that give rise to formulas involving inequalities. A number of approaches have been used to decide the feasibility of sets of inequalities [2,7,8,14,21], ranging from goal-driven rewriting mechanisms [25] to the powerful simplex techniques [7] of linear programming.

The method presented here has the generality of the full-scale techniques without sacrifice of speed on the rather trivial problems one encounters most often. It builds on V. Pratt's observation [18,16] that most of the inequalities that arise from verification conditions are of the form $x \leq y + c$, where x and y are variables and c is a constant. Pratt showed that a conjunction of such inequalities can be decided quickly by examining the loops of a graph constructed from the inequalities of the conjunction. We generalize this approach, first to inequalities with no more than two variables and with arbitrary coefficients, and then to arbitrary linear inequalities. Our generalization reduces to Pratt's test for inputs having the simple structure he describes.

The discussion is presented in six sections, Sections 2 and 3 are concerned with preliminary definitions and with a statement of the method for inequalities with two variables and arbitrary coefficients. Section 4 discusses issues of complexity and usefulness for integer problems, and relates the method to Pratt's. Sections 5 and 6 deal with the extension of the method to sets having strict inequalities and to sets with arbitrary linear inequalities. The last section presents a proof of the theorem that underlies the method.

2. Definitions

Let S be a set of linear inequalities each of whose members can be written in the form $ax + by \leq c$, where x, y are real variables and a, b, c are reals. Without loss of generality, we require that all variables appearing in S other than a special variable v_0, called the <u>zero variable</u>, have nonzero coefficients. We also assume that v_0 appears only with coefficient zero.

Construct an undirected multi-graph G from S as follows. Give g a vertex for each variable occurring in S and an edge for each inequality. Let the edge associated with an inequality $ax + by \leq c$ connect the vertex for x with the vertex for y. Label each vertex with its associated variable* and each edge with its associated inequality. G is said to be the <u>graph</u> for S.

This work was supported by NSF, AFOSR, and RADC.

*In what follows, it is notationally convenient to write v for both the variable v and the vertex associated with that variable.

Now let P be a path through G, given by a sequence $v_1, v_2, \ldots, v_{n+1}$ of vertices and a sequence $e_1, e_2, \ldots, e_n$ of edges, $n \geq 1$. The triple sequence for P is given by:

$$\langle a_1, b_1, c_1\rangle, \langle a_2, b_2, c_2\rangle, \ldots, \langle a_n, b_n, c_n\rangle$$

where for each i, $1 \leq i \leq n$, $a_i v_i + b_i v_{i+1} \leq c_i$ is the inequality labeling e_i.* P is admissible if, for $1 \leq i \leq n - 1$, b_i and a_{i+1} have the opposite signs; i.e., one is strictly positive and the other is negative.

Intuitively, admissible paths correspond to sequences of inequalities that form transitivity chains. For example, the sequence $x \leq y$, $y \leq z$, $z \leq 3$ gives rise to an admissible path, as does

$$2x \geq 3y - 4,\ 2y \geq 4 - z,\ -z \geq x \quad .$$

Note that the sequence:

$$x \leq y,\ y \leq z,\ -z \leq r$$

cannot label an admissible path, since the coefficients of z have the wrong relative signs.

A path is a loop if its first and last vertices are identical. A loop is simple if its intermediate vertices are distinct.

Note that the reverse of an admissible loop is always admissible, and that the cyclic permutations of a loop P are admissible if and only if a_1 and b_n are of opposite sign, where $\langle a_1, b_1, c_1\rangle \ldots \langle a_n, b_n, c_n\rangle$ is the triple sequence for P. In this case, we say P is permutable. Note also that, since v_0 appears in S only with coefficient 0, no admissible loop with intial vertex v_0 is permutable.

Now define, for a given admissible path P, the residue inequality of P as the inequality obtained from P by applying transitivity to the inequalities labeling its edges. For example, if the inequalities along P are

$$x \leq 2y + 1,\ y \leq 2 - 3z,\ -z \leq w \quad ,$$

we have:

$$\begin{aligned} x &\leq 2y + 1 \leq 2(2 - 3z) + 1 \\ &\leq 2(2 + 3w) + 1 = 6w + 5 \end{aligned}$$

The residue inequality of P is thus $x - 6w \leq 5$.

More formally, define the residue r_p of P as the triple $\langle a_p, b_p, c_p\rangle$ given by:

$$\langle a_p, b_p, c_p\rangle = \langle a_1, b_1, c_1\rangle * \langle a_2, b_2, c_2\rangle * \ldots * \langle a_n, b_n, c_n\rangle ,$$

where $\langle a_1, b_1, c_1\rangle \ldots \langle a_n, b_n, c_n\rangle$ is the triple sequence for P and where $*$ is the binary operation on triples defined by:

$$\langle a, b, c\rangle * \langle a', b', c'\rangle = \langle kaa', -kbb', k(ca' - c'b)\rangle$$

$$\text{and } k = \frac{a'}{|a'|}$$

The residue inequality of P is then given by $a_p x + b_p y \leq c_p$, where x and y are the first and last vertices, respectively, of P.

It is straightforward to show that $*$ is associative, so that r_p is in fact uniquely defined. The idea that the residue inequality of a path is implied by the inequalities labeling that path is expressed in the following lemma:

Lemma 1. Any point (i.e., assignment of reals to variables) that satisfies the inequalities labeling an admissible path P also satisfies the residue inequality of P.

Pf. Straightforward by induction on the length of P.

*In the case where v_i and v_{i+1} happen to be identical (i.e., e_i is a self-loop), an arbitrary choice is made as to the ordering of the first two components of the associated triple.

3. Procedure for Inequalities with Two Variables

In the case where P is a loop with initial vertex, say, x, Lemma 1 asserts that any point satisfying the inequalities along P must also satisfy $a_p x + b_p x \le c_p$. If it happens that $a_p + b_p = 0$ and $c_p < 0$, the residue inequality of P is false, and we say that P is an infeasible loop.

It follows that a set S of inequalities is unsatisfiable if the graph G for S has an infeasible loop. The converse, however, does not hold in general. Figure 1, for example shows the graph for $S = \{x \le y, 2x + y \le 1, z \le x, w \le z, z \le 1 + w, z \ge \frac{1}{2}\}$. Although S is unsatisfiable, the graph has no infeasible loops, simple or otherwise.

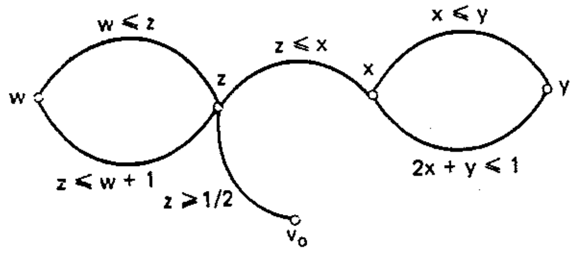


FIGURE 1 GRAPH G FOR S = $x \leq y, 2x + y \leq 1, z \leq x, w \leq z, z \leq w + 1, z \geq$

The gist of our main theorem is that G can be modified to obtain a graph G' that has an infeasible simple loop if and only if S is unsatisfiable:

Definition: Let G be the graph for S. Obtain a closure G' of G by adding, for each simple admissible loop P (modulo permutation and reversal) of G a new edge labelled with the residue inequality of P.

Note that closures are not necessarily unique, since the initial vertex of each permutable loop can be chosen arbitrarily.

Theorem: S is unsatisfiable if and only if G' has an infeasible simple loop.

Figure 2 shows the unique closure of the graph of Figure 1. Note that the only loop of G contributing an edge to G' is the xyx loop. The v_0xzv_0 loop of G' is infeasible (having residue (0, 0, -1/3)); hence the example S, according to the theorem, must be unsatisfiable.

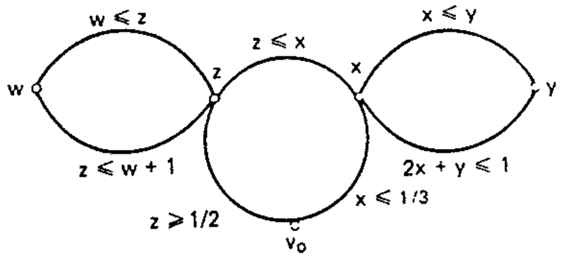


FIGURE 2 CLOSURE OF G

We show later that any cyclic permutation of an infeasible permutable loop is itself infeasible, and that the reverse of an infeasible loop is also infeasible. We thus have the following decision procedure for satisfiability of S:

(1) The simple admissible loops of G are enumerated modulo cyclic permutation and reversal, and their residues are computed. If any loops are found to be infeasible, S is unsatisfiable.

(2) Otherwise, the closure of G is formed by adding a new edge for each residue inequality. The residues of all newly formed simple admissible loops are now computed. If any are found to be infeasible, S is unsatisfiable. Otherwise S has solutions.

Note that this procedure, as stated, does not actually construct a solution if S is feasible. The proof of the main theorem, given in Section 7, provides such a construction. Note also that the new admissible loops formed in (2) must have initial vertex v_0.

4. Efficiency and Other Issues

Any implementation of the procedure must, of course, incorporate some means of generating the simple loops of a graph. For this purpose, several algoriths exist (Johnson [13], Read and Tarjan [19], Szwarcfiter and Lauer [23]) that operate in time order $\ell|V| + |E|)$, and space order $|V| + |E|$, where ℓ is the number of loops generated. These algorithms are easily modified to generate only admissible loops without adversely affecting efficiency. Since each loop has length on the order of $|V|$, these algorithms require little more time than that needed for output. A graph may, of course, have quite a few simple loops - exponentially many (in $|E|$), in fact, in the worst case. One can show that the procedure we have described, like the simplex method, exhibits exponential worst-case asymptotic behavior.

In practice, however, one does not encounter such behavior. The sets of inequalities that arise from verification conditions usually have the form of transitivity chains. The corresponding graphs are treelike, seldom having more than a few loops. Most of the loops that do occur are 2-loops, which are easily tested at the time the graph is constructed.

V. Pratt [18] has noted that these sets often fall within what he has termed separation theory. All the inequalities of such sets are of the form $x \le y + c$. The residue of a loop whose labeling inequalities are of this form is given by one of $\langle 1, -1, m\rangle$, $\langle -1, 1, m\rangle$, where m is the sum of the constants c around the loop. The graph for a set S in separation theory is thus its own closure, so the main theorem of the last section reduces, in this case, to Pratt's observation that such a set S is infeasible if and only if the sum of the constants around some simple loop is negative. Pratt notes that this condition can be tested in order $(|V| + |E|)^3$ time by taking a max/+ transitive closure of the incidence matrix of the graph. In practice, however, it may be more efficient to generate loops using one of the algorithms mentioned earlier.

Note that a set of inequalities in separation theory with integer constants is integer feasible if and only if it is real feasible. While the main theorem therefore decides integer feasibility in this case, it cannot decide integer feasibility in general. It has been observed [21], however, that the transformations Bledsoe [3] describes for reducing formulas in integer arithmetic to sets of inequalities tends to produce sets that are integer feasible if and only if they are real feasible. The main theorem thus provides a useful, but not complete, test for integer feasibility.

5. Strict Inequalities

The procedure is trivially generalized to handle strict inequalities (i.e., inequalities of the form $ax + by < c$). Let an admissible loop be strict if one or more of its edges is labeled with a strict inequality. A strict loop P with residue $\langle a_p, b_p, c_p\rangle$ is infeasible if $a_p + b_p = 0$ and $c_p \le 0$. If the definition of closure is now modified in such a way that new edges arising from strict loops are labeled with strict inequalities, the main theorem still holds.

6. Extension to Arbitrary Sets of Inequalities

The method can be further generalized to sets of inequalities with arbitrary coefficients and arbitrary numbers of variables.

The basic idea is illustrated by the following example. Consider the set

$$S = \{x \le y,\ y \le z,\ z \le y - x + 1,\ x \ge 2\} \quad .$$

Note that the inequality $z \le y - x + 1$ has three variables. As shown in Figure 3, we choose two of the three (say z and y) as the endpoints of the edge corresponding to this inequality in the graph G for S. The term $(-x + 1)$ becomes the "constant" of this inequality. The residue of the only simple loop (y z y) is given by

$$\langle 1, -1, 0\rangle * \langle 1, -1, -x + 1\rangle$$

and is computed "symbolically" to obtain $\langle 1, -1, -x+1 \rangle$ Note that this loop is infeasible unless $-x+1 \geq 0$. If the residue inequality $-x+1 \geq 0$ is now added to the graph, an infeasible simple loop $(v_0 x v_0)$ results, thus making S unsatisfiable.

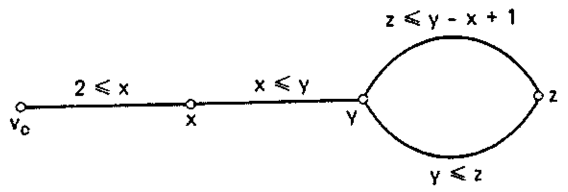


FIGURE 3 GRAPH G FOR $x \leq y$, $y \leq z$, $z \leq y - x + 1$, $x \geq 2$

We now describe the procedure for an arbitrary set S. We assume that the variables of S other than v_0 are ordered in some way. Each variable that is the lowest or second lowest ranked variable in every inequality in which it appears is said to be a primary variable. We adopt the convention that the edge corresponding to a given inequality is always attached to the two nodes corresponding to its primary variables. If it has only one primary variable, one end is attached to v_0, and if it has no primary variables, both ends are attached to v_0. The procedure is as follows:

(1) Compute a closure G' of the graph G for S as usual, evaluating residues "symbolically" as in the example. If G' has an infeasible loop, terminate returning "unsatisfiable." Otherwise, if all variables of S are primary, terminate returning "satisfiable."

(2) Otherwise, repeat the procedure using the set of residue inequalities of G' in place of S.

Note that the procedure must terminate since the number of non-primary variables must decrease at each iteration. One can prove as an extension of the main theorem that the general procedure is complete as well as sound.

R. Tarjan* has observed that any set of inequalities can be polynomially transformed to one with no more than three variable per inequality through the addition of new variables. The inequality $w + x + y + z \leq 1$, for example, is replaced by $w + x \leq v$, $w + x \geq v$, $v + y + z \leq 1$. For sets with inequalities having no more than three variables, only two iterations of the procedure are ever required. There does not seem to be any fast way to transform a set of inequalities to one having inequalities with no more than two variables.

7. Proof of the Main Theorem

It follows from Lemma 1 and from the definition of closure that a set S of inequalities (each having no more than two variables) is satisfiable if and only if S' is satisfiable, where S' labels the edges of a closure of the graph for S. If we define a closed graph as one that is a closure of itself, the main theorem can thus be restated as follows:

Theorem: If G is a closed graph for S, then S is satisfiable if and only if G has no infeasible simple loop.

The proof of the theorem requires a number of technical lemmas. Proofs are omitted for the more trivial of these.

Notation: Where P and Q are paths, let PQ denote the concatenation of P with Q.

Lemma 2. If P and Q are admissible paths, then PQ is admissible if and only if b_P and a_Q are of opposite sign.

Notation: Let $T = \langle a, b, c \rangle$ be a triple of reals. Then $T^{\sim}$ denotes the triple $\langle b, a, c \rangle$.

*Private Communication.

Lemma 3. If T_1, T_2 are triples, $T_1 * T_2 = (\tilde{T}_2 * \tilde{T})^{\sim}$.

Corollary 4. If Q is the reverse of an admissible path P, then $r_p = \tilde{r_Q}$.

Corollary 5. The reverse of an infeasible loop is itself infeasible.

Lemma 6. Any cyclic permutation of an infeasible loop is infeasible.

Notation: Where u, v, w are reals, let $u \overset{w}{<} v$ mean that $u < v$ if $w \geq 0$ and $u > v$ if $w < 0$.

Definition: Where P is an admissible path, the discriminant d_p of P is given by $\frac{c_p}{a_p + b_p}$

Note that an infeasible loop is one with discriminant $-\infty$.

Lemma 7. If PQ is an admissible loop from v_0 to v_0, then PQ is infeasible iff $d_p \overset{a_Q}{>} d_Q$ iff $d_p \overset{a_p}{<} d_Q$.

Notation: In the following, let $\langle a_1, b_1, c_1\rangle$, $\langle a_2, b_2, c_2\rangle$, $\langle a_3, b_3, c_3\rangle$, and $\langle a_p, b_p, c_p\rangle$, respectively, denote the residues of P_1, P_2, P_3, and P.

Lemma 8. If G is closed and has an infeasible loop from v_0 to v_0, G has an infeasible simple loop.

Pf. Let P be a shortest infeasible loop from v_0 to v_0 in G. If P is simple, we are done. Otherwise, since, by admissibility, the intermediate vertices of P are distinct from v_0, P can be expressed $P_1P_2P_3$, where P_2 is simple. We claim that P_2 is also infeasible.

Suppose not. Then either $a_2 + b_2 = 0$, and $c_2 \geq 0$, or d_{P_2} is finite. In the former case, a_2 and b_2 have opposite signs. It follows from Lemma 2 that b_1 and a_3 must as well, hence P_1P_3 is admissible. Now since

$$r_{P_1P_2} = \langle 0, b_1, c_1\rangle * \langle a_2, b_2, c_2\rangle$$
$$= \frac{a_2}{|a_2|} \langle 0, -b_1b_2, c_1a_2 - c_2b_1\rangle \quad ,$$

we have:

$$d_{P_1P_2} = \frac{c_1a_2 - c_2b_1}{-b_1b_2} = \frac{c_2}{b_2} - \quad \frac{a_2}{b_2} \quad d_{P_1}$$

$$= \frac{c_2}{b_2} + d_{P_1} \quad .$$

Since P is infeasible, we have from Lemma 7 that

$$\frac{c_2}{b_2} + d_{P_1} \overset{a_3}{>} d_{P_3} \quad .$$

Thus,

$$c_2 + b_2d_{P_1} \overset{a_3b_2}{>} b_2d_{P_3}$$

$\therefore$ $c_2 + b_2d_{P_1} < b_2d_{P_3}$ (since a_3 and b_2 have opposite signs)

$\therefore$ $b_2d_{P_1} < b_2d_{P_3}$ (since $c_2 \geq 0$)

$\therefore$ $d_{P_1} \overset{b_2}{<} d_{P_3}$

$\therefore$ $d_{P_1} \overset{a_3}{>} d_{P_3}$ (since b_2 and a_3 are of opposite sign).

But then P_1P_3 is infeasible by Lemma 7, contradicting our assumption that P is the shortest such loop.

Now if d_{P_2} is finite, the closedness of G provides that some vertex x on P_2 must be connected to v_0 via an edge E labeled $ax \le c$, where c/a is the discriminant of some cyclic permutation P_2' (possibly = P_2) of P_2. We now have three cases:

Case I. P_2 is not permutable.

Then $P_2' = P_2$, $a = a_2 + b_2$, $c = c_2$, and by Lemma 2, a_2 and b_2 are of the same sign. Also, a must be of this sign; hence both P_1E and EP_3 are admissible. An argument similar to the one above gives that one or the other of P_1E, EP_3 must be infeasible, contradicting the shortness of P.

Case II. P_2 is permutable and $P_2' = P_2$.

In this case, we have from Lemma 2 that a_2 and b_2 have opposite signs; hence b_1 and a_3 do as well. An argument similar to that given earlier shows that one of P_1P_3, P_1E, and EP_3 must be infeasible, again contradicting the shortness of P_2.

Case III. P_2 is permutable and $P_2' \neq P_2$.

Let P_4 be the initial subpath of P_2 which terminates at x, and let P_5 be the final subpath of P_2 which originates at x (so that $P_2 = P_4P_5$). In this case, it can be shown that P_1P_3 is admissible, that one of P_1P_4E, EP_5P_3 is admissible, and that one of these three paths must be infeasible. The shortness of P is thus once again contradicted.

Theorem. Let G be a closed graph for S. Then S is satisfiable if and only if G has no simple infeasible loop.

Pf. It follows from Lemma 1 that, if G has a simple, infeasible loop, S must be unsatisfiable. Conversely, Suppose G has no such loop. We will show that S is satisfiable by constructing a solution.

Let $v_1, \ldots, v_r$ be the variables of S other than v_0. We construct a sequence $\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_r$ of reals and a sequence $G_0, G_1, \ldots, G_r$ of graphs inductively as follows:

(1) Let $\hat{v}_0 = 0$ and $G_0 = G$.

(2) Suppose $\hat{v}_i$ and G_i have been determined for $0 \le i < j \le r$. Let $\sup_j = \min\{d_P | P$ is an admissible path from v_j to v_0 in G_{j-1} and $a_P > 0\}$. $\inf_j = \max\{d_P | P$ is an admissible path from v_0 to v_j in G_{j-1} and $b_P < 0\}$. (where it is understood that $\min\emptyset = \infty$ and $\max\emptyset = -\infty$). Then let $\hat{v}_j$ be any value in the interval $[\inf_j, \sup_j]$. (We show momentarily that $\inf_j \le \sup_j$.) Let G_j be obtained from G_{j-1} by adding two new edges from v_j to v_0, labeled $v_j \le \hat{v}_j$ and and $\hat{v}_j \ge \hat{v}_j$, respectively.

To ensure that the $\hat{v}_j$'s and G_j's are well defined, we must show that, for $1 \le j \le r$, $\inf_j \le \sup_j$. It will then remain to show that the $\hat{v}_j$'s do indeed give a solution for S.

We need the following claim:

Claim. (i) For $1 \le j \le r$, $\inf_j \le \sup_j$

(ii) For $0 \le j \le r$, G_j has no infeasible simple loops.

Pf. By induction on j.

Basis. $j = 0$.
In this case, (i) holds vacuously, and (ii) holds since $G_0 = G$.

Induction Step. $0 < j \le r$.

For (i), suppose, to the contrary, that $\inf_j > \sup_j$. Then in G_{j-1} admissible paths P_1, P_2 exist from v_0 to v_j and v_j to v_0, respectively, with $b_{P_1} < 0$, $a_{P_2} > 0$, and $d_{P_1} > d_{P_2}$. By Lemma 2, P_1P_2 is an admissible loop, and by Lemma 7, P_1P_2 is infeasible. By Lemma 8, then, G_{j-1} has a simple infeasible loop, contradicting (ii) of the induction hypothesis.

For (ii), suppose G_j has an infeasible simple loop P. Since G_{j-1} has no such loop, and since the loop formed by the two new edges added to G_{j-1} to obtain G_j is not infeasible, P (or its reverse) must be of the form P'E, where E is one of the two new edges (say the one labeled $v_j \le \hat{v}_j$; the other case is handled similarly), and P' is a path from v_0 to v_j in G_{j-1}. But then, by Lemma 7, $d_{P'} > d_E = \hat{v}_j$, contradicting $\hat{v}_j \ge \inf_j \ge d_{P'}$. (Note that $b_{P'} < 0$ from the admissibility of P'E.

Q.E.D.

It now remains to show that the $\hat{v}_j$'s satisfy S. So let $ax + by \le c$ be an inequality of S. We claim that $a\hat{x} + b\hat{y} \le c$. We treat the case in which $a > 0$ and $b < 0$; the other cases are argued similarly. Let E be the edge labeled $ax + by \le c$ in G_r. Then, where E_1 is the edge labeled $\hat{x} \le x$ in G_r, and E_2 is the one labeled $y \le \hat{y}$, E_1EE_2 forms an admissible loop. The residue of this loop is

$$\langle 0, -1, -\hat{x}\rangle * \langle a, b, c\rangle * \langle 1, 0, \hat{y}\rangle = \langle 0, 0, -a\hat{x} - b\hat{y} + c\rangle \quad .$$

Since, by the claim proved above, and by Lemma 8, G_r has no infeasible loops from v_0 to v_0, we have $-a\hat{x} - b\hat{y} + c \ge 0$. Thus $a\hat{x} + b\hat{y} \le c$ as required.

Q.E.D.

Acknowledgments

The author gratefully acknowledges the insights provided by R. Tarjan, R. Boyer, J Moore, M. W. Green, and W. Bibel.

REFERENCES

1. W. W. Bledsoe. Program correctness. The University of Texas at Austin Mathematics Department Memo ATP-14 (January 1974).

2. W. W. Bledsoe. The sup-inf method in Presburger arithmetic. The University of Texas at Austin Mathematics Department Memo ATP-18 (December 1974).

3. W. W. Bledsoe. A new method for proving certain Presburger formulas. Advance Papers Papers, 4th Int. Joint Conf. on Artif. Intell., 15-21, Tibilisi, Georgia U.S.S.R. (Septemver 1975).

4. W. W. Bledsoe, R. S. Boyer, and W. H. Henneman. Computer proofs of limit theorems. Artif. Intell. 3, 27-60 (1972).

5. W. W. Bledsoe and P. Bruell. A man-machine theorem-proving system. Artif. Intell. 5, 51-72 (1974).

6. D. C. Cooper. Programs for mechanical program verification. In Mach. Intell. 6, 43-59, American Elsevier, New York (1971).

7. G. B. Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, New Jersey (1962).

8. L. P. Deutch. An interactive program verifier. Ph.D. Thesis, University of California, Berkeley (1973).

9. B. Elspas, R. E. Boyer, R. Shostak, and J. Spitzen. A verification system for JOVIAL/J3 programs. SRI Technical Report 3756-1, Stanford Research Institute, Menlo Park, California (January 1976).

10. R. E. Gomory. An algorithm for integer solutions to linear programs. Princeton IBM Math. Res. Report (November 1958); also in R. L. Graves and P. Wolfe (eds.), Recent Advances in Mathematical Programming, 269-302, McGraw-Hill, New York (1963).

11. D. I. Good, R. L. London, and W. W. Bledsoe. An interactive system. Proc. 1975 Int. Conf. on Reliable Software, Los Angeles (April 1975).

12. S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification 1: A logical basis and its implementation. Stanford AI Memo 200 (May 1973) and USC Information Sciences Institute Rept. ISI/RR-73-11 (May 1973).

13. D. B. Johnson. Finding all the elementary circuits of a directed graph. SIAM J. Computing 4, 77-84 (1975).

14. J. King. A program verifier. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh (1969).

15. R. D. Lee. An application of mathematical logic to the integer linear programming problem. Notre Dame J. Formal Logic XIII, 2 (April 1972).

16. S. D. Litvintchouk and V. R. Pratt. A proof checker for dynamic logic. 5th Int. Joint Conference on Art. Int., 552-558, Boston (August 1977).

17. M. Prabhaker and N. Deo. On algorithms for enumerating all circuits of a graph. SIAM J. Computing 5, I. (March 1976.

18. V. R. Pratt. Two easy theories whose combination is hard. M.I.T. Technical Rept., Cambridge, Massachusetts (September 1977).

19. R. C. Read and R. E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. ERL Memo M 433, Electronics Research Laboratory, University of California, Berkeley (1973).

20. R. Shostak. An efficient decision procedure for arithmetic with function symbols. Presented at 5th Int. Joint Conf. on Art. Int., Cambridge, Massachusetts (August (1977), Also to appear in J. ACM.

21. R. Shostak. On the sup-inf method for proving Presburger formulas. J. ACM 24, 4, 529-543 (October 1977).

22. N. Suzuki. Verifying programs by algebraic and logical reduction. Proc. Int. Conf. on Reliable Software (Sigplan Notices) 10, 6 (June 1975).

23. J. L. Szwarcfiter and P. E. Lauer, Finding the elementary cycles of a directed graph in. O(n + m) per cycle, No. 60, University of Newcastle upon Tyne, Newcastle upon Tyne, England (May 1974).

24. R. Tarjan. Enumeration of the elementary circuits of a directed graph. SIAM J. Computing 2 (1973).

25. R. J. Waldinger and K. N. Levitt. Reasoning about Programs. J. Artif. Int. 5, 235-316 (1974).

A GLIMPSE OF TRUTH MAINTENANCE

Jon Doyle
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139

Abstract: To choose their actions, reasoning programs must be able to draw conclusions from limited information and subsequently revise their beliefs when discoveries invalidate previous assumptions. A truth maintenance system is a problem solver subsystem for performing these functions by recording and maintaining the reasons for program beliefs. These recorded reasons are useful in constructing explanations of program actions in "responsible" programs, and in guiding the course of action of a problem solver. This paper describes the structure of a truth maintenance system, methods for encoding control structures in patterns of reasons for beliefs, and the method of dependency-directed backtracking.

Acknowledgements: I thank Gerald Jay Sussman, Richard M. Stallman, Guy L. Steele Jr., Johan de Kleer, Drew McDermott, David McAllester, Scott Fahlman, Howard Shrobe, Charles Rich, Marilyn Matz, Beth Levin, Jim Stansfield, Mitchell Marcus, and Richard Brown for ideas, comments and advice, and the Fannie and John Hertz Foundation for supporting my research with a graduate fellowship. This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643, and in part by NSF grant MCS77-04828.

Introduction

One important problem faced by reasoning programs is the need to make decisions based on limited information. This problem arises both in programs interacting with an external environment and in contemplative programs searching a data base for an answer to some question. There are two consequences of this need to predict; the program must have some way to make decisions based on limited information, and the program must have some way to revise its beliefs if these decisions are found to be in error. The first of these abilities is provided by utilizing epistemic classifications of possible program beliefs so that conclusions may be drawn from the lack of belief as well as from other beliefs. The second ability is in general a very complex problem for which no complete solutions are known. (See Quine and Ullian [1978] and Rescher [1964] for surveys of the problem.) However, the simpler problem of revising beliefs based on limited information is solvable by recording the reasons for each program belief. These records can be used to find the set of extant beliefs by determining which beliefs have valid reasons. These recorded reasons also are useful in resolving conflicts that arise when limited knowledge gives rise to incompatible conclusions. This paper describes a mechanization of these abilities, embodied in a general-purpose problem solver subsystem called a *truth maintenance system*.

A truth maintenance system (TMS) records and maintains "proofs" of program beliefs. It manipulates two data structures; *nodes*, which represent beliefs, and *justifications*, which represent reasons for beliefs. The fundamental actions the TMS can be called upon to perform are the creation of a new node, to which the problem solving program can attach the statement of a belief, and the addition of a new justification to a node, to represent assertion of the belief associated with the node by some rule or procedure in the problem solver. The addition of new justifications may invoke the automatic procedure of *truth maintenance* to make any revisions necessary in the set of beliefs. The TMS revises beliefs by using the recorded justifications to compute non-circular proofs of beliefs from basic hypotheses. These proofs distinguish one or more justifications as the *well-founded support* for each believed node, and are used during truth maintenance to determine the set of beliefs to update by finding those nodes whose well-founded support depends on changed beliefs. These proofs allow another process, *dependency-directed backtracking*, to resolve conflicts by tracing the well-founded supports of conflicting beliefs to remove one of the assumptions causing the conflict and to make a record used to prevents similar future conflicts.

The TMS employs a special type of justification, called a *non-monotonic justification*, to draw conclusions based on limited or incomplete knowledge. This type of justification allows belief in a node to be based not only on other beliefs, as occurs in the standard forms of deduction and reasoning, but also on lack of belief in certain nodes. For example, a node N-1 representing a statement P might be justified on the basis of a lack of belief in a node N-2 representing the belief ~P. (Distinct nodes are used to represent P and ~P.) In this case, the TMS would have N-1 believed as long as N-2 was not believed, and we would call N-1 an *assumption*. (More generally, by assumption we mean any node whose well-founded support is a non-monotonic justification.)

As a small example, suppose an office scheduling program is considering holding a meeting M on Wednesday. To do this, the program assumes that the meeting is on Wednesday. The data base of the program includes a rule which draws the conclusion that due to regular commitments, any meeting on Wednesday must occur at 1:00 PM. However, the fragment of the schedule for the week constructed so far has something else scheduled for that time already, and so another rule in the data base concludes that the day for the meeting cannot be Wednesday. These beliefs might be notated as follows:

Node	*Statement*	*Justification*
N-1	DAY(M) = WEDNESDAY	(SL () (N-2))
N-2	DAY(M) ≠ WEDNESDAY	
N-3	TIME(M) = 13:00	(SL (R-37 N-1) ())

As seen in the above notation for justifications, each justification consists of two lists. The meaning of the notation is that the statement depends on each of the nodes in the first list being believed, and on each of the nodes in the second list not being believed. Since there is no known justification for N-2, it is not believed. The justification for N-1 specifies that it depends on the lack of belief in N-2, and so N-1 is believed. The justification for N-3 shows that it is believed due to rule R-37 acting on N-1. When the assumption N-1 is rejected by some rule,

N-2	DAY(M) ≠ WEDNESDAY	(SL (R-9 N-7 N-8) ())

where N-7 and N-8 represent the day and time of some other engagement, the TMS will revise the beliefs so that N-1 and N-3 are not believed.

Representation of Knowledge about Belief

A node may have several justifications, each of which represents a different reason for belief in the node. The node is believed if at least one of these justifications is *valid*. The conditions for validity of justifications are described below. We say that a node which has a valid justification is *in*, and that a node without a valid justification is *out*. The distinction between *in* and *out* is not that of *true* and *false*. The former classification denotes conditions of knowledge about reasons for belief; the existence or non-existence of valid reasons. *True* and *false*, on the other hand, classify statements according to truth value independent of any reasons for belief. In this way, there can be four states of knowledge about a proposition P, corresponding to the node representing P being *in* or *out* and the node representing ~P being *in* or *out*.

There are two basic forms of justifications. These are inspired by the typical forms of arguments in natural deduction inference systems. A sample proof in such a system might be as follows:

Line	*Statement*	*Justification*	*Dependencies*
1.	A⊃B	Premise	{1}
2.	B⊃C	Premise	{2}
3.	A	Hypothesis	{3}
4.	B	MP 1,3	{1,3}
5.	C	MP 2,4	{1,2,3}
6.	A⊃C	Discharge 3,5	{1,2}

Each step of the proof has a line number, a statement, a justification, and the set of line numbers the statement depends on. Premises and hypotheses depend on themselves, and other lines depend on the set of premises and hypotheses derived from their justifications. The above proof proves A⊃C from the premises A⊃B and B⊃C by hypothesizing A and concluding C. The assumption A is then discharged to provide the proof of A⊃C. There are two effects that justifications can have on the set of dependencies in natural deduction systems; either the justifications can sum the dependencies of the referenced lines (as in line 4), or they can subtract the dependencies of some lines from those of other lines (as in line 6). The two types of justifications used in a TMS account for these effects on dependencies. A *support-list (SL) justification* says that the justified node depends on a set of other nodes, and thus in effect sums the dependencies of the referenced nodes. A *conditional-proof (CP) justification* says that the node it justifies depends on the validity of a certain hypothetical argument, and as in the example above, subtracts the dependencies of some nodes (the hypotheses of the hypothetical argument) from the dependencies of others (the conclusion of the hypothetical argument). These two types of justifications can be used to construct a variety of forms of dependency relationships.

The support-list justification is of the form

(SL <*in*list> <*out*list>).

A SL-justification is valid if each node in its *in*list is *in*, and each node in its *out*list is *out*. A SL-justification can be used to represent several types of deductions. When both the *in*list and *out*list are empty, we say the justification forms a *premise* justification. A premise justification is always valid, and so the node it justifies will always be believed. Normal deductions are represented by support-list justifications with empty *out*lists. These represent monotonic deductions of the justified node from belief in the nodes of the *in*list. *Assumptions* are nodes whose well-founded support is a support-list justification with a nonempty *out*list. These justifications can be interpreted by viewing the nodes of the *in*list as comprising the reasons for making the assumption; the nodes of the *out*list represent the specific incompletenesses of knowledge authorizing the assumption.

The conditional-proof justification takes the form

(CP <consequent> <*in*hypotheses> <*out*hypotheses>).

A node justified by such a justification represents an implication, whose support is derived by a conditional proof of the consequent node from the hypothesis nodes. A justification of this form is valid if the consequent node is *in* whenever (a) each node of the *in*hypotheses is *in* and (b)each node of the *out*hypotheses is *out*. Except in a few esoteric uses, the set of *out*hypotheses is empty. Standard conditional proofs in natural deduction systems specify a single set of hypotheses, which correspond to our *in*hypotheses. The truth maintenance system requires that the set of hypotheses be divided into two disjoint subsets, since nodes may be derived both from some nodes being *in* and other nodes being *out*.

Encoding Control Structures in Sets of Justifications

The use of justifications for recording normal, monotonic deductions is straightforward. Non-monotonic justifications augment the standard deductive relationships between beliefs by allowing the encoding of control structures (patterns of assumptions) into sets of justifications among nodes. The virtue of such an encoding is that the choices underlying problem solver actions become explicit, thus allowing careful failure analysis. In some cases, an automatic procedure like dependency-directed backtracking can perform this failure analysis and set the problem solver onto the next step of its investigation. We describe in detail two important types of control structures; default assumptions and sets of alternatives.

Default Assumptions

One very common technique used in problem solving systems is to specify a default choice for the value of some quantity. This choice is made with the intent of overriding it if either a good reason is found for using some other value, or if making the default choice leads to an inconsistency. The assumption of the day of the week for a meeting in the first example above is such a default assumption.

In the case of a binary choice, a default assumption can be represented by believing a node if the node representing its negation is *out*. When the default is chosen from a set of alternatives, the following generalization of the binary case is used. Let $\{F_1, ..., F_n\}$ be the set of the nodes which represent each of the possible values of the choice. Let G be a node which represents the reason for making the default assumption. Then F_i may be made the default choice by providing it with the justification

(SL (G) $(F_1 \dots F_{i-1}\ F_{i+1} \dots F_n)$).

If no information about the choice exists, there will be no reasons for believing any of the alternatives except F_i. Thus F_i will be *in* and each of the other alternatives will be *out*. If some other alternative receives a valid justification from other sources, that alternative will become *in*. This will invalidate the support of F_i, and F_i will become *out*. If a contradiction is derived from F_i, the dependency-directed backtracking mechanism will recognize that F_i is an assumption by means of its dependence on the other alternatives being *out*. (See the section on dependency-directed backtracking for an explanation of this.) The result of backtracking may be to justify one of the other alternatives, say F_j, causing F_i to go *out*. The justification for F_j will be of the form

(SL <various things> <remainders>)

where the remainders are the F_k's remaining after F_i and F_j are taken away. In effect, backtracking will cause the removal of the default choice with the set of alternatives, and will set up a new default assumption structure from the remaining alternatives. As a concrete example, our scheduling program might default a meeting day as follows:

```
N-1     DAY(M) = MONDAY
N-2     DAY(M) = WEDNESDAY         (SL () (N-1 N-3))
N-3     DAY(M) = FRIDAY
```

In this example, Wednesday is assumed to be the day of the meeting M, with Monday and Friday being the alternatives. Wednesday will be the default choice until a valid reason is supplied for either Monday or Friday.

If the complete set of alternatives from which the default assumption is to be chosen cannot be known in advance but must be discovered piecemeal, a slightly different structure is necessary. This ability to extend the set of alternatives is necessary, for example, when the default is a number, due to the large set of possible alternatives. For cases like this the following structure may be used instead. Retaining the above notation, let $\sim F_i$ be a new node which will represent the negation of F_i. We will arrange for F_i to be believed if $\sim F_i$ cannot be proven, and will set up justifications so that if F_j is distinct from F_i, F_j will imply $\sim F_i$. This is done by giving F_i the justification

(SL (G) $(\sim F_i)$),

and by giving $\sim F_i$ a justification of the form

(SL (F_j) ())

for each alternative F_j distinct from F_i. As before, F_i will be assumed if no reasons for using any other alternative exist. Furthermore, new alternatives can be added to the set simply by giving $\sim F_i$ a new justification corresponding to the new alternative. This structure for default assumptions will behave as did the fixed structure in the case of an unselected alternative receiving independent support. Backtracking, however, has a different effect. If a contradiction is derived from the default assumption supported by the extensible structure, $\sim F_i$ will be justified so as to make F_i become *out*. If this happens, no alternative will be selected to take the place of the default assumption. The extensible structure requires an external mechanism to construct a new default assumption whenever the current default is ruled out. For example, a census program might make assumptions about the number of children in a family as follows:

```
N-1     #-CHILDREN(F) = 2          (SL () (N-2))
N-2     #-CHILDREN(F) ≠ 2          (SL (N-3) ())
                                   (SL (N-4) ())
                                   (SL (N-5) ())
                                   (SL (N-6) ())
N-3     #-CHILDREN(F) = 0
N-4     #-CHILDREN(F) = 1
N-5     #-CHILDREN(F) = 3
N-6     #-CHILDREN(F) = 4
```

With this system of justifications, N-1 would be believed because no different number of children is known. If it turns out that the family has 5 children, a new statement would have to be made, along with a new justification of N-2 in terms of this new statement.

Sets of Alternatives

The default assumption structures allow a choice from a set of alternatives, but do not specify the order in which new alternatives are to be tried if the initial choice is wrong. Such advice sometimes is a linear ordering on the set of alternatives. Linearly ordered sets of alternatives are useful whenever heuristic information is available for making a choice such as the state of a transistor or the day of the week for a meeting.

If it is certain that rejected alternatives are rejected permanently and will never again be believed, the linear ordering on the set of alternatives can be specified by a controlled sequence of default assumptions. This can be implemented in a ladder-like structure of justifications by justifying each F_i with

(SL $(G\ \sim F_{i-1})$ $(\sim F_i)$),

where G is the reason for the set of alternatives. The first alternative F_1 will be selected initially. As each alternatives is ruled out through its negation being justified, the next alternative in the list will be assumed. For example, we might have:

```
N-1     DAY(M) = WEDNESDAY      (SL () (N-2))
N-2     DAY(M) ≠ WEDNESDAY
N-3     DAY(M) = THURSDAY       (SL (N-2) (N-4))
N-4     DAY(M) ≠ THURSDAY
N-5     DAY(M) = TUESDAY        (SL (N-4) ())
```

This would guide the choice of day for the meeting M to Wednesday, Thursday and Tuesday, in that order.

If previously rejected alternatives can be independently rejustified (say by special case rules correcting a choice made by the backtracking system), a more complicated structure is necessary. Such a set of alternatives can be described by the following justifications. For each alternative A_i, three new nodes should be created. These new nodes are PA_i (meaning "A_i is a possible alternative"), NSA_i (meaning "A_i is not the selected alternative"), and ROA_i (meaning "A_i is a ruled-out alternative"). Each PA_i should be justified with the reason for including A_i in the set of alternatives. Each ROA_i is left unjustified. Each A_i and NSA_i should be given justifications as follows:

A_i:	(SL $(PA_i\ NSA_1 \ldots NSA_{i-1})$ (ROA_i))
NSA_i:	(SL () (PA_i))
	(SL (ROA_i) ())

Here the justification for A_i means that A_i is an alternative, no better alternative is selected, and A_i is not ruled out. The justification for NSA_i means that either A_i is not a valid alternative, or A_i is ruled out. With this structure, processes can independently rule in or rule out an alternative by justifying the appropriate alternative node or ruled-out-alternative node.

This structure is also extensible. New alternatives may be added simply by constructing the appropriate justifications as above. These additions are restricted to appearing at the end of the order. That is, new alternatives cannot be spliced into the linear order between two previously inserted alternatives.

Dependency-Directed Backtracking

Making assumptions admits the possibility of making errors. When a contradiction or other inconsistent state of the data base occurs, the TMS employs a process called dependency-directed backtracking to find and remove incorrect assumptions so as to restore consistency. There are several steps involved in dependency-directed backtracking, but first the inconsistency must somehow be signalled to the TMS, as there is no built-in notion of inconsistency. This signalling consists of informing the TMS that a node represents an inconsistency. With this knowledge, the TMS will try to restore consistency whenever the node comes *in* by rejecting enough assumptions to force the node *out*. Any node may be marked for such treatment. A node so marked is called a *contradiction*.

The steps of dependency-directed backtracking are as follows. First, the well-founded support of the contradiction node is traced backwards to find the set of assumptions (nodes with a non-monotonic justification as their well-founded support) underlying the contradiction. Belief in at least one of these assumptions must be retracted to remove the contradiction. This is done by creating a new justification for one of the *out* nodes underlying one of the assumptions. Since the backtracker may be mistaken in its assignment of blame to that assumption, the justification used to retract the assumption must indicate the alternatives that were available but not utilized by the backtracker. Thus the new justification includes (a) the reason why the contradiction occurred and (b) the other assumptions involved. Thus the second step of the backtracking process is to construct a node recording the reason why the contradiction occurred, and the third step is to use this node and the other assumptions in justifying an *out* node supporting the assumption selected for removal.

In more detail, the first step of the backtracking process traces backwards through the well-founded support of the contradiction node to collect the set of "maximal" assumptions supporting the contradiction. Not all assumptions found by tracing the well-founded support are used; instead, only those assumptions which do not support other assumptions underlying the contradiction as well. That is, the well-founded support relationships induce a natural partial-ordering on nodes, where one node is said to be "lower" than a second node if the first occurs in the second's well-founded support. The maximal assumptions are then those assumptions which are maximal in this partial order. Only this "front line" of assumptions is used because if the reason for revoking a lower-level assumption involves a higher-level assumption, then the removal of the lower-level assumption would cause truth maintenance to remove the higher-level assumption that it supports, so the reason for removing the lower-level assumption would not hold up. This reflects the fact that there may not be enough information to definitely rule out a lower-level assumption.

The second step summarizes the reason for the contradiction in terms of the set of selected assumptions. Let $S = \{A_1, \ldots, A_n\}$ indicate the set of inconsistent assumptions. The backtracker then creates a node called a *nogood*, a new node signifying that S is inconsistent. Since contradiction nodes really represent the false statement, the nogood node can be taken to represent

$$A_1 \wedge \ldots \wedge A_n \supset \mathit{false},$$

or alternatively,

(1) $$\sim(A_1 \wedge \ldots \wedge A_n).$$

S is recorded as the *nogood-set* of the nogood. This meaning for the nogood node is produced by justifying it with the conditional proof of the contradiction node relative to the assumption nodes, that is, with the justification

(2) (CP <contradiction node> S ()).

In this way, the inconsistency of the set of assumptions will be remembered even after the contradiction has been resolved by the retraction of some hypothesis.

The final step is selecting an assumption A_i (the "culprit") from S and justifying one of the *out* nodes listed in its well-founded supporting justification. (If these underlying *out* nodes are thought of as "denials" of the assumption, then this step is much like reasoning by *reductio ad absurdum*.) Let NG be the nogood, and let the inconsistent assumptions be

$A_1, \ldots, A_n$. Let $D_1, \ldots, D_k$ be the *out* nodes appearing in the justification which supports belief in the assumption A_i. This justification for the assumption can be invalidated by justifying D_1 with the justification

(3) (SL $(NG\ A_1 \ldots A_{i-1}\, A_{i+1} \ldots A_n)$ $(D_2 \ldots D_k)$).

This justification is valid whenever the nogood and other assumptions are believed and the other "denials" of the culprit are not believed. If the choice of culprit was in error, then another contradiction will occur in the future involving D_1, and by this justification will be led to suspect the remaining assumptions, as well as D_1 if there are any other *out* nodes listed in its justification. If, by means of other previously existing justifications, the current contradiction is still *in* following the addition of this justification, backtracking is repeated. Presumably the new invocation of the backtracking process will find that the previous culprit is no longer an assumption. Backtracking halts when the contradiction becomes *out*, or when no assumptions can be found underlying the contradiction.

As an example, consider a program scheduling a meeting, preferably at 10 AM in either room 813 or 801. This might be represented as:

```
N-1     TIME(M) = 1000        (SL () (N-2))
N-2     TIME(M) ≠ 1000
N-3     ROOM(M) = 813         (SL () (N-4))
N-4     ROOM(M) = 801
```

With these justifications, N-1 and N-3 are *in*, and the other two nodes are *out*. If some previously scheduled meeting exists, it might cause this combination of time and room for the meeting to be ruled out by means of a contradiction.

```
N-5     CONTRADICTION         (SL (N-1 N-3) ())
```

The dependency-directed backtracking system then traces the well-founded support of the contradiction to find that it depends on two assumptions, N-1 and N-3, both of which are maximal.

```
N-6     NOGOOD N-1 N-3        (CP N-5 (N-1 N-3) ())
N-4     ROOM(M) = 801         (SL (N-6 N-1) ())
```

A nogood node is created which means, in accordance with form (1) above,

~(TIME(M) = 1000 ∧ ROOM(M) = 813)

and this nogood is given a justification corresponding to form (2) above. The assumption N-3 is selected arbitrarily as the culprit, and is rejected by providing its only *out* supporting node, N-4, with a justification of the form (3) above. Following this, N-1, N-4, and N-6 are *in*, and N-2, N-3, and N-5 are *out*. N-6, the nogood node, has an always-valid CP-justification since the contradiction node N-5 depends directly on the two assumptions N-1 and N-3 without any additional beliefs intervening. If some further consideration determines that room 801 cannot be used after all, another contradiction node could be created to force a different choice.

```
N-7     CONTRADICTION         (SL (N-4) ())
N-8     NOGOOD N-1            (CP N-7 (N-1) ())
N-2     TIME(M) ≠ 1000        (SL (N-8) ())
```

Tracing backwards from N-7 through N-4, N-6, and N-1, the backtracker finds that the contradiction depends on only one assumption, N-1. The nogood node N-8 is created and justified with a CP-justification which in effect is equivalent to the SL-justification

(SL (N-6) ()),

since the nogood N-6 contributes to the contradiction but does not itself depend on the assumption N-1. The revocation of the assumption N-1 removes N-5, the previous objection to the choice of room, so at the close of this bit of decision making N-2, N-3, N-6, and N-8 are *in*, and N-1, N-4, N-5, and N-7 are *out*.

There are a number of variations on this particular scheme for dependency-directed backtracking. All of these variations are great improvements over the chronological backtracking systems used in classical systems like MICRO-PLANNER and many early theorem provers. The improvements stem from the non-chronological nature of dependency-directed backtracking, in which the support relationships rather than the temporal orderings determine the choices responsible for an error. Another improvement is that the cause of the contradiction is summarized via a nogood node. This summarization keeps the system from making the same mistake in the future. Stallman and Sussman [1977] have shown that these two improvements lead to enormous gains in efficiency.

Truth Maintenance Mechanisms

Consider the statements:

F	(= (+ X Y) 4)
G	(= X 1)
H	(= Y 3).

If both *F* and *G* are *in*, then belief in *H* can be justified by

(SL (*F G*) ()).

This justification will cause *H* to become *in*. If *G* subsequently becomes *out* due to changing hypotheses, and if *H* becomes *in* by some other justification, then *G* can be justified by

(SL (*F H*) ()).

Suppose the justification supporting belief in *H* then becomes invalid, thus causing the TMS to reassess the grounds for belief in *H*. If the decision to believe a node is based on a simple evaluation of each of the justifications of the node, then both *G* and *H* will be left *in*, since the two justifications form circular proofs for *G* and *H* in terms of each other. These justifications are mutually satisfactory if *F*, *G* and *H* are *in*. This example points out one of the major concerns in truth maintenance processing: the avoidance of using circular proofs to support beliefs. This is the reason why well-founded support is maintained.

Essentially three different kinds of circularities can arise in purported proofs. The first and most common is a circularity in which all nodes involved can, consistently with their justifications, be taken to be *out*. Such circularities arise routinely through equivalences and simultaneous constraints, in which many beliefs may be mutually supporting without any of the beliefs having non-circular reasons for being believed.

The above algebra example falls into this class of circularity.

The second type of circularity is one in which at least one of the nodes involved must be *in*. An example is that of two nodes F and G, such that F has the justification

(SL () (G)),

and G has the justification

(SL () (F)).

Here either F must be *in* and G *out*, or G must be *in* and F *out*. This type of circularity arises in defining some sets of alternatives. Frequently other ordered alternative structures can be used to avoid such circularities.

The third form of circularity which can arise is the unsatisfiable circularity. In this type of circularity, no assignment of *in* or *out* to nodes is consistent with their justifications. An example of such a circularity is a node F with the justification

(SL () (F)).

This justification implies that F is *in* if and only if F is *out*. Unsatisfiable circularities are bugs, indicating a misorganization of the knowledge of the program using the truth maintenance system. Unsatisfiable circularities are violations of the semantics of *in* and *out*, which can be interpreted as meaning that the lack of reasons for belief in a node is equivalent to the existence of reasons for belief in the node. (It has been my experience that such circularities are most commonly caused by confusing the concepts of *in* and *out* with those of *true* and *false*. For instance, the above example could be produced by this misinterpretation as an attempt to assume belief in the node F by giving it the justification (SL () (F)).)

In addition to the problems caused by circular proofs, the TMS must also handle problems introduced by conditional-proof justifications. There are two parts to the implemented approach. The validity of CP-justifications is checkable only in case the consequent and *in*hypotheses are *in* and the *out*hypotheses are *out*. This is a rare circumstance, however, particularly in the case of backtracking when a nogood node justified with a CP-justification is used to force *out* the contradiction node appearing as the consequent of the CP-justification. The TMS thus takes the opportunistic and incomplete strategy of using CP-justifications to compute SL-justifications which are equivalent in terms of the dependencies they specify, but which can be checked for validity at any time. Specifically, whenever the CP-justification is found to be valid, an equivalent SL-justification is computed by tracing through the well-founded support of the consequent node of the CP-justification to find the "front line" of nodes which are not in turn supported by any of the *in* or *out*hypotheses. This set of nodes can be divided into the *in* nodes, which form the *in*list of the equivalent SL-justification, and the *out* nodes, which form the *out*list of the equivalent SL-justification. The way these sets of nodes are computed from the well-founded support of the consequent of the CP-justification ensures that the consequent will be in whenever the *in*hypotheses are *in*, the *out*hypotheses are *out*, and the nodes of the equivalent SL-justification respectively *in* and *out*.

The details of the truth maintenance mechanisms will not be pursued here. Many details, along with an annotated implementation, are presented in [Doyle 1978].

Discussion

Truth maintenance systems solve part of the belief revision problem, and provide an associated mechanism for making assumptions based on limited information. It has long been recognized that making assumptions is a necessary part of AI systems, and many systems have employed some mechanism for this purpose. (For example, [Bobrow and Winograd 1977, de Kleer et al. 1977, 1978, Hayes 1973, 1977, McCarthy 1977, McCarthy and Hayes 1969, McDermott 1974, Minsky 1962, 1975, Reiter 1978, Roberts and Goldstein 1977, Sandewall 1972, Sussman et al. 1971].) Unfortunately, the related problem of belief revision received somewhat less study. Most work on revising beliefs was done in the framework of backtracking algorithms operating on rather simple systems of states and actions. The more general problem of revising beliefs based on records of deductions has only been examined in more recent work. (See [Cox and Pietrzykowski 1976, Doyle 1978, Fikes 1975, Hayes 1975, Katz and Manna 1976, Latombe 1977, London 1978, McAllester 1978, McDermott 1974, 1977, Nevins 1974, Srinivasan 1976, Stallman and Sussman 1977].) The literature of philosophy and logic contains a large amount of work on the belief revision problem (see [Quine and Ullian 1978, Rescher 1964]), as well as some work on formal methods for making decisions based on limited information. The history of attempts at formalizing the AI methods for making assumptions is surveyed by McDermott and Doyle [1978], who also present a mathematical semantics for what is termed *non-monotonic logic*.

Truth maintenance systems lend themselves to other uses as well as belief revision and making assumptions. Generating explanations is an immediate application. The recorded reasons for beliefs can form the basis of an explanation system in "responsible" programs [Sussman, personal communication] which can justify their actions and beliefs to a user. A crucial aspect of the problem of explanation is that unless care is taken in structuring the knowledge of the program, the explanations will contain information at many levels of detail, thus making the explanation incomprehensible. It is thus important to try to structure the use of a truth maintenance system so that levels of detail in explanations are separated automatically. Doyle [1978] describes a method by which conditional proofs are used to factor unwanted low-level details from explanations. When such factoring is done at each level, a hierarchical structure emerges in explanations.

Truth maintenance systems can be applied to the problem of controlling problem solvers in several ways. The simplest method is that of using an automatic procedure like dependency-directed backtracking for guiding the search. More sophisticated methods can be designed which represent control decisions as explicit program beliefs, and separate the reasons for control decisions from the reasons for beliefs derived in response to the control decisions. With such a separation, careful failure and choice analysis routines can examine the history of the problem solver, and much information can be salvaged from mistakes. (See [de Kleer et al. 1977, Doyle forthcoming, Stallman and Sussman 1977].)

References

D. G. Bobrow and T. Winograd, "An Overview of KRL, a Knowledge Representation Language," *Cognitive Science*, Vol. 1, No. 1, 1977.

P. T. Cox and T. Pietrzykowski, "A Graphical Deduction System," Department of Computer Science Research Report CS-75-35, University of Waterloo, July 1976.

J. de Kleer, J. Doyle, C. Rich, G. L. Steele Jr., and G. J. Sussman, "AMORD: A Deductive Procedure System," MIT AI Lab, Memo 435, January 1978.

J. de Kleer, J. Doyle, G. L. Steele Jr., and G. J. Sussman, "Explicit Control of Reasoning," MIT AI Lab, Memo 427, June 1977.

J. Doyle, "Truth Maintenance Systems for Problem Solving," MIT AI Lab, TR-419, January 1978.

J. Doyle, "Reflexive Interpreters", forthcoming.

R. E. Fikes, "Deductive Retrieval Mechanisms for State Description Models," *IJCAI4*, September 1975, pp. 99-106.

P. J. Hayes, "The Frame Problem and Related Problems in Artificial Intelligence," in A. Elithorn and D. Jones, editors, *Artificial and Human Thinking*, San Francisco: Josey-Bass, 1973.

P. J. Hayes, "The Logic of Frames", University of Essex, November 1977.

P. J. Hayes, "A Representation for Robot Plans," *IJCAI4*, September 1975, pp. 181-188.

S. Katz and Z. Manna, "Logical Analysis of Programs," *Comm. ACM*, Vol. 19, No. 4, pp. 188-206.

J.-C. Latombe, "Une Application de l'intelligence Artificielle a la Conception Assistee par Ordinateur (TROPIC)," Universite Scientifique et Medicale de Grenoble, thesis D.Sc. Mathematiques, November 1977.

P. E. London, "A Dependency-Based Modelling Mechanism for Problem Solving," Computer Science Department TR-589, University of Maryland, November 1977.

D. A. McAllester, "A Three-Valued Truth Maintenance System", MIT AI Lab, Memo 473, May 1978.

J. McCarthy and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in B. Meltzer and D. Michie, *Machine Intelligence 4*, New York: American Elsevier 1969, pp. 463-502.

J. McCarthy, "Epistemological Problems of Artificial Intelligence," *Proc. Fifth International Joint Conference on Artificial Intelligence*, pp. 1038-1044, 1977.

D. McDermott, "Assimilation of New Information by a Natural Language-Understanding System," MIT AI Lab, AI-TR-291, February 1974.

D. McDermott, "Flexibility and Efficiency in a Computer Program for Designing Circuits," MIT AI Lab, TR-402, June 1977.

D. McDermott and J. Doyle, "Non-Monotonic Logic I", MIT AI Lab, Memo 486, August 1978.

M. Minsky, "Problems of Formulation for Artificial Intelligence," *Proc. Symp. on Mathematical Problems in Biology*, American Mathematical Society, Providence, RI, 1962, pp. 35-46.

M. Minsky, "A Framework for Representing Knowledge," in *The Psychology of Computer Vision*, P. H. Winston, ed., New York: McGraw-Hill 1975, pp. 211-277.

A. J. Nevins, "A Human-Oriented Logic for Automatic Theorem Proving," *J. ACM 21*, #4 (October 1974), pp. 606-621.

W. V. Quine and J. S. Ullian, *The Web of Belief*, second edition, New York: Random House, 1978.

R. Reiter, "On Reasoning by Default," *Proc. Second Symp. on Theoretical Issues in Natural Language Processing*, Urbana, Illinois, August 1978.

N. Rescher, *Hypothetical Reasoning*, Amsterdam: North Holland 1964.

R. B. Roberts and I. P. Goldstein, "The FRL Manual," MIT AI Lab AI Memo 409, September 1977.

E. Sandewall, "An Approach to the Frame Problem, and its Implementation," *Machine Intelligence 7*, pp. 195-204, 1972.

C. V. Srinivasan, "The Architecture of Coherent Information System: A General Problem Solving System," *IEEE Transactions on Computers*, Vol. C-25, No. 4, April 1976, pp. 390-402.

R. M. Stallman and G. J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," *Artificial Intelligence*, Vol. 9, No. 2, (October 1977), pp. 135-196.

G. J. Sussman, T. Winograd and E. Charniak, "MICRO-PLANNER Reference Manual," MIT AI Lab, AI Memo 203a, December 1971.

Explicit Control of Reasoning In The Programmer's Apprentice

by
Howard Elliot Shrobe
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139

Abstract

This paper describes a reasoning system called REASON which is used as part of a larger program understanding system. REASON's key features are: (1) Its use of the Truth Maintenance System [Doyle, 1977] to record and manipulate the justifications for its deductions, (2) It use of non-monotonic logic [Doyle and McDermott, 1978] and (3) A discipline of explicitly recording control information [DeKleer, et. al., 1977] in a form which may be manipulated by the reasoning system itself. In this paper we present the basic formalisms of REASON and a description of how these can be used to build more flexible versions of well known problem solving mechanisms such as the contexts of QA4 [Rulifson, et. al., 1972] or Conniver [McDermott, 1972].

Introduction

REASON is a deductive system used as the reasoning component of the programmer's apprentice system [Rich and Shrobe, 76]. It is characterized by it's use of the Truth Maintainence System [Doyle, 77] to maintain a network of logical dependencies.

REASON is part of a "program understander"; it not only has the ability to show that a program is "correct" but it can also explain why the program is correct, showing how the intended behavior of the whole program results from the known (or assumed) behaviors of its parts. A second and stronger feature of REASON's design is that it is able to analyze the effect of a proposed change to a program, viewing the modification as a perturbation, rather than as a whole new program requiring a new proof.

Finally, REASON is intended to be an Advice Taker [McCarthy, 1968] which can be given new strategies for efficient deduction. To be effective such strategies must be capable of reacting not only to the facts currently believed to be true, but also to the lack of belief in other facts. Such systems must understand advice such as "while doing x don't use type y rules" which refer to the state of the problem solver itself [McDermott, 1977].

These criteria have led to a system in which not only facts but also the reasons for their belief are explicitly recorded. In addition, all goal states and the reasons for their existence are recorded within the same formalism.

This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643, and in part by NSF grant MCS77-04828.

The Basic Formalism: Facts, Rules, and Justifications

REASON is implemented in a variant of the language AMORD [DeKleer, et. al. 1977] we begin by reviewing the basic concepts of this language.

Since REASON's goal is not only to prove properties of a program but to understand how these properties follow from known or assumed properties of sub-modules, *justifications* are a crucial form of information in REASON. When an assertion is entered into REASON's data base, it is always accompanied by a justification explaining why the new assertion is believed. To make this convenient, each assertion is assigned a unique "fact-name". For example:

```
Assertion        System-Supplied-Fact-Name

(< X Y)                  F-1
(< Y Z)                  F-2
```

When REASON applies the transitivity rule to F-1 and F-2, it enters the newly deduced fact using the ASSERT function:

```
(Assert (< X Z)  (Transitivity F-1 F-2))
```

ASSERT takes two arguments: the new assertion to be added to the data base and the justification for the assertion; a justification is a list whose first element is a justification name and whose remaining elements are the fact-names upon which the new assertion depends.

One important justification is PREMISE; premise justifications involve no supporting facts. A premise is a fact which is believed unconditionally; belief in a premise does not depend on the truth value of any other fact. The three facts above might have been entered into the system as follows:

```
User Types                              System Supplied Fact-Name

(Assert (< X Y) (Premise))                      F-1
(Assert (< Y Z) (Premise))                      F-2
(Assert (< X Z) (Transitivity F-1 F-2))         F-3
```

REASON makes deductions using *rules* which consist of two elements: a *trigger set* and a *body*. The trigger set is a list of patterns each of which has two parts: a fact-name variable and an assertion pattern. The body is a LISP expression which is evaluated in an environment in which the variables of the patterns are bound to the objects which they matched. The following is a fairly typical REASON rule:

```
(rule ((:f (Rest :list-1 :list-2))
       (:g (Member :list-2 :obj-1)))
      (assert (member :list-1 :obj-1)(List-Membership :f :g)))
```

Variables are indicated by a leading colon (:), the body is the assert statement, and the trigger set is the list:

```
((:f (Rest :list-1 :list-2))
 (:g (Member :list-2 :obj-1)))
```

In these triggers, the leading single variable (:f or :g) is the fact-name variable, the remaining part of each trigger (Rest :list-1 :list-2) or (Member :list-2 :obj-1)) is the assertion pattern. A rule is applicable if all its patterns are matched by facts which have currently valid justifications (see below). The body is then executed in the environment of the match.

As each trigger is matched to an assertion, the fact-name variable of that trigger is bound to the fact-name of the matched assertion. This allows the body of the rule to refer to its triggering facts. In particular, assert statements in the body of the rule may include a justification mentioning these facts.

Assertions have one of two statuses in REASON, they are either *in* or *out*. A fact which is *in* is believed to be true. An assertion whose negation is *in* is believed to be false. If both an assertion and its negation are *in* then the data base is contradictory and corrective action is required. If neither the assertion nor its negation is *in*, then the fact is simply unknown.

assertion	negated assertion	meaning
in	out	assertion true
out	in	assertion false
in	in	contradiction
out	out	truth value unknown

The meaning of justifications such as the transitivity justification shown above is that whenever F-1 and F-2 are *in* then F-3 should also be *in*. If for some reason either F-1 or F-2 became *out*, then F-3 would lack support and would also become *out*. The *in*ing and *out*ing of facts is managed by the Truth Maintainence System [Doyle, 1978].

It is frequently necessary to *assume* that some fact holds even though no reason exists for believing it. This is often done in hypothetical reasoning as when one proves that A implies B by assuming A and deriving B. One assumes a fact because there is no apparent reason not to believe it; thus, the assumed fact is justified by making it depend on the *out*ness of its negation. This is done by the function ASSUME:

```
(Assume (Made-of The-Moon Green-Cheese) (Bill F-23))
```

which states that the system will believe that the moon is made of green cheese as long as it has no reason to believe that the moon is not made of green cheese and as long as it believes fact F-23. The function ASSUME builds a justifiction which has two parts: A list of assertions upon whose *in*ness the fact depends and a list of assertions upon whose *out*ness the fact depends. When the above ASSUME form is invoked it creates the assertion:

```
(Not (Made-of The-Moon Green-Cheese))          F-1001
```

and then justifies its argument assertion by stating that it depends on F-1001's being *out* and on F-23's being *in*.

```
F-1002  (Made-of The-Moon Green-Cheese) (Bill (F-23)(F-1001))
```

where the first list in the justification is the list of facts whose *in*ness supports F-2 and the second list is the *out* list. Whenever an assertion changes its status, the Truth Maintenance system propagates the change until only assertions with well-founded support remained *in*.

A final type of justification arises in the proofs of implications. As mentioned above, typically one proves A implies B by assuming A and deriving B. However, (Implies A B) depends only on those facts which were involved in deriving B from A and which were not themselves derived from A. The system is instructed to find these supporting facts by the conditional proof justification:

```
(Assert (Implies A B) (Conditional-Proof F-5 F-3))
```

The first fact-name in a Conditional-Proof statement is the consequent of the implication, the second fact-name is the hypothesis of the conditional proof. When such a statement is executed the TMS is invoked to examine the justifications supporting the consequent F-5 and the antecedent F-3.

The implication is supported by the set of facts which support F-5 but which do not include F-3 in their support. The implication is given a justification including exactly these facts. This process is described in more detail in [Doyle, 1978].

We now turn to the issue of control within the reasoning system.

Explicit Control and The Task Network

The traditional weakness of automatic deduction systems is that they are prone to blind searches. Even large amounts of overhead are justified if they can cut down the size of the search space.

The approach we follow here is that all control of the deductive process must be represented explicitly in a form which can be manipulated by the same mechanisms as those which conduct the logical deductive process itself. The ultimate goal motivating this discipline is the requirement that the deductive apparatus be self conscious and able to explain what it is doing at any time. The system can then reason about whether it ought to continue to pursue a particular task, or rather abandon it as hopeless or of too little importance to command further resources and attention.

REASON organizes its operation around a data-structure called the *task network* [McDermott, 77] represented by assertions in the same data base as are facts about the program being analyzed. However, the control assertions in the data base have a justification structure which *outs* them from the data base once their usefulness has passed.

A simple example of the use of control assertions in consequent reasoning will clarify this discipline. For any particular goal there might be several different methods for deriving the desired fact, each of which might create several (conjunctive or disjunctive) subgoals. For example, a particular fact about a list might be derived by backward chaining though some implication, but it might also require a proof by structural induction.

A goal is first proposed as a sub-goal of some higher level task; in REASON these higher level tasks are always sub-tasks of the symbolic program evaluation. However, for simplicity we will refer to this higher level task as "top-level". The creation of a sub-goal stimulates several actions. First REASON assumes that the sub-goal is neither satisfied nor refuted; it then asserts (in the data base) that methods are needed for the sub-goal. This request for methods is justified by a dependency pointing back to the assumption that the goal is neither satisfied nor refuted. If the particular goal is of a type for which a method is known, then the rule capturing this knowledge triggers and asserts the proposal. The proposal is justified with a dependency pointing to the statement which issued the request for methods. The system must now make a conscious choice of which (if any) methods to accept; we shall discuss this below.

REASON accepts a method proposal by making a *show* assertion, triggering the rule which does the actual work of the method. If the desired goal is deduced, a rule is triggered and asserts that the goal is satisfied. If the negation of the goal is ever deduced, a second rule is triggered, and asserts that the goal has been refuted. Either of these events will cause the initial assumption that the goal was neither satisfied nor refuted to go *out*, taking with it all of the dependent control assertions.

The underlying mechanisms of the reasoning system establish a well-defined point at which the system may chose which method(s) to pursue. REASON is a queue based system, the main loop of which consists of finding pairs of rules and matching facts which are added to the process queue. At each iteration one such pair is processed, potentially creating new facts, rules and matches. However, since methods must be accepted before they may act, the queue runs out of matching pairs relatively often. This is the occasion for methods to be chosen. A procedure called the *acceptor* is called to make the the choice.

To illustrate this discipline, suppose we want to prove P and we have (Implies Q P), (Implies R Q) and R. We start by stating that P is a sub-goal of "top-level" (or some more worthy task):

```
F-1        (Implies Q P)                  (Premise)
F-2        (Implies R Q)                  (Premise)
F-3        R                              (Premise)
F-4        (Subgoal P (top-level))        (Premise)
```

Since a sub-goal statement has been asserted, the system makes the assumption that the sub-goal has as yet been neither satisfied nor refuted. Also it creates a goal statement for the newly created goal and justifies this statement with a dependency on the assumption:

```
F-5    (Not (Satisfied F-9))    (Assumption ()(F-6))
F-6    (Satisfied F-9)          (): note lack of justification
F-7    (Not (Refuted F-9))      (Assumption () (F-8))
F-8    (Refuted F-9)            (): note lack of justification
F-9    (Goal P (top-level))     (Sub-goal F-5 F-7)
```

In addition a rule is created to watch for the goal becoming true. When triggered, the rule asserts that the goal is satisfied A second rule for refutation is also created.

```
R-1    (Rule ((:f P))                          (Sat-Rule F-9)
             (Assert (Satisfied F-9)
                     (Satisfaction :f)))
R-2    (Rule ((:f (Not P)))                    (Ref-Rule F-9)
             (Assert (Refuted F-9)
                     (Refutation :f)))
```

The assertion of the goal statement also leads to a request for methods to achieve the goal.

```
F-10  (Methods-Needed (Goal P (toplevel))) (Meth-goal F-9)
```

Now the various method proposers come into play. One obvious method is backward chaining, i.e. finding an implication whose consequence is the desired goal, and then posing the antecedent of the implication as a sub-goal. This would result in the following:

```
F-11    (Method (Goal P (top-level))        (BC-Meth F-10 F-1)
                (Backward-Chain
                      (Implies Q P)))
```

The queue now runs out since there are no other actions possible. At this point the acceptor procedure is invoked. Seeing only one method available, the acceptor makes the obvious choice, accepting the method proposed in F-11.

```
F-12    (Show (Goal P (top-level))             (Acceptor F-11)
           by (Backward-Chain (Implies Q P)))
```

The show assertion triggers the rule which does the actual work of backward chaining:

```
R-10  (Rule
       ((:f (show (Goal :consequent :stack)
               by (backward-chain
                    (Implies :antecedent :consequent)))))
       (assert (subgoal :antecedent (:consequent . :stack))
               (bc-sub-goal :f))
       (Rule ((:g (Implies :antecedent :consequent))
              (:h :antecedent))
             (Assert :consequent (Modus-ponens :g :h))))
```

Thus, we now get a new sub-goal assertion as well as a modus-ponens rule:

```
F-13   (Subgoal Q (P top-level))          (bc-sub-goal F-12)
R-11   (Rule ((:g (Implies Q P))          (sub-rule F-10 R-10)
              (:h Q))
             (Assert P (Modus-ponens :g :h)))
```

The creation of the new sub-goal Q triggers off a series of assertions similar to those triggered by the original sub-goal P. We get the following:

```
F-14 (Not (Satisfied F-18))           (Assumption ()(F-15))
F-15 (Satisfied F-18)                 ()
F-16 (Not (Refuted F-18))             (Assumption () (F-17))
F-17 (Refuted F-18)                   ()
F-18 (Goal Q (P top-level))           (Sub-Goal F-14 F-16)

R-3  (Rule ((:f Q))                   (Sat-goal F-18)
        (Assert (Satisfied F-18)
                (Satisfaction :f)))
R-4  (rule ((:f (not Q)))             (Ref-Goal F-18)
        (Assert (Refuted F-18)
                (Refutation :f)))
F-19 (Methods-Needed                  (Meth-goal F-18)
      (Goal Q (P top-level)))
F-20 (Method                          (BC-Meth F-19 F-2)
      (Goal Q (P top-level))
      (Backward-Chain (Implies R Q)))
```

The acceptor is now invoked and Method F-20 is accepted.

```
F-21  (Show (Goal Q (P top-level))          (Acceptor F-20)
         by  (Backward-chain (Implies R Q)))
```

which in turn triggers the rule for backward chaining. Resulting in:

```
F-22 (subgoal R (Q P top-level))        (bc-sub-goal F-21)
R-5  (Rule ((:g (Implies R Q))          (sub-rule F-21 R-0)
            (:h R))
           (Assert Q (Modus-ponens :g :h)))
```

At this point the necessary facts become available allowing rule R-5 to run on the fact F-3. We obtain:

```
F-23        Q        (Modus-Ponens F-3 F-2)
```

However, Q now triggers the rule R-3 which is watching for an assertion satisfying the goal F-18 (Goal Q (P top-level)). This causes a justification to be added to the "satisfied assertion" F-15 which was created when the goal F-18 was created:

```
F-15        (Satisfied F-18)        (Satisfaction F-23)
```

However, F-14 (Not (Satisfied F-18)) was justified by an assumption dependency on the *outness* of F-15, which now has a valid justification and is, therefore, *in*. Thus, F-14 is made *out* by the truth maintainence system:

```
F-14     (Not (Satisfied F-18))      ()
F-15     (Satisfied F-18)            (Satisfaction F-23)
```

Furthermore, F-14 was the only support for the goal assertion and its dependent *method-needed, method and show* assertions. A quick inspection of the justifications will show that the following support structure exists at this time:

```
F-14 -> F-18 -> F-19 -> F-20 -> F-21 -> F-22 -> R-5
          \
           -> R-3
```

Thus, when F-14 goes *out*, so do all of these assertions. Notice, however, that these are all control assertions. The fact assertion F-23 Q depends only on F-3 and F-2; it stays *in*. Furthermore, F-23 triggers the rule R-2 which represents the modus-ponens deduction for Q and (Implies Q P). We obtain:

```
F-24        P        (Modus-Ponens F-23 F-1)
```

As before, this triggers a goal-satisfied rule, this time R-1 for the goal F-9:

```
F-6       (Satisfied F-9)        (Satisfaction F-24)
```

which causes the not-satisfied assertion F-5 to go *out*. The chain of dependencies causes assertions F-9, F-10, F-11, F-12, F-13 and rules R-1, R-2 to go *out* as well. This leaves us with only the following useful assertions.

```
F-1    (Implies Q P)              (Premise)
F-2    (Implies R Q)              (Premise)
F-3    R                          (Premise)
F-4    (Subgoal P (top-level))    (Premise)
F-6    (Satisfied F-9)            (Satisfaction F-24)
F-15   (Satisfied F-18)           (Satisfaction F-23)
F-23   Q                          (Modus-Ponens F-3 F-2)
F-22   P                          (Modus-Ponens F-23 F-1)
```

Of course, this entire deduction might have been achieved more easily by a simple forward chaining rule for modus ponens. However, we have gone through this detail to illustrate the steps of the protocol. In general, uncontrolled forward chaining is a bad strategy since it allows deductions to lead into endless loops. What is important to notice at this stage is that we have used the dependencies to achieve the effect of contexts as used in Conniver [McDermott, 1972] or QA4 [Rulifson, 1972] without their inflexibility. When we now longer need a control environment we leave it by *outing* certain facts; however, we are free to intermingle any set of control assertions we please, pursuing goals in any order and sharing information learned in the pursuit of one goal with any routine that might profit from this information. There is, however, a virtue to the context mechanism: it neatly isolates assumptions used in hypothetical reasoning. In the next section we extend our formalism to satisfy this need as well.

Hypotheticals

REASON's actual protocol is somewhat more complex than so far illustrated. As a paradigmatic case consider the following problem (we will omit the refutation assertions in this example for the sake of brevity):

```
Given:          (Or A B)
                (Implies A C)
                (Implies B C)
                (Implies C D)
To Show:        D
```

Let us assume that REASON decides to conduct the proof by case-splitting the disjunction (OR A B). This method causes a set of conjunctive sub-goals, in this case, (Implies A D) and (Implies B D). Each of these is proven by the standard conditional proof method, assuming the antecedent and attempting to prove the consequent. We extend the goal assertions to include an (unordered) set of asssertions which have been assumed as part of the hypothetical reasoning process leading to the current sub-goal. We obtain:

```
F-30 (or a b)                              (premise)
F-31 (implies a c)                         (premise)
F-32 (implies b c)                         (premise)
F-33 (implies c d)                         (premise)
F-34 (subgoal d for (top-level) in())      (premise)

F-55 (goal d for (top-level) in ())        (subgoal f-47)


F-67 (show d by  (splitting f-30)          (acceptor f-64 f-66)
       for (top-level) in ())

F-100 (goal (implies a d)                  (subgoal f-74)
        for  (d (top-level)) in ())

F-110 (show (implies a d)                  (acceptor f-105 f-107)
         by  (standard implication)
         for (d (top-level)) in ())

F-112  a                                   (assumption f-113 f-114)

F-126  (goal d                             (subgoal f-116)
         for ((implies a d)
               d (top-level))
          in (a))

F-122  c                                   (mp f-31 f-112)

F-127  d                                   (mp f-33 f-122)
```

The keyword "for" indicates the subgoal stack while the keyword "in" indicates the assumption context. Notice also that two rules are generated to watch for an occurance of the fact D; each such rule asserts that its respective goal is satisfied if the fact D ever comes *in*. However, at this stage the deductions performed do not indicate that the top level goal D (F-55) is satisfied by the fact F-127, only that the sub-goal D (F-126) is. If the system declared F-55 satisfied, then the entire set of control assertions would be *out*ed and the deduction would stop, incorrectly claiming that the main goal had been proven. This would be a mistake.

The classic solution to this problem in Artifical Intelligence languages such as CONNIVER and QA4 is to use a context mechanism to represent the "echelon" in which the implication will be derived. Typically, a new context is created in which the assumption A as well as the new goal D (or its analogue) are asserted. When the fact D comes in, the satisfied assertion is added to this new context which is then discarded. The problem with this approach is that it is altogether possible that the fact D derived in this new context might not in any way depend on the assumption A, in which case the main goal D ought to be satisfied; the context system is incapable of doing this since the contexts are strictly nested. Thus, the chronology of the deductive process, as opposed to logical dependency, play the leading role in such systems.

Our system instead uses the dependencies maintained by the truth maintainence system as well as the explicit control assertions to guide itself to appropriate conclusions. Given an assertion P triggering the pattern of some goal-watching rule, the system must:

1. Request the Truth Maintainence System to prepare a list of all assumptions which support the satisfying fact P.

2. Fetch all goal statements whose goal matches the satisfying fact P.

3. For each goal assertion test whether the assumptions found in 1 are a subset of the assumptions listed in the goal-assertion's context list.

4. Discard those goal assertions which fail the test in 3.

5. For each of the remaining goal assertions in 4 assert that the fact P satisfies the goal assertion.

We can see that this procedure would result in the following assertion when applied to the situation described above:

```
F-120    (satisfied                 (goal-found f-127 f-112)
          (goal d
            for ((implies a d)
                  d (top-level))
            in  (a)))
```

but that the goal assertion F-55 would not be satisfied since it has an empty assumption context while the assertion F-127 depends on the assumption F-112.

The algorithm we have just stated is one which determines whether a certain pattern of dependencies obtains and acts only in that case. This is expressed by a new kind of rule (called an S-RULE, for support rule) which is triggered whenever the pattern of support underlying its trigger fact changes. The rule may then perform whatever test it likes by examining the Truth Maintainence System's justifications. These are included in a list of tests which must succeed before the body is executed. The following is the support rule implementing the above algorithm.

```
(Rule ((:f1 (goal :g for :stack in :context)))
      (S-RULE ((:f2 :g))
              ((subset (assumption-support :f2) :context))
              (assert (satisfied :f1)(goal-found :f2))))
```

We can see that the above rule will trigger correctly and will only assert that the sub-goal F-126 is satisfied. This will *out* the control assertions concerned with this part of the task, achieving the effect of leaving the context. Notice, however, that if the assertion F-127 had not depended on the assumption F-112, then the main goal F-55 would have been satisfied. In that case we would be through and the entire set of control assertion would go *out*.

The system next moves on to proving (Implies B D) which proceeds as above; the support rule triggers and satisfies the sub-goal but not the main goal. Finally, the case-splitting rule completes its work and asserts the fact F-127 D one final time. However the support for this conclusion is the two implications (Implies A D), (Implies B D) and the disjunction (Or A B).

Thus, the support pattern does not involve any assumption and the support rule can now, correctly, conclude that the goal F-55 is satisfied.

Conclusions

Although our discipline of representing control states explicitly involves considerable overhead, it also gives the problem solver greater flexibility. With the price of computational power decreasing with its current breakneck speed, even a large fixed factor overhead can be tolerated as long as this results in flexibility and naturalness in the reasoning system.

This flexibility plays an important role in the program understanding system of which REASON is a part. Programs are understood in this system through a process of symbolic evaluation [Smith & Hewitt, 1975], [Rich & Shrobe, 1976], [King, 1976], [Yonezawa, 77]. One of the difficulties in such systems is the complexity of symbolically evaluating side effects on complicated and shared list structures. The approach taken in REASON is to initially make simplifying assumptions and develop a "first order theory" of the program. This first order theory is represented by TMS dependencies. During the more careful analyses which follow, various of these simplifying assumptions can be removed; the TMS then outs those facts which no longer have valid support, signalling the programmer if crucial facts no longer hold. The programmer can intervene at this point, modifying the program so that it behaves correctly under all conditions. Again the TMS can analyze the effects of the proposed changes by examination of the dependency structure. Thus, the full blown analysis of the program is developed by perturbation of the simpler "first order theory". This technique is explained in greater detail in [Shrobe, 1978].

Acknowledgements: Thanks go to Charles Rich, Richard Waters, Gerald Jay Sussman, Carl Hewitt, Johan de Kleer, and especially Jon Doyle for their collaboration and help.

References

J. DeKleer, et. al. 1977, "AMORD: Explicit Control of Reasoning", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, SigPlan Notices vol 12, no. 8, SigArt Newletter No 64, August 1977.

J. Doyle 1977, "Truth Maintainence Systems for Problem Solving", MIT AI-Lab TR-419.

J.C. King 1976, "Symbolic Execution and Program Testing", *Comm. of the ACM*, Vol. 19, No. 7, July 1976.

J. McCarthy 1968, "Programs With Common Sense" in Semantic Information Processing, Marvin Minsky ed., MIT Press Cambridge, Mass. 1968.

D. McDermott 1972, "The CONNIVER Reference Manual", M.I.T. AI Lab. Memo 259.

D. McDermott 1977, "Flexibility and Efficiency in A Computer Program for Designing Circuits", MIT AI Lab Tr-402.

C. Rich and H.E. Shrobe 1976, "Initial Report On A LISP Programmer's Apprentice", MIT Artificial Intelligence Laboratory Technical Report 354, December 1976.

J. Rulifson, et. al. 1972, "QA4 A Procedure Calculus for Intuitive Reasoning", SRI AI Center, Technical Note 73.

Howard E. Shrobe 1978, "Logic and Reasoning for Complex Program Understanding", Ph.D. thesis MIT Department of Electrical Engineering and Computer Science, August 1978.

B. Smith and C. Hewitt 1975, "Towards a Programming Apprentice", IEEE Transaction on Software Engineering, vol 1, number 1, pp. 26 - 45.

A. Yonezawa 1977, "Specification and Varification Techniques for Parallel Programs Based on Message Passing Semantics", MIT LCS TR-191.

INVERTIBILITY OF LOGIC PROGRAMS

Sharon Sickel
Information Sciences and Crown College
University of California
Santa Cruz, Ca. 95064

Abstract

Predicates can describe functions; the arguments of the predicate are the input and output parameters of the function. Logic programs describe relationships between objects rather than merely sequential instructions, and it is common for both a function and its inverse to be computable by the same logic program [2]. Given values for some subset of the arguments to a function-describing predicate, we may be able to decide, in general, which of the remaining arguments are computable by the logic program.

The concept of functional inverse can be generalized in the context of logic programming. A new kind of inverse, called j-inverse is defined. Two algorithms which analyze and test the recursive structure of logic programs for any specific invertibility are presented. A set of guidelines to help the logic programmer construct j-invertible programs is given.

1. Introduction

A function is a mapping $f:D \rightarrow R$ or $f(x) = y$. $f(x)$ must have a unique value, that is to say it must map to exactly one value in R for any given element of D. We can extend that definition by allowing both the input and the output to be tuples. For example

$$f:D_1 \times \ldots D_n \rightarrow R_1 \times R_2 \times \ldots R_k \quad \text{or}$$
$$f(x_1,\ldots,x_n) = (y_1,\ldots y_k).$$

We say that f is invertible if there exists a function f^{-1} such that $f^{-1}(y) = x$ iff $f(x) = y$.

We wish to restrict the notion of functional inverse as follows:

If $f(x_1,\ldots,x_n) = y$, then we say that f is j-invertible if for $j \le n$, f_j exists as follows:

$$f_j(x_1,\ldots x_{j-1},x_{j+1},\ldots,x_n,y) = x_j$$

That is to say , given all values of the n+1 tuple $(x_1,\ldots,x_n,y)$ except x_j, we can find x_j. f_j is the j-inverse of f. The value y maybe a k-tuple but for the purposes here we do not decompose it as we do the input n-tuple.

If a function f is invertible, then it is j-invertible for all j, $1 \le j \le n$, since invertibility implies that given y we can find $x_1,\ldots,x_n$, so any values x_i that are provided are redundant. The opposite is not true however. Given a function f, even if it is j-invertible for all $1 \le j \le n$, f^{-1} does not necessarily exist. A counterexample is the arithmetic function, add, on integers:

$$\text{add}(a,b) = c$$

There are, in general, many pairs (a,b) that add maps to c; so there is no way to define add^{-1}. However, add is 1- and 2-invertible;

$$\text{add}_1(a,c) = c\text{-}a \text{ and } \text{add}_2(b,c) = c\text{-}b.$$

A function is invertible, in general, when the functional mapping is 1-1 and onto. A function is j-invertible if and only if all pairs of (n+1)-tuples $(x_1,\ldots,x_n,y)$ such that $f(x_1,\ldots,x_n) = y$ differ at position j only if they differ in at least one more position. There are many interesting functions that are not invertible in general but they are j-invertible for some j. Given a function, we frequently would like to be able to define a single program to compute this function that can also compute the partial inverses. We may also want to start with more than one of the parameters unknown and compute one or more of the unknown values.

2. Invertibility of Functions Expressed as Logic Programs

A logic program [4] is a set of WFs in the form $L \leftarrow R$ and one WF, the call, of the form $\leftarrow R$ where L is a predicate and R is a conjunction of predicates. All variables are implicitly universally quantified. Logic programs are theorems describing relationships among objects. While an interpretation of these logic theorems can drive a computation, there is no notion of certain parameters for input and others for output. A nontrivial invertible logic program is given in another paper [3].

We now define some terms that will be needed later. A logic program is a set of logic procedures. A logic procedure is an individual implication $A \leftarrow B$ where B may be empty; A is the procedure head; B is the procedure body. A termination condition of a recursion is a procedure $A \leftarrow B$ such that B does not contain a predicate whose name is the same as the predicate name of A. That is to say, it contains no recursive calls to the procedure. We say that a variable, x, drives the computation of a procedure, if every recursion causes the value of x to be nearer a value that will cause the recursion to terminate.

We now construct a logic program to compute

$$\exp(x,y) = x^y = z$$

by successive multiplications. We assume a 2-invertible predicate MULT(x,y,z) which is true if and only if $x \cdot y = z$, i.e. given x and y computes z and given x and z computes y. This exponential function is defined by predicate EXP1, which has the semantics that EXP1(x,y,z) is true if and only if $x^y = z$.

The logic program is:

$\mathrm{EXP1}(x+1,0,1) \leftarrow$[§]

$\mathrm{EXP1}(x,y+1,w) \leftarrow \mathrm{EXP1}(x,y,z) \wedge \mathrm{MULT}(x,z,w)$

The meaning of $A \leftarrow B$ is the standard logic interpretation "A is implied by B". The meaning of the program is

$(x+1)^0 = 1$

$(x^{y+1} = x \cdot z) \leftarrow (x^y = z)$

It is computationally effective as can be seen by the example below. Starting with the call, and using the procedural interpretation of Horn clauses given by Kowalski [2], we get a terminating sequence of sub-goals:

$\leftarrow \mathrm{EXP1}(3,2,\mathrm{answer})$

$\mathrm{EXP1}(3,2,\mathrm{answer}) \leftarrow \mathrm{EXP1}(3,1,z) \wedge \mathrm{MULT}(3,z,\mathrm{answer})$

$\mathrm{EXP1}(3,1,z) \leftarrow \mathrm{EXP1}(3,0,z') \wedge \mathrm{MULT}(3,z',z)$

$\mathrm{EXP1}(3,0,1) \leftarrow$

The last call to EXP1 gives z' = 1 by the termination condition; MULT then gives z = 3 and then answer = 9, or EXP1(3,2,9), as desired.

Suppose we wish to 2-invert EXP1. For example, evaluate $3^Y = 9$ to get Y = 2. The sequence of calls starts

$\leftarrow \mathrm{EXP1}(3,Y,9)$

$\mathrm{EXP1}(3,Y,9) \leftarrow \mathrm{EXP1}(3,y,z) \wedge \mathrm{MULT}(3,z,9)$

Since MULT is 2-invertible, we get z = 3; hence substituting 3 for z we derive EXP1(3,y,3) where Y is bound to y+1. Continuing we get new subgoals

$\mathrm{EXP1}(3,y,3) \leftarrow \mathrm{EXP1}(3,y',z') \wedge \mathrm{MULT}(3,z',3)$

with y bound to y'+1. MULT(3,z',3) gives z' = 1. Then in evaluating EXP1(3,y',1), the terminating condition applies; thus, y' = 0, y = 1, Y = 2, and we derive EXP1(3,2,9), as desired.

Suppose we wish to 1-invert EXP1. For example, for what value of X will $X^3 = 27$? The following sequence of subgoals is generated:

$\leftarrow \mathrm{EXP1}(X,3,27)$

$\mathrm{EXP1}(X,3,27) \leftarrow \mathrm{EXP1}(X,2,w) \wedge \mathrm{MULT}(X,w,27)$

$\mathrm{EXP1}(X,2,w) \leftarrow \mathrm{EXP1}(X,1,w') \wedge \mathrm{MULT}(X,w',w)$

$\mathrm{EXP1}(X,1,w') \leftarrow \mathrm{EXP1}(X,0,w'') \wedge \mathrm{MULT}(X,w'',w')$

$\mathrm{EXP1}(X,0,1) \leftarrow$

[§] In $\mathrm{EXP}(x+1,0,1) \leftarrow$ the expression x+1 is used to denote a nonzero value. Another representation for this is

$\mathrm{EXP}(x,0,1) \leftarrow (x > 0)$, but the latter has a slight computational disadvantage.

The termination condition gives us w" = 1 and the definition of MULT yields X = w' giving

$\mathrm{EXP1}(X,1,X) \wedge \mathrm{MULT}(X,X,w)$

at the first level of recursive return. But we fail here because while MULT can, awkwardly, compute square root by calling MULT(x,x,C) where C is a given constant natural number, it cannot compute square roots where the third argument is unknown as well. In effect, the collective calls to MULT are taking n-th roots of the original third argument, and that is beyond the capability of MULT.

Now we make a distinction between mathematical invertibility and computational invertibility. A function f may be mathematically invertible but be specified as a logic program that will not compute the inverse. If a function is computationally invertible it is necessarily mathematically invertible.

It may also be the case that a mathematically correct logic program will be computationally reasonable in one direction and horribly inefficient in the other. The path of choices in a computation may be forking such that in one direction many cases are being joined together as in Figure 1a, and in the other direction a choice must be made from many alternatives as in Figure 1b. The latter case may require much backtracking where incorrect choices are made.

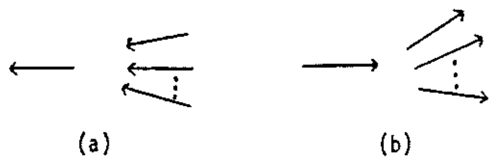


Figure 1

3. When are Logic Programs Invertible?

The process of interpreting a logic program starts at the call, descends recursively to a termination condition, then returns eventually to the call. The question of j-invertibility depends upon the sequence of bindings of variables as the recursion is carried out; success is achieved when the (unbound) j-th variable of the call is bound to a constant value. More generally, if any subset of the variables is known at the call, we can ask if any of the unknown values can be computed.

Parameter list patterns are used to designate in a parameter list which parameters are known (i.e. constants) and which are unknown (i.e. variables or functions of variables). Given a parameter list input to a procedure, we can construct an <u>input pattern</u> or <u>input template</u> by replacing constants in the parameter list by 1's, and replacing variables or functions[§] of variables by 0's. So, for example, from parameter list (2,x,y+1) we construct template (1,0,0). We may tie parameters together if some of the unknown variables are the same or simple functions of each other, such that if one can be computed, the other is known as well. E.g. parameter list (2,x,x+1) corresponds to pattern (1,0,0).

We have two algorithms that map input templates, i.e. known vs. unknown input parameters, to output templates that show what total set of values is known after the computation. For example, for procedure MULT, input pattern (101) maps to (111), since given the first factor and the product, the other factor can be determined.

The first algorithm is a simple, easy to compute, function that gives a strong indicator of the pattern mapping, but cases can be constructed in which the computed mapping will be stronger than reality. The second algorithm is similar to the first, but makes added assumptions that require some meta-analysis of the given function. The second algorithm is a sufficient test, i.e. at least all of the values claimed to be known, will be known.

An input pattern tells what input values are known. From that we can show where known values appear in the entire procedure. For example, input pattern (101) given to the head of the recursive case of EXP1 gives procedure pattern:

EXP1(101) ← EXP1(100) ∧ MULT(101)

I.e. the first and third parameters of the head, x and w, are known throughout the procedure.

The procedure operates in two stages:

1) Going down the recursion, we check off values when we know that they are computable. Even though the recursion can go arbitrarily deep, there are a bounded number of procedure patterns.

2) Once all the derivable procedure patterns are found, patterns of the termination conditions are applied to show which values will be given by termination.

Throughout both stages, propagate new values for variables wherever the variables appear, and when applicable, replace the input patterns of subfunctions by the output patterns to which they map. E.g., for the procedure above, MULT(101) is known to map to MULT(111); fill in the "known" indicator wherever the newly computed variable appears. So, the procedure pattern becomes:

EXP1(101) ← EXP1(101) ∧ MULT(111)

[§] We assume the only functions that are allowed as part of the parameter lists are simple constructor functions, e.g. +1.

Then, since the recursive call to EXP1 carries the same pattern as the original, we may assume that that pattern is continued as long as we make recursive calls. Eventually, when (and if) the termination case applies, we indicate the known values supplied by termination. In the above example, the termination condition

EXP(x+1,0,1) ←

has pattern (011). So, at the time of termination, the second and third parameters, at least, are known. We perform an inclusive or on the terminating pattern and the pattern of the subgoal to which it applies:

EXP1(101) ← EXP1(101) ∧ MULT(111)
EXP1(011)

deriving

EXP1(111) ← EXP1(111) ∧ MULT(111).

From this we know that at the deepest level of recursion, the desired unknown is supplied, and returned to the calling subgoal, and likewise at every level of recursion, since the return supplies as many knowns as the termination.

<u>Algorithm 1: Heuristic</u>

Purpose: To discover the input-to-output mappings of a predicate, Q.

Hypotheses: All auxiliary functions' input-to-output mappings are given. The only functions appearing in terms are constructor functions.

1. <u>Set-up</u>.

   The input pattern is determined from the call: ← Q(p1,...,pn). We replace in the call:

   a) each constant or constructor applied to a constant by the value 1.

   b) each constructor applied to a variable by the variable itself. Note that now our patterns are made up of variables and 1's. Connections between unknowns are now implicit by their having a single name.

   The <u>procedure templates</u> are determined from logic procedures of the form

   Q(t1,...,tn) ← R1 ∧ ... ∧ Rk.

   We replace each term in the procedure as in 1a and 1b, above. A template is recursive if the logic procedure was recursive. The call is considered recursive.

   Let PPS be the set containing the call as modified in 1a & b.

2. <u>Propagation of known values</u>.

   While there is a conjunct Q(p1,...,pn) appearing in the body of a procedure pattern of PPS such that Q(p1,...,pn) does not appear as the head of an element of PPS (up to a change of variables), create new, additional elements of PPS from the procedure templates as follows:

a) Unify the head of a new instance of each recursive template with Q(p1,...,pn).

b) The auxiliary functions are the subgoals of the procedure template that are not recursive calls to Q. Unify the inputs of the auxiliary functions with their corresponding output mapping. E.g. MULT maps (1z1) to (111). Unifying these two patterns binds z to 1. (This records which variables become bound in the course of the computation of the auxiliary function.) This step is repeated until no more changes are possible.

c) Add this new procedure pattern to PPS.

3. Applying termination cases.

a) Add all non-recursive procedure templates to PPS.

b) Partition the elements of PPS into recursive and non-recursive sets.

c) Select a recursive subgoal from the body of an element of PPS, and unify it with the nonempty head of any non-recursive element. Apply the bindings only to the recursive pattern. Drop the recursive subgoal that was just unified. Handle auxiliary functions as in 2b. The modified recursive procedure may now be non-recursive. If so, move it to the non-recursive partition.

d) Repeat b) until the recursive partition is empty.

4. Interpretation of results.

The output pattern Q(p1,...,pn) such that (p1,...,pn) is the most specific parameter list (i.e. contains the most number of 1's) that will unify with the parameter lists of all the heads of the procedures of PPS and the unified call, without binding to 1's any variables of the heads, then inclusive or'd with the input pattern.

Example

Show Expl maps (101) to (111).

Step 1

PPS = {←EXP1(1,Y,1)}

Procedure templates are

EXP1(x,1,1) ←

EXP1(X,y,w) ← EXP1(x,y,z) ∧ MULT(x,z,w)

Step 2 (only one iteration required)

PPS' = {← EXP1(1,Y,1),
EXP1(1,Y,1) ← EXP1(1,Y,1) ∧ MULT(1,1,1)}

Step 3

PPS" = {← EXP1(1,Y,1),
EXP1(1,1,1) ← MULT(1,1,1),
EXP1(x,1,1) ←}

Step 4

Output Pattern = (1,1,1) (or(111))

Example

Show Expl maps (011) to itself, i.e. the unknown is not computed.

Step 1

PPS = {← EXP1(X,1,1)}

Procedure templates as before

Step 2 (requires two iterations)

PPS' = {← EXP1(X,1,1),
EXP1(X,1,1) ← EXP1(X,1,z) ∧ MULT(X,z,1)}

PPS" = {← EXP1(X,1,1),
EXP1(X,1,1) ← EXP1(X,1,z) ∧ MULT(X,z,1),
EXP1(X,1,z) ← EXP1(X,1,z') ∧ MULT(x,z',z)}

Step 3

PPS''' = {← EXP1(X,1,1),
EXP1(1,1,1) ← MULT(1,1,1),
EXP1(X,1,z) ← MULT(X,1,z),
EXP1(X,1,1) ←}

Step 4

Output Pattern = (u,1,v) ∨ (011)
= (010) ∨ (011)
= (011)

So, nothing new is computed.

Algorithm 2: Precise

This algorithm is the same as the previous one except that in steps 2a and 3c when 1's are unified with 1's, a case-by-case meta-argument is required to show that the actual constants are unifiable. If such is not the case, the procedure pattern being generated is discarded.

The meta-arguments are handled by expressing the range of a variable as recurrence relations. Good programs already exist for handling such analysis [1]. Then, so long as the input value can be shown (or assumed) to be in the range of the recurrence relation, the binding is allowed to proceed.

The following theorem shows the soundness of

Algorithm 2. Without loss of generality, the argument lists are represented as two variables, x and y. The input vector is x and the output vector is y.

Theorem. Algorithm 2 is sound, i.e. if procedure P(x,y) is shown by Algorithm 2 to map input template (10) to output template (11), then procedure P, given constant input x within the domain required by the meta-analysis of Algorithm 2, will compute y.

Proof. The proof is by induction on the depth of recursion of the computation for an arbitrary constant x. Suppose that Algorithm 2 claims that y is computable given a constant x, and x is within the domain demanded for x, then there are two cases, depending upon whether x is an initial value of its recurrence relation or not.

Basis Case. x is an initial value of its recurrence relation. Then the result of unifying the call with the proper termination condition must have supplied value y, since the output pattern is restricted by that binding.

General Case. x is not an initial value, but is included by the recurrence relation. Then a recursive procedure

$$P(u,v) \leftarrow Q_1(u,v,w) \wedge \dots \wedge Q_n(u,v,w)$$

must exist such that the meta-analysis showed that (u,v) is unifiable with (x,y), and we assume by induction that the Qi's collectively compute the components of vector v, given u. Therefore, P computes v given u, by returning the computed v. Q.E.D.

Following the execution of Algorithm 1, the only case of unifying 1's with 1's in steps 2a and 3c, takes place in step 3c, when head EXP1(x,1,1) is unified with body subgoal EXP1(1,Y,1). We need to show that since unifying the third parameters unifies 1's, that the third argument of the recursive subgoal eventually reaches the third argument of the termination case, namely constant value 1. (It is coincidental that the constant is the same as the placeholder for constants.) The recursive relations for the third parameter are:

$$z_0 = 1$$
$$z_{n+1} = z_n \cdot x$$

where x is a given natural number, the first argument. So, as long as the third parameter of the input is, in fact, a power of x, the application of the termination case in 3c is guaranteed.

In the second example, again the only unification of 1's with 1's takes place in step 3c, where EXP1(X,1,z) and EXP1(X,1,z') each are unified with EXP1(x,1,1). Here the recurrence relations are the same for the two situations and involve the second parameter. The recurrence relations are:

$$y_0 = 0$$
$$y_{n+1} = y_n + 1$$

So, if the input value of the second parameter is a natural number, the binding is allowed and termination is guaranteed.

4. Constructing Invertible Programs

The preceding algorithms determine feasibility of various directions of computation of a program. The following are a set of guidelines that help the logic programmer to construct j-invertible programs. These guidelines exclude some j-invertible logic programs and are thus overly restrictive. More liberal, but less intuitive guidelines, can be derived from the preceding algorithm.

A. Termination conditions: There should be a set of termination conditions, one of which can always be reached from a call in which the j-th parameter is the only unknown and in which the j-th parameter is known or can be directly computed as a function of some of the other arguments.

For example, consider the termination condition MULT(x,0,0) ←. It is acceptable for 2-invertibility because the second parameter is a constant and we always reach this termination condition. However, it is not acceptable for 1-invertibility since once we recur and bottom-out, we still have no way to establish the value of x except by evaluating algebraic formulas. For example, MULT(x,2,6) calls MULT(x,1,z) where z = 6-x which calls MULT(x,0,z') where z = 6-x-x which = 0. So with a little algebra or exhaustive backtracking we can deduce that x = 3. However, this is not what is meant by directly computable.

B. Invertible subfunctions: The other functions used in the definition must be inverible as called.

For example, consider the following predicate:

$$F(x_1,x_2+1,y) \leftarrow F(x_1,x_2,y') \wedge G(y',y)$$

For F to be 1-invertible, G must be 1-invertible, since y is supplied and y' needs to be computed.

C. Driving the Computation: At least one known value must drive the computation toward termination.

For example, consider the function F, above. If x_2 is unknown, it cannot drive the computation. So for F to be 2-invertible, since x_1 does not change, the mapping $y \underset{G}{\rightarrow} y'$ must drive toward a termination condition.

A way to guarantee that this property holds for any j-invertibility, $1 \le j \le n$ is to have at least two arguments driving the computation.

D. Preconditions: A deciding precondition must apply to a known value.

A precondition is a predicate, used in the body of a logic procedure, that gives the criterion for choosing that procedure. The precondition may or may not be essential to the mathematical definition. But without the aid of preconditions, much backtracking may, in general, be required.

There is nothing special syntactically about the preconditions, and recognition of predicates as preconditions is totally a control issue, i.e. something known by the system that provides an interpretation for the logic program.

The following is an example of logic procedures named F in which the choice among the procedures is determined by preconditions which test the relative sizes of x and y.

$$F(x,y,z) \leftarrow (x=y) \wedge G_1(x,y,z)$$

$$F(x,y,z) \leftarrow (x>y) \wedge G_2(x,y,z)$$

$$F(x,y,z) \leftarrow (x<y) \wedge G_3(x,y,z)$$

(Another example of preconditions is in the FACTOR procedure presented later. GCD is necessary to the mathematical definition, but can also be used as both precondition and termination condition if w and n are given. In the case where p is not given, i.e. 3-invertibility, its use as a termination condition is essential to the efficiency of the algorithm.)

Since the preconditions determine the efficiency of the algorithm, one must have some way of tying the preconditions to the parameters such that certain preconditions are used if certain parameters are known. A way around this problem is to have at least two preconditions, one of which can always succeed if, at most, one of the n+1 arguments of the original function are unknown. The following is an example:

$$F(x,y,z) \leftarrow (x=y) \wedge P_1(z) \wedge G_1(x,y,z)$$

$$F(x,y,z) \leftarrow (x>y) \wedge P_2(z) \wedge G_2(x,y,z)$$

$$F(x,y,z) \leftarrow (x<y) \wedge P_3(z) \wedge G_3(x,y,z)$$

Now, so long as at least two out of three of the values x, y, and z are given, either the relative sizes of x and y can be determined, or the predicate P_i can be applied to z. So, if the comparisons between x and y and the P_i's serve as preconditions, at least one will always be computable in each procedure. For the P_i's to efficiently serve as preconditions, exactly one of $P_1(z)$, $P_2(z)$, and $P_3(z)$ should be true for a given z.

5. Why is EXP1 2-invertible but not 1-invertible?

The 2-invert test on EXP1 succeeds, i.e. (101) maps to (111), but the 1-invert test fails, i.e. (011) maps to (011). Intuitively, that makes sense according to the guidelines. EXP1 fails condition A for 1-invertibility since the first parameter of the termination condition is neither constant nor can be computed from the others. The guidelines are met, however, for 2-inversion.

6. Another Exponentiation Algorithm

There is an entirely different approach to exp(x,y) that is more obviously j-invertible. It is based on the fact that every positive integer, x, is a product of powers of primes, that is to say

$$x = 2^{v1}\, 3^{v2} \ldots p_k^{vk}.$$

We have $z = x^y$ iff

$$z = 2^{y\cdot v1}\, 3^{y\cdot v2} \ldots p_k^{y\cdot vk}.$$

If we are given x and y, and we can factor x, then we can construct z out of the factors; given x and z, the exponents of the primes in the factorization of z must be a constant multiple, y, of the corresponding prime factors of x; given y and z, factor z, divide the exponents of the prime factors by y, and collect the factors to create x.

We must first provide an invertible factoring predicate. Suppose we have a predicate, GCD(x,y,z), such that z is the greatest common divisor of x and y, and z is undefined if x or y is zero. The FACTOR(W,N,P,R) is true if and only $W > 0$, $N > 0$, $R > 0$, $W = N^P \cdot R$, and N does not divide R. FACTOR is defined as follows:

```
FACTOR(w,n,0,w) ← GCD(w,n,1)
FACTOR(w,n,p+1,r) ← GCD(w,n,n)
                    ∧ MULT(w',n,w)
                    ∧ FACTOR(w',n,p,r)
```

The normal call is, for example:

← FACTOR(36,2,p,r)

which yields

← FACTOR(36,2,2,9).

A 1-invertible call is:

FACTOR(answer,2,2,9)

which yields

FACTOR(36,2,2,9).

These are the only ways that FACTOR will be called by EXP3.

EXP3(X,Y,Z) is true if and only if $X^Y = Z$. We have auxiliary function E such that E(X,N,Y,Z) is true if and only if $X^Y = Z$ for $X > 0$, $Z > 0$ and for all m, $2 \le m \le N$, m does not divide X.

The formal definitions of EXP3 and E are:

$$\mathrm{EXP3}(x,y,z) \leftarrow E(x,2,y,z)$$

$$\begin{aligned}
&E(1,n,y,1) \leftarrow \\
&E(x,n,y,z) \leftarrow \mathrm{FACTOR}(x,n,p,r) \\
&\qquad \wedge \mathrm{FACTOR}(z,n,p',r') \\
&\qquad \wedge \mathrm{MULT}(p,y,p') \\
&\qquad \wedge E(r,n+1,y,r')
\end{aligned}$$

E repeatedly removes a prime factor, p_i, from each of x and z and checks to see that their powers are in the proper relationship, i.e. p_i^{vi} for x and $p_i^{y \cdot vi}$ for z.

The semantics of E's two procedures is:

$$1^y = 1 \text{ and no } m,\ 2 \le m \le n \text{ divides } 1$$

$$(n^p \cdot r)^y = (n^{y \cdot p} \cdot r') \leftarrow (r^y = r') \text{ and no } m,\ 2 \le m < n \text{ divides } r$$

EXP3 is both 1- and 2-invertible. Calling EXP3 with pattern (011) calls E with pattern (0111). The invert test for E on (0111) yields (1111). Calling EXP3 with pattern (101) calls E with pattern (1101), and the invert test on (1101) yields (1111). Notice that E passes the 3-invert test, even though the third parameter of the termination condition is neither constant nor computable from the other parameters. This demonstrates the conservatism of guideline A.

Note that E and FACTOR are not j-invertible for all j. However, that creates no difficulties. E is called with only patterns (0111), (1101), and (1110) and FACTOR with only patterns (1100) and (0111). All of those computations succeed. The invert test gives the proper answers for all cases of EXP3, E, and FACTOR, as well as all other examples in this paper.

If we wished to have a factor program that is j-invertible for all j, we could define a new one, FACTOR2.

$$\begin{aligned}
\mathrm{FACTOR2}(w,n,p,r) &\leftarrow \mathrm{EXP3}(n,p,z) \\
&\quad \wedge \mathrm{MULT}(z,r,w)
\end{aligned}$$

This new program is totally j-invertible, but will not produce p and r, given w and n, as FACTOR would. These characteristics of FACTOR2 are demonstrated by the mappings of the invert test on the input patterns:

(0111) → (1111)
(1011) → (1111)
(1101) → (1111)
(1110) → (1111)
(1100) → (1100)

7. Conclusions

We have defined the concept of j-invertibility for function and given two algorithms to test logic programs for invertibility, and while the answer is not definitive, it is indicative. The algorithms do more than just test for j-invertibility; they map arbitrary input patterns to output patterns.

We have also presented some guidelines for constructing j-invertible functions. The guidelines are in terms of the predicate form of definition of the function, and are syntactic in nature.

The algorithm could also be used by a logic interpreter in choosing the order of evaluation of subgoals of a given procedure. That is, given a procedure invocation, including its list of arguments, the interpreter can determine a partial order on the subgoals which places the most completely evaluable subgoals first, preventing procedures from being called before they have enough information to carry out their computations. Such control can be applied dynamically by the interpreter.

Invertibility can be looked at in another way. The number of unique variables appearing in the parameters of a procedure call is the degree of freedom of that call. In general, the procedure may be able to reduce the degree of freedom by binding some of the variables. For example, MULT(x,x,9) has one degree of freedom and in fact the usual definition of MULT will yield MULT(3,3,9), computing the square root and leaving zero degrees of freedom. Similarly, MULT(x,1,z) has two degrees of freedom. MULT will determine that x = z, i.e. MULT(x,1,x), reducing the degree of freedom to one. Such situations can arise naturally in inverting predicates. How to treat them is an interesting question for which we do not yet have an answer.

References

1. Cheatham, T.E. Jr., Program Loop Analysis by Solving First Order Recurrence Relations, Technical Report TR-13-78, Center for Research in Computing Technology, Harvard University, Cambridge, Mass. (1978).
2. Kowalski, Robert. Predicate Logic as a Programming Language. Information Processing 74, North-Holland Publishing (1974).
3. Sickel, Sharon and McKeeman, W.M. Axiomatic Specification of Syntax-directed Translation. Technical Report 78-8-002, Information Sciences, University of California, Santa Cruz, Ca. (1978).
4. van Emden, M.H. and Kowalski, R.A. The Semantics of Predicate Logic as a Programming Language. J.ACM 23,4 (Oct. 1976), 733-742.

SUB-PROBLEM FINDER AND INSTANCE CHECKER

TWO COOPERATING PREPROCESSORS FOR THEOREM PROVERS

Dennis de Champeaux
Bedrijfsinformatica
University of Amsterdam
Mirror Co

Abstract

Two pre-processors for theorem provers are described, which when applicable, will lead to search space simplification.
Both were implemented and integrated with an existing resolution type connection graph theorem prover. Examples are provided which confirm our claim of search space simplification.

Key words and phrases: Theorem proving, pre-processing, search space simplification.

Contents:

1. Pre-processors for theorem provers in general

The main paradigm in automatic theorem proving is or should be that search is to be avoided, postponed, or else to be minimized. This should be done by any means one can lay one's hand on while maintaining completeness, generality and not succumbing to the ad-hocness as advertised by the proceduralists.

Search space simplification has been a major goal in the resolution school. Exploiting 'larger' more numerous operators (derivation rules) is the main activity in the natural deduction school. There are however many more options to be developed with which the role of search can be limited to those circumstances where there is no sensible alternative.

Already slumbering for some years is the application of multi-level search in theorem proving (Sacerdoti, 1973, /7/). It is incomprehensible that the support from a model technique (Gelernter, 1959, /3/) has not been pursued. Search guidance with heuristic functions and automatic improvement of them is still inconclusive, although it has become clear that syntactic features can contribute only modestly to such functions. Chosing an appropriate representation (within one language say predicate calculus) is a wide open problem. Determining the right representation language is not yet a problem since it is not known whether multiple representation languages are necessary.

The former two issues belong to the preprocessing repertoire. We mention a few more. Selection of relevant axioms and/or definitions and/or already proven theorems/lemmas to prove a conjecture; finding a counter example of a conjecture to make sure that a proof is impossible; finding a similar already proven theorem to see whether its proof can be generalized and/or modified to handle a conjecture; instantiating a second order theorem (e.g. induction scheme); reducing a conjecture to independent sub-problems (reduction to weakly dependent sub-problems can be done in the multi-level search framework); recognizing that a conjecture is an alphabetic variant and/or an instantiation of an axiom or an already proven theorem; etc.

This paper reports the results of implementing the two last mentioned. The next section describes the independent sub-problem recognizer. Section 3 deals with the instance checker. Section 4 discusses how they are cooperating and how they should be related from a process point of view. Two examples are given in section 5.

2. The independent sub-problem recognizer

The sub-problem recognizer we developed is based on components of a predicate calculus - conjunctive normal form translator. Our translator was inspired by the procedure as described in /Manna, 6/.
A small improvement to this translator led to sub-problem decomposition. First we present the original translation. Then we expand one of the steps (as was also partially done in /Loveland, 5/ page 34). A subset of these translator rules make up the sub-problem recognizer.

The translator in /Manna, 6/, omitting here irrelevancies, consists of the following steps:

1- Eliminate 'if ... then' and 'if and only if'.
   Replace $A \supset B$ by $\vee(\sim A, B)$, and
   $A \equiv B$ by $\wedge(\vee(\sim A, B), \vee(A, \sim B))$.
2- Move 'not' inwards.
   Replace $\sim(x)A$ by $(\exists x)\sim A$,
   $\sim(\exists x)A$ by $(x)\sim A$,
   $\sim\vee(A1, \ldots, An)$ by $\wedge(\sim A1, \ldots, \sim An)$,
   $\sim\wedge(A1, \ldots, An)$ by $\vee(\sim A1, \ldots, \sim An)$, and
   $\sim\sim A$ by A.
3- Push quantifiers to the right.
   Let (Qx) be (x) or $(\exists x)$, and let $\bowtie$ be $\wedge$ or $\vee$;
   replace $(Qx) \bowtie (A1, \ldots, Ai, \ldots, An)$ by
   $\bowtie\{Ai, (Qx)\bowtie(A1, \ldots, Ai-1, Ai+1, \ldots, An)\}$
   if x not free in Ai.
4- Eliminate existential quantifiers (introduction of Skolem functions). Pickout the leftmost well formed part $(\exists y)(B(y)$ and replace it by $B(f(x_{i1}, \ldots, x_{in}))$ where
   a) $x_{i1}, \ldots, x_{in}$ are the free variables of $(\exists y)(B(y))$ which are universally quantified to the left of $(\exists y)(B(y))$;
   b) f is a 'fresh' n-ary function constant.
5- Eliminate universal quantifiers.
6- Distribute 'and' over 'or'.
   Replace $\vee(A1, \ldots, Ai, \wedge(B1, \ldots, Bk), Ai+1, \ldots, An)$ by
   $\wedge\{\vee(A1, \ldots, B1, \ldots, An), \ldots, \vee(A1, \ldots, Bk, \ldots, An)\}$.

If step(i) can be reapplied it has precedence over step (i+1).

Remark: A sequence of say universal quantifiers should get special attention in step 3. E.g. $(x)(y)\{\vee P(x,y),\ Q(y)\}$ can be replaced by $(y)\vee\{Q(y),\ (x)P(x,y)\}$.
Thus every universal (existential) quantifier in a sequence of universal (existential) quantifiers should be moved to the right of the sequence in its turn to check whether it can be pushed further to the right.

This procedure can be improved by expanding step 3 with:

3.1 Straighten out 'and's and 'or's.
Let $\mathbb{X}$ be $\wedge$ or $\vee$.
Replace $\mathbb{X}(A1, \ldots, Ai, \mathbb{X}(B1, \ldots, Bk), Ai+1, \ldots, An)$ by
$\mathbb{X}(A1, \ldots, Ai, B1, \ldots, Bk, Ai+1, \ldots, An)$.

3.2 Distribute 'and' over 'or' or vice versa.
Replace
$(x)\vee(A1, \ldots, Ai, \wedge(B1, \ldots, Bk), Ai+1, \ldots, An)$ by
$(x)\wedge(\vee(A1, \ldots, B1, \ldots, An), \ldots, \vee(A1, \ldots, Bk, \ldots, An))$ and
$(\exists x)\wedge(A1, \ldots, Ai, \vee(B1, \ldots, Bk), Ai+1, \ldots, An)$ by
$(\exists x)\vee(\wedge(A1, \ldots, B1, \ldots, An), \ldots, \wedge(A1, \ldots, Bk, \ldots, An))$.

3.3 Distribute quantifiers over connectives.
Replace
$(x)\wedge(A1, \ldots, An)$ by $\wedge((x)A1, \ldots, (x)An)$ and
$(\exists x)\vee(A1, \ldots, An)$ by $\vee((\exists x)A1, \ldots, (\exists x)An)$.

Example: Using the extended step 3 the formula:
$(x)[P(x)\wedge(y)\{Q(x,y)\wedge(\exists z)R(y,z)\}]$
can be rewritten into:
$(x)\ P(x)\wedge(y)(x)Q(x,y)\wedge(y)(\exists z)R(y,z)$.

Observe that this replacement simplifies the Skolem function that has to be generated for the elimination of the $(\exists z)$-quantifier.

Remark: It is worthwhile to check for redundancies (using the instance checker) after application of step 3.2.
E.g. $\vee((x)A(x),\ A(a), \ldots)$ can be simplified to $\vee(A(a), \ldots)$ while
$\wedge((x)A(x),\ A(a), \ldots)$ can be replaced by $\wedge((x)A(x), \ldots)$.

Remark: For easy readability we used in 3.3 the same variable 'x' in all the terms of the connective in the replacement. This does not hurt the translator. The instance checker however can get confused and demands step 3.3 to be modified such that each term in the replacement has a new fresh variable.

The INdependent SUB-problem REcognizeR - INSURER - consists of step 1 upto the modified step 3 and with step 6. It is obvious that in case INSURER applied on a problem P produces a form with leading connective 'and' its terms are independent sub-problems and if all sub-problems can be proven the P has been solved. The reverse statement that INSURER will find a maximal decomposition if P can be decomposed we leave as an intriguing task for Aiello/Weyrauch's program FOL /1/.

We end this section with the 'real-life' example given in box 1.

(1) $(x)(y)(z)\quad x(yz)=(xy)z$
(2) $(x)\ xe=x$
(3) $(x)\ ex=x$
(4) $(x)\ xI(x)=e$
(5) $(x)\ I(x)x=e$
(6) $(H)(\mathrm{SUBGR}(H)\leftrightarrow\{(\exists x)H(x)\wedge (x)(y)[H(x)\wedge H(y)\rightarrow H(xy)]\wedge (x)(H(x)\rightarrow H(I(x)))\})$
(7) $(H1)(H2)\{\mathrm{SETEQ}(H1,H2)\leftrightarrow(x)(H1(x)\leftrightarrow H2(x))\}$
(8) $(g)(xx)(H)(\mathrm{COSET}(g,xx,H)\leftrightarrow \{\mathrm{SUBGR}(H)\wedge (x)(xx(x)\leftrightarrow(\exists y)(H(y)\wedge x=yg))\})$
(9) $(g)(xx)(H)(\mathrm{COSET}(g,xx,H)\rightarrow\{H(g)\leftrightarrow\mathrm{SETEQ}(xx,H)\})$

box 1 Axioms (1-5), definitions (6-8) and a theorem (9) from group theory.

(1-5) are non-minimal axioms defining a group; (6-8) are definitions of respectively sub-groups, equality of subsets and of right-cosets; (9) is a theorem expressing a property of cosets. Observe that subsets are represented by 1-argument first order predicates, and that SETEQ and COSET are 2nd order predicates. Direct translation of (1-8) and the negation of (9) into conjunctive normal form yields 39 clauses with together 109 literals. INSURER however recognizes that (9) can be decomposed into:

(10) $(g)(H)\{H(g)\vee(xx)[\sim\mathrm{COSET}(g,xx,H)\vee \sim\mathrm{SETEQ}(xx,H)]\}$ and
(11) $(g)(H)\{\sim H(g)\vee(xx)[\sim\mathrm{COSET}(g,xx,H)\vee \sim\mathrm{SETEQ}(xx),H)]\}$.

Working on (10) (not done by INSURER, but by another program component) the definition of COSET, SETEQ and SUBGR are respectively substituted. The result is negated and together with (1-5) translated into conjunctive normal form yielding 14 clauses with 23 literals. After each substitution of a definition INSURER is called to check whether further decomposition is possible. When working on (11) this strategy is successful after substituting away SETEQ. Two new sub-problems are found both ending up with 14 clauses and 21 literals.

Although our connection graph theorem prover COGITO is not yet able to handle these three sub-problems, the chance to find a solution has increased with an 'infinite' amount when compared to the non-decomposed situation.

INSURER also can handle the sorted predicate calculus that was introduced in /Champeaux, 2/. The same coset example formulated in sorted predicate calculus - without decomposition - yields 28 clauses with 61 literals. INSURER finds here also three sub-problems each having 12 clauses with respectively 16, 14 and 14 literals. A significant reduction again, possibly bringing this problem within reach of the with paramodulation extended COGITO.

3. The Instance checker

The instance checker (INSTANCE) we designed and implemented is in fact a special case theorem prover. It is an instrument with which a conjecture can be recognized as being an alphabetic variant or as a special case of an already accepted theorem. Conjecture and theorem are expressed as closed, slightly restricted - see below - predicate calculus formulas.

Let T and K be respectively a conjecture and an accepted piece of 'knowledge'. The input of the recursive INSTANCE consists of three elements:

-- two forms T and K for which must hold that they do not share variables and that disregarding permutations of sequences of quantifiers and/or arguments of 'and' and 'or' it is the case that T = INSURER(T) and K = INSURER(K); and
-- a list of variables, VR, to be explained in the sequel, which at the top level call is empty.

Although at the top level one is mostly interested in a yes-no answer, for reasons that should become clear in the description of INSTANCE, the output is more substantial in the positive case. The output of INSTANCE is:

-- NO, signifying that T is not an instance of K, or else
-- a list of triples, where each triple is of the form $\{\sigma, T\sigma, K\sigma\}$, with σ a substitution, $T\sigma$ a "without loss of generality substitution instance" of T being a logical instance of the special case $K\sigma$ of K. It should now be clear that if NO $\neq$ INSTANCE(T,K,$\emptyset$) then K $\vdash$ T.

An example where more than one triple will be returned by INSTANCE is:
$T = (\exists x)A x,x)$,
$K = (y)(A(p,y) \wedge A(q,y))$, with the output:

$$(\{\begin{smallmatrix}x\leftarrow p\\ y\leftarrow p\end{smallmatrix}, A(p,p), A(p,p)\wedge A(q,p)\}$$
$$\{\begin{smallmatrix}x\leftarrow q\\ y\leftarrow q\end{smallmatrix}, A(q,q), A(p,q)\wedge A(q,q)\}).$$

We give a simplified description of INSTANCE omitting subtilities that have to do with the equality predicate. [$\alpha\epsilon y$ stands for: the identifier α occurs in the expression y; $S^{\alpha}_{\beta}y$ is the result of substituting α for β in the expression y; thus $S^{\alpha}_{\beta}y = y\sigma$ when $\sigma = \{\beta\leftarrow\alpha\}$.

We assume that the employed unification algorithm is willing to accept also non-literal predicate calculus formulas and so we get for example $\{z\leftarrow a\}$ when calling UNIFY((x)A(x,a), (x)A(x,z),{z})].

```
INSTANCE(T,K,VR):=
if T and K are unifyable, taking into account VR,
under substitution $\sigma$
  then return with ($\{\sigma, T\sigma, K\sigma\}$)
  else
if K = $(\alpha)(\mathrm{Form}(\alpha))$
  then U:=INSTANCE(T, Form($\alpha$), VR$\cup\{\alpha\}$)
    if U=NO then return with NO
           else (thus U=$\bigcup_i\{\sigma_i, x_i, y_i\}$)
           return $\bigcup_i\{\sigma_i, x_i, \tilde{y}_i\}$,
           where $\tilde{y}_i=(\alpha)y_i$ when $\alpha\epsilon y_i$ or else $y_i$
   else
if T = $(\exists\alpha)(\mathrm{Form}(\alpha))$
     then U:=INSTANCE(Form($\alpha$), K, VR$\cup\{\alpha\}$)
       if U=NO then return with NO
               else return $\bigcup_i\{\sigma_i, \tilde{x}_i, y_i\}$
               where $\tilde{x}_i=(\exists\alpha)x_i$ when $\alpha\epsilon x_i$ or else $x_i$
     else
if T = $(\alpha)(\mathrm{Form}(\alpha))$
     then let c be a fresh constant
          U:=INSTANCE($S^c_\alpha$Form($\alpha$), K, VR)
          if U=NO then return with NO
             else return $\bigcup_i\{\sigma_i, \tilde{x}_i, y_i\}$
             where $x_i=(\alpha)S^\alpha_c x_i$ when $c\notin y_i$
             or else $x_i$
     else
if K = $(\exists\alpha)(\mathrm{Form}(\alpha))$
     then let c be a fresh constant
        U:=INSTANCE(T, $S^c_\alpha$ Form($\alpha$), VR)
        if U=NO then return with NO
           else return $\bigcup_i\{\sigma_i, x_i, \tilde{y}_i\}$
           where $\tilde{y}_i=(\exists\alpha)S^\alpha_c y_i$ when $c\notin x_i$ or else $y_i$
     else
if K=$\vee(K_1, \ldots, K_n)$
     then return $\bigcup_i\{\sigma_i, x_i, y_i\}$where $x_i = T\sigma_i$, $y_i = K\sigma_i$
     and for all j $x_i$ is an instance of $K_j\sigma_i$ or if
     no such triple exists NO [see the appendix
     for a more detailed description of this case]
    else
if T = $\wedge(T_1, \ldots, T_n)$
     then return $\bigcup_i\{\sigma_i, x_i, y_i\}$ where $x_i = T\sigma_i$,
     $y_i = K\sigma_i$ and for all j $T_j\sigma_i$ is an instance
     of $y_i$ or if no such triple exists NO [we omit
     specification of this case since it runs
     parallel to the former case]
    else
if T = $\vee(T_1, \ldots, T_n)$
     then U: = $\emptyset$
       for all $T_j$ do
       U2:=INSTANCE($T_j$, K, VR)
       if U2 $\neq$ NO thus U2 = $\bigcup_i\{\sigma_i, x_i, y_i\}$ then
                  U: = U$\cup\bigcup_i\{\sigma_i, T\sigma_i, y_i\}$
       if U = $\emptyset$ then return NO else return U
    else
if K = $\wedge(K_1, \ldots, K_n)$
     then U: = $\emptyset$
       for all $K_j$ do
       U2: = INSTANCE(T, $K_j$, VR)
       if U2 $\neq$ NO then U: = U$\cup\bigcup_i\{\sigma_i, x_i, K\sigma_i\}$
       else return NO.
```

As mentioned in the former section INSTANCE is called in INSURER after application of step 3.2. The effectiveness of doing so was proven by the conjecture of the second example in section 5:
(x)(∃y){Λ(Equal(x,nil) ... Without INSTANCE INSURER will decompose this example into eight (8!) sub-problems of which six are redundant. When INSTANCE is incorporated the two non-redundant sub-problems only remain. INSURER and INSTANCE can also be coupled in another way as we will describe in the next section.

4. Interplay between INSURER, INSTANCE and COGITO and what is to be desired.

INSURER, INSTANCE, COGITO and the predicate calculus - conjunctive normal form translator - were embedded in a 'fixed' regime. Input for the prover consists of axioms, supporting theorems, definitions and the conjecture. For the next description we want to remind that activation of the connection graph theorem prover COGITO should be postphoned at all costs.
Roughly a supervisor triggers the following activities:

```
step 1: If the conjecture is an instance of an
        axiom, a theorem or an already proven
        theorem (see step 2) then return with
        success.
step 2: If the conjecture decomposes into the sub-
           problems C1, ..., Cn
        then for each Ci go (recursively) to step 1
              if the value returned for treating
                 Ci is succesful
              then add Ci to the collection of
                   already proven theorems
              else quit with failure
            return with success.
step 3: If the conjecture contains a predicate de-
        fined in one of the definitions then sub-
        stitute for each occurrence in the con-
        jecture the instantiated body of the de-
        finition *) and go to step 1.
step 4: Translate the axioms, supporting theorems
        and the negation of the conjecture into
        conjunctive normal form, call COGITO and
        return the value returned by it. (COGITO
        gets a resource parameter ensuring termina-
        tion).
```

The results reported in the next section have been obtained with this, slightly different, regime. Although this particular fixed connection between INSURER, INSTANCE and COGITO makes sense and is certainly effective we do not like it. We prefer considering INSURER, INSTANCE and COGITO as being members of a potentially larger family of deductive, cooperating, independent specialists. This would require the supervisor to be implemented as a multi-processes scheduler. The overall structure would be more transparant, making more easily addition of a new specialist. Not having available language as QLISP, INTERLISP and MAGMALISP prevented us of doing so.

5. Examples

Our first example looks terribly simple but a straightforward treatment after translation, by COGITO had not yet found a contradiction after generating 35 clauses. It consists out of only:
definition: $(s)(t)\{SETEQ(s,t)\leftrightarrow(x)(x\varepsilon s\leftrightarrow x\varepsilon t)\}$ and
conjecture: $(u)(v)\{SETEQ(u,v)\leftrightarrow SETEQ(v,u)\}$.
INSURER immediately recognizes that the conjecture decomposes into two sub-problems of which INSTANCE shows that one is an alphabetic variant of the other.
Remains to be proven:
$(u)(v)\{\sim SETEQ(u,v) \vee SETEQ(v,u)\}$.
After substituting away SETEQ with the definition INSURER finds again two sub-problems and - surprise - INSTANCE also finds that one is a variant of the other.
Remains this time:

$$(u)(v)\{\vee(\exists y1)(y1\varepsilon v\wedge y1\notin u)$$
$$(\exists y2)(y2\notin v\wedge y2\varepsilon u)$$
$$(y3(y3\varepsilon v\vee y3\notin u)\}.$$

Negating this conjecture and translating into C.N.F. yields four clauses. The 2nd clause generated by COGITO was already the empty one. Generating the non-solution with 35 clauses was 20 times more costly than finding this solution.

*) Recursive definitions are not allowed

The next example was taken from /Green, 4/ and was already worked on as reported in /Champeaux, 2/.
definition:
$(x)(y)\{R(x,y)\leftrightarrow(Same(x,y)\wedge Sd(y)\}$.
axioms:
(1) $(x)(y)\{Sd(y)\rightarrow Sd(merge(x,y))\}$,
(2) $(x)(y)(u)\{(Sd(y)\wedge Same(x,y))\rightarrow$
$\rightarrow Same(cons(u,x), merge(u,y))\}$,
(3) $(x)\{Equal(x,nil)\rightarrow R(x,nil)$,
(4) $(x)\{\sim Equal(x,nil)\rightarrow$
$\rightarrow Equal(x,cons(car(x)cdr(x)))\}$,
(5) $(x)(u)(v)\{(Equal(x,u)\wedge Same(u,v)\rightarrow Same(x,v)\}$.
conjecture:
$(x)(\exists y)\{(Equal(x,nil)\rightarrow R(x,y))\wedge$
$(((\sim Equal(x,nil))\wedge$
$R(cdr(x),sort(cdr(x)))\}\rightarrow$
$\rightarrow R(x,y))\}$.

The conjecture - already mentioned in section 3 - decomposes into two sub-problems of which INSTANCE finds that one is an instance of axiom(3). The remaining sub-problem was solved as well as without definition substitution (by adding the definition to the axioms) as with substitution. In both cases a contradiction was found more easily than in the non-decomposed case as can be seen from the g-penetrance values in table 1.

program and strategy	input + generated clauses	g-penetrance
QA3 /Green, 4/	286	0.091
COGITO (plain)	38 (25)	0.579 (0.680)
+ sub-problem recognition	28 (17)	0.785 (0.882)
+ definition substitution	20 (12)	0.800 (0.917)

Table 1 Showing the effectivenes of INSURER and INSTANCE. The numbers between brackets refer to values obtained when the sorted predicate calculus is used /Champeaux, 2/. (The g-penetrance is defined as # clauses in proof/#(input+generated clauses).)

6. Summary

Two algorithmic (always halting) components from the natural deduction realm are described. Both were implemented and integrated as pre-processors with a resolution type, connection graph theorem prover. Examples are given that illustrate the augmented power of this complex with respect to the sole theorem prover.

Factoring out other algorithmic components and/or adding plan/strategy generators to the pre-processor family (possibly leading to multi-level search /Sacerdoti, 7 /), we consider a more promising future task than investing more intelligence deep inside resolution and/or natural deduction theorem provers.

Excellent typing was done by Pom van der Horst.

7. References

(1) Aiello, M. & Weyrauch, R.W., Checking Proofs in the Metamathematics of First Order Logic, Stanford A.I. Lab.Memo AIM-22, August 1974.

(2) Champeaux, D., A theorem prover dating a semantic network, Proceedings of the AISB/GI Conference on A.I., pp. 82-92, Hamburg 1978.

(3) Gelernter, H. et al, Empirical Explorations of the Geometry Theorem Proving Machine, reprinted in: Computers and Thought, Ed. Feigenbaum-Feldman, McGraw-Hill, 1963, pp. 153-163.

(4) Green, C., The application of Theorem Proving to Question-answering Systems, AI Group, Technical Note 8, SRI, Stanford 1969.

(5) Loveland, D.W., Automated Theorem Proving: A Logical Basis, North-Holland, 1978.

(6) Manna, Z., Introduction to Mathematical Theory of Computation, McGraw-Hill, 1972.

(7) Sacerdoti, E., Planning in a Hierarchy of Abstraction Spaces, IJCAI3, pp. 412-422, Stanford, 1973.

Appendix

We give here a more detailed description of the sub-case
INSTANCE(T, V(K1, ..., Kn), VR).

```
INSTANCE(T, V(K1, ..., Kn), VR):=
    INSORK(({(K1, ..., Kn)(∅,T,∅}))
```

So we have now to describe the procedure INSORK. Its input consists of a non-empty list (Task1, ..., Taski, ...), where an element Taski is of the form: $\{(\tilde{K}m, \ldots\tilde{K}n)(\sigma, \tilde{T}, (\tilde{K}1, \ldots, \tilde{K}m-1))\}$, with $\tilde{K}j=Kj\sigma$, $\tilde{T}=T\sigma$. [$\tilde{T}$ is already an instance with respect to $V(\tilde{K}1, \ldots, \tilde{K}m-1)$].

```
INSORK($\overset{U}{i}$ Taski):=
    Newtask:=∅
    for each Taski do
        U:=INSTANCE($\tilde{T}$, $\tilde{K}m$, VR)
        if U≠NO  thus U=U{$\sigma_j$, $x_j$, $y_j$}
             then for each element of U do
              NN:={($\tilde{K}m+1\sigma_j$, ..., $\tilde{K}n\sigma_j$)
                   ($\sigma+\sigma_j$, $x_j$, ($\tilde{K}1\sigma_j$, ..., $\tilde{K}m-1\sigma_j$,$y_j$))}
              Newtask:=NewtaskUNN
    if Newtask=∅ then return NO
    if  m=n thus all elements Ki have been treated
          then U2:=∅
               for each element of Newtask do
               (which have the form
                 {∅,($\sigma$, $\tilde{T}$, ($\tilde{K}1$, ..., $\tilde{K}n$))})
                  U2:=U2U{$\sigma$, $\tilde{T}$, V($\tilde{K}1$, ..., $\tilde{K}n$)}
               return U2
          else return INSORK(Newtask).
```

PARTIAL PROOFS AND PARTIAL ANSWERS

Philip Klahr
The Rand Corporation
Santa Moncia, Calif. 90406

Abstract

In many cases an automatic deduction system cannot find complete proofs for particular theorems, goals, or questions requiring deductive support. In some cases information needed to complete proofs is missing from the data base. In other cases processing limits may have been reached before proofs could be completed. Rather than disregarding such partial proofs as most systems do, the DADM system displays them to users and identifies subgoals that remain unresolved. Missing information is given in the form of partial, or conditional, answers. Examples are presented to show the value and importance of such partial proofs and partial answers.

1. INTRODUCTION

In those cases when an automatic deduction system is unable to find a complete proof for a goal or theorem, there is a good deal of valuable information available in the form of partial proofs and partial answers that could be given to a user. Such partial information could provide the following types of feedback:

1. The system could identify what information is missing and needed to complete a proof.

2. The system could specify what deductive paths it has taken in its effort to find a proof.

3. If the system ran out of time, it could specify where its processing was interrupted.

In the case of missing information, the user may supply the necessary information to complete a proof, or at least be aware of what information he would need in order to obtain an answer. Certain knowledge-based systems, such as MYCIN [Shortliffe, 1976] and PROSPECTOR [Duda et al., 1978], interact with the user to extract from him needed information during deductive processing. This assumes the user has access to, or is knowledgeable about, the domain under consideration. Often times he is not. For example, he may be asking questions of a particular data base about which he has little knowledge. Giving him a conditional answer (an answer conditional on specified missing information being true) and a partial proof may satisfy his needs, as well as identifying what informational content is available in the data base and what information is absent.

In specifying what deductive paths and proofs the system has considered, the system becomes more transparent. The user can see not only how the deductive system operates, but he can also identify how the system is interpreting his original query. It may be the case, for example, that the system is exploring alternatives that the user feels are inappropriate, or that the user's formulation of the query was not what he really intended. In the former case, the user might be able to advise the system on the use of rules or facts that are particularly relevant. In the latter case, the user may reformulate his query to be more precise or consistent with the terminology in the knowledge base.

If the system exceeded its allowed processing time, the user could similarly examine the relevance of the partial proofs being constructed. In this case however, he may also be able to extend the processing limits so the system may continue along fruitful paths.

This paper demonstrates how partial proofs and partial answers are given in the DADM system [Kellogg et al., 1977, 1978], and how valuable they can be to a user. The research described here is derived from the original suggestion of adding a conditional answer capability to the CONVERSE question-answering system [Travis et al., 1973]. We will briefly overview the DADM system, before proceeding with a discussion of partial proofs.

2. OVERVIEW OF DADM

DADM (Deductively Augmented Data Management) is a natural-deduction system (Bledsoe [Bledsoe, 1977] reviews such systems) designed to interface with existing and emerging relational data base management systems. To facilitate this design cirterion, there is a distinction and separation between rules and facts. Rules (axioms, theorems, rule-based knowledge) are in the form of predicate-calculus implications and are used to perform deductions. Facts are single literals containing only constants (predicates whose arguments contain no variables), to be consistent with the form of facts typical in relational data bases.

Another design criterion is that the system should be efficient in dealing with large numbers of rules. To this end, DADM uses PATHFINDER [Klahr, 1975, 1978], a planning system designed to locate relevant rules before rules are actually applied in the course of constructing proofs. PATHFINDER uses the process of middle-term chaining to locate deductive implication chains by combining forward chaining from assumptions and backward chaining from goals. This process may be envisioned as one of generating expanded wavefronts in the two directions. (A recent proposal by Nilsson [Nilsson, 1977] suggests a similar expansion in the construction of fact and goal trees.) The purpose of middle-term chaining is not to construct proofs but to locate potentially relevant deductive implication chains through the rules.

Middle-term chaining does not operate directly on the rules. It uses a predicate connection graph which is abstracted from the rules. The predicate connection graph contains information about the deductive connections (unifications) among the rules and implication connections within rules. (The connection graph is similar to other theorem proving connection graphs, e.g., [Sickel, 1976] and [Kowalski, 1975], although here it is used as a planning tool within a natural-deduction system.) This graph is compiled when rules are first entered into the system. Thus during proof planning and proof construction, the system has knowledge about all the deductive interactions among the rules and need not compute them dynamically.

Middle-term chains form the basis of the planning process designed to focus in on potentially relevant rules. The planning process forms skeleton proofs whose variable substitutions remain to be shown consistent throughout the proof. This latter task is the function of the verifier. Verification is delayed until the system has planned out potential proofs. The verification process examines the variable substitutions involved in the deductions (unifications) of a proof to test that no variable takes on conflicting values ([Klahr, 1978]).

A middle-term chain represents a deductive implication chain through a sequence of rules. The system then examines these rules to determine if subproblems exist. If subproblems do exist, the system has three methods available to resolve them:

1. Resolve a subgoal by deduction, i.e., further application of rules to prove the subgoal.

2. Resolve a subgoal by data-base search, i.e., leave the subgoal for the data management system (DMS) to search over the file of facts.

3. Resolve a subgoal by computation, i.e., the predicate involved in the subgoal has been defined by a computational procedure which gets executed to determine the validity of the subgoal.

The particular method used is based on the predicate involved in a subgoal. If the predicate is defined by a computational procedure, that procedure is executed when the predicate occurs as a subgoal. If the predicate is defined primarily by its data-base values, i.e., knowledge about the predicate exists mostly in the fact file rather than in the rules, the predicate is left for data-base search. If a predicate is defined primarily by the rules, it is resolved by deduction. For example, the predicate GREATER-THAN would probably be defined computationally. The predicate ATTEND-CONFERENCE might be a data-base predicate, since information about the predicate is typically known or available, rather than deduced. On the other hand, the predicate KNOWS-RESULT might be a deduce predicate, since knowledge about who knows particular results is not usually available in data bases.

Each predicate has a "support indicator" that tells the system what to do with subgoals involving the predicate. This indicator is set by the data-base administrator (the person who enters rules and facts into the system). Ideally, the system should resolve a subgoal in any way that it can. If a data-base predicate cannot be resolved by data-base search, then the system should try to deduce it. However, frequent interaction between the deductive system and the DMS may be costly and time-consuming, particularly if they exist at two different sites. Thus, in the initial implementation of DADM, there is only one call on the DMS, that being when there are data-base subgoals needed to complete a proof. DADM first discovers a potential proof, then verifies the proof, and then completes the proof through the DMS (if needed).

3. MISSING INFORMATION

DADM currently deals with two types of missing information within partial proofs:

1. A particular fact needed for proof completion does not exist in the fact file.

2. A particular deduce predicate has not been resolved either because deductive support does not exist for the predicate or because DADM ran out of processing time.

We assume that the fact file is incomplete or "open." For each predicate in the data base, there are positive instances (positive facts) of that predicate and negative instances (negative facts) of the predicate. If a particular instance of the predicate does not exist in the data base, its truth value is unknown. In this respect, we differ from Reiter's closed world assumption [Reiter, 1978] that if an instance does not exist as a positive fact, treat it as false. In real-world applications involving data bases, it is rarely the case that the data base is complete. More often than not, information missing is not known to be true or false. As a result, not finding a fact (or its negation) in the data-base says nothing of its validity.

A set of search requests is sent to the DMS. These search requests are subgoals that DADM has identified as requiring data-base support. Each subgoal contains a predicate and an argument string. If the argument string contains only constants, then the DMS need only match for that fact in the data base. If the fact exists in the positive extension of the predicate, then the subgoal is resolved. If the fact exists in the negative extension, then the potential proof is invalid and DADM proceeds to the next proof plan. If the argument string contains variables, the DMS searches the data base for positive instances of the predicate that satisfy the subgoal, and returns values for the variables (which are subsequently incorporated into the proof and into the answer).

When the DMS is unable to resolve subgoals (and is unable to disconfirm them), it returns those subgoals as missing information. At this point DADM has a partial proof and a set of remaining data-base subgoals that cannot be resolved by the DMS. It then outputs this information to the user. In the initial implementation of DADM, a LISP relational data management system was written ([Klahr, 1975]) to interface with DADM. But note that no DMS is needed for DADM's operation. All data-base subgoals could simply be treated as missing information.

Thus the first type of missing information, namely missing facts, is identifiable when the DMS does not find a subgoal in the data base (nor disconfirms the subgoal). This follows neatly from the system separation of rules and facts.

The other type of missing information exists when a particular deduce subgoal (a subgoal identified as requiring deductive support) does not have deductive support. This can occur for a deduce subgoal when the system has exhausted its allowable processing time or when there are no rules that apply to the subgoal (i.e., the subgoal does not unify with any literal in any of the rules). In the former case, the system is identifying a partial proof it is working on when interrupted. In the latter case, it identifies dead ends in partial proofs. But such dead ends may still be meaningful, at least in identifying the scope of the rules, i.e., what can be derived and what cannot. It may also suggest other rules that may be meaningful and appropriate to add to the rule set.

It must be emphasized that DADM tries to find complete proofs and answers. It will find and display complete proofs before giving the user any information about partial proofs. Also, the system does not find just one proof. It will continue its deductive processing at the user's request or until it exhausts its processing limits. It is often the case in real-world applications that rules are of various degrees of plausibility. Thus several different proofs leading to the same conclusion will give much more credence to the answer derived. It is also the case that the system may be able to derive additional answers and conclusions through alternative proofs. Partial proofs are shown only after all complete proofs (within processing limits) are given.

An important concern in displaying partial proofs is deciding which proofs to output to a user, particularly when the system has generated a large number of partial proofs. It is always the case that a partial proof is verified before it is displayed. The deductions in a proof must be consistent with one another in terms of the variable substitutions required. Those partial proofs that do not successfully verify are ignored and never shown. Any partial proof that does verify may be of potential importance to a user. The system should display partial proofs that seem most relevant. The primary concern is thus the order in which partial proofs are displayed.

Two main methods are currently used for ordering partial proofs. These methods are actually used during proof construction in the planning process rather than after proofs are formed. Thus the methods apply to constructing proofs in general, partial or otherwise. These methods involve the use of advice and plausibility.

DADM allows a user to give advice on the use of rules that he feels may be particularly appropriate for deducing an answer to a query. Furthermore, a user may advise the system to key on particular predicates that may be appropriate middle-terms for chaining. In addition to problem-specific advice, a permanent advice file exists for storing general domain advice which is accessed for each query. Advice here is also on the use of particular rules and predicates. However the advice is invoked when certain specified conditions occur in the query. Conditions can refer to certain predicates occurring as assumptions or goals in the query, and to certain arguments within predicates, e.g., particular constants and domain classes of variables. (DADM also allows a user to specify negative advice. In this case, advised rules and predicates are avoided in proof generation.)

Advice is transformed into rule and predicate alert lists which are used during middle-term chaining to order and prune the predicates and rules used. The system will try chaining through advised predicates and through advised rules whenever it can. Advice thus serves as a focus of attention mechanism for the chaining and planning processes. Proofs using advised rules and predicates will be generated and displayed first.

The use of plausibility measures serves a similar focusing function. Rules have plausibility measures associated with them (similar to certainty factors in MYCIN [Shortliffe, 1976]). During middle-term chaining, rules are ordered according to their plausibilities. Resulting proofs will be generated and displayed on the basis of the plausibility of the rules used in the proof. Thus most plausible proofs will be displayed first. (Note that advice is given highest priority in the generation of proofs.)

A third method (not currently implemented) for ordering proofs concerns the number of remaining subgoals in the proof. This is particularly important for ordering the display of partial proofs. Since unresolved subgoals result in conditionals in the partial answer, the fewer the number of such conditionals, the more valuable a partial proof and partial answer will be to a user. Partial answers are conditional on the missing information being true. The greater the number of conditionals, the less likely that all of them would be true. Thus partial proofs should be ordered according to the fewest number of remaining subgoals.

4. EXAMPLES

Suppose Mike is filling out his income tax. He has recently moved from San Francisco to Los Angeles, and wants to know if he can claim moving expenses. A partial proof is given in Figure 1.

```
    MOVED(Mike,SF,LA)
          |
          |
          |
&(MOVED(x,y,z), GREATER-THAN((distance-between y z),50-miles),
                ------------------------------------------
                              compute

                              WORKS-IN(x,z,6-months))  =>  CLAIM(x,ME)
                              --------------------              |
                                    data-base                   |
                                                                |
                                                           CLAIM(Mike,ME)
```

Variable-flow Classes: (Mike,x) (SF,y) (LA,z)

Compute Subgoal: GREATER-THAN((distance-between SF LA),50-miles)

Data-base Subgoal: WORKS-IN(Mike,LA,6-months)

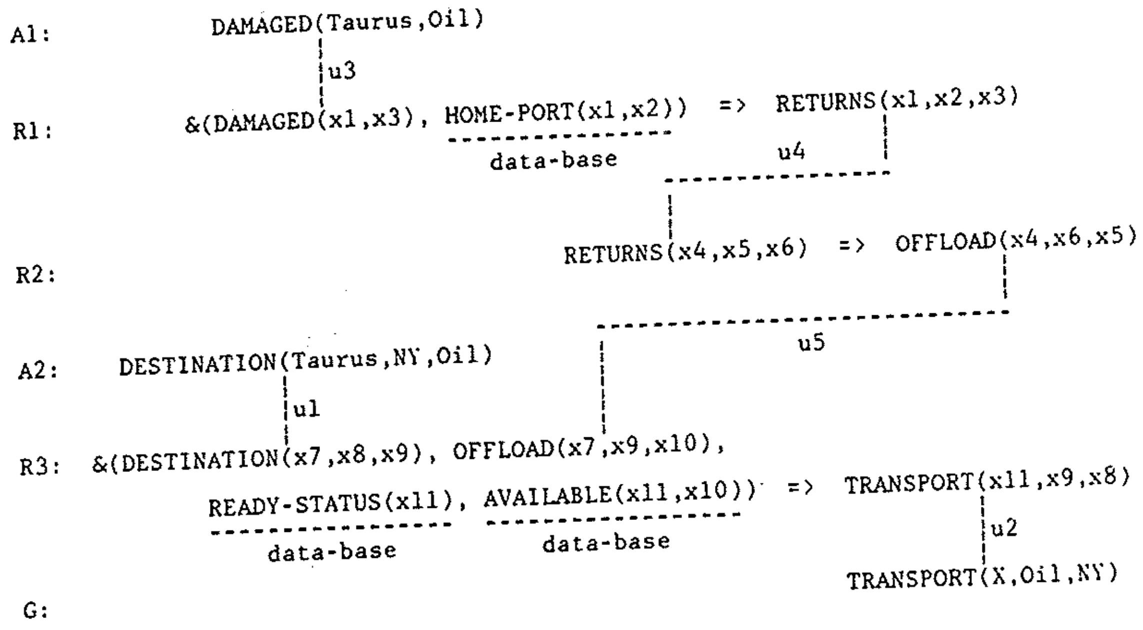
Answer: yes if WORKS-IN(Mike,LA,6-months)

Figure 1.

This example shows a simple middle-term chain through one rule from the MOVED assumption to the CLAIM goal. The vertical lines represent deductive interactions (unifications). The variables in the rule are considered universally quantified. The rule specifies what conditions are necessary for someone to claim moving expenses (ME). GREATER-THAN has been identified as a compute predicate (note the computational function distance-between as well). WORKS-IN has been identified as a data-base predicate. The variable-flow classes are formed by the verifier and indicate variable substitutions required in the partial proof. In the example assume that the computational procedure for GREATER-THAN has been defined and that it returns "true" for this subproblem. Also assume that the DMS was unable to resolve the WORKS-IN subproblem. Consequently, a conditional (partial) answer is given. Thus, even though the system was unable to discover a complete proof, it can still display relevant and important information.

Consider the more complicated partial proof in Figure 2. The query asks if the Taurus, with its oil cargo, were damaged (assumption A1) while destined for New York (assumption A2), are there any ships that could transport the oil to New York (goal G). (For this example, variable "typings" or semantic class restrictions, e.g., X being a ship, are not shown. See [McSkimin and Minker, 1977] and [Klahr, 1978] for such examples.)

The first middle-term chain involves unifications u1 and u2 through rule R3 (if a ship is destined somewhere with a cargo and offloads the cargo somewhere else and there are ships available there that are ready, then those ships can transport the cargo to the original destination). Three subgoals result. READY-STATUS and


A1:
DAMAGED(Taurus,Oil)
u3
R1:
&(DAMAGED(x1,x3), HOME-PORT(x1,x2)) => RETURNS(x1,x2,x3)
data-base
u4
R2:
RETURNS(x4,x5,x6) => OFFLOAD(x4,x6,x5)
u5
A2:
DESTINATION(Taurus,NY,Oil)
u1
R3:
&(DESTINATION(x7,x8,x9), OFFLOAD(x7,x9,x10),
READY-STATUS(x11), AVAILABLE(x11,x10)) => TRANSPORT(x11,x9,x8)
data-base
data-base
u2
TRANSPORT(X,Oil,NY)
G:

Variable-flow Classes: (Taurus,x1,x4,x7) (Oil,x3,x6,x9)
(NY,x8) (X,x11) (x2,x5,x10)

Data-base Subgoals: HOME-PORT(Taurus,x2)
READY-STATUS(X)
AVAILABLE(X,x2)

Facts Found: HOME-PORT(Taurus,Freeport)
READY-STATUS(Pisces)

Updated Variable-flow Classes: (Pisces,X,x11) (Freeport,x2,x5,x10)

Conditional Answer: Pisces if AVAILABLE(Pisces,Freeport)

Figure 2.

AVAILABLE are data-base predicates. OFFLOAD is a deduce predicate. The second middle-term chain is from the assumption A1 to the OFFLOAD subgoal and is shown by the unifications u3, u4, u5, through rules R1 (if a ship with cargo is damaged, it returns with its cargo to its home port) and R2 (if a ship returns somewhere, it unloads its cargo there). HOME-PORT results in another data-base subproblem.

Verification of the partial proof is successful. The variable-flow classes combine the substitutions required by the five unifications in the partial proof. Each variable-flow class specifies the variables and constants that must be equal for the deductions in the proof to be consistent. Verification consists of forming these classes from the substitutions involved in the unifications and determining if any variable is required to take on conflicting values. (For example, a variable-flow class containing two different constants would indicate that the variables in that class must equal two distinct values. Verification of the proof would be unsuccessful.) Those proofs or partial proofs that fail to verify are disregarded and not shown to the user.

Three data-base subgoals remain in the partial proof in Figure 2. Their arguments are updated relative to the substitutions required by the variable-flow classes. For example, the variable x1 in the HOME-PORT subgoal is in the same variable-flow class as the constant Taurus. Thus x1 is required to be Taurus in the proof. The constant is thus substituted for the variable in the subgoal before data-base search.

In this example, two facts were found in the data base. The AVAILABLE subgoal was not and consequently occurs as part of the conditional answer. Note that if none of the data-base subgoals were resolved, the conditional answer would involve all of these subgoals, i.e., those ships that are ready and available at Taurus' home port can transport the oil to New York.

Consider the very simple partial proof in Figure 3. John originates some result on learning called the Learning Method (LM). The query asks who knows about this result. The single rule used states that if a person originating a result has scientific contact with others, they will know of the result. The subgoal SCIENTIFIC-CONTACT requires deductive support. This is a partial proof since a subgoal remains unresolved. The partial answer that would be given here is SCIENTIFIC-CONTACT(w,John), i.e., those who have scientific contact with John (know about LM).

```
 ORIGINATES(John,LM)
           |
           |
&(ORIGINATES(x,y),SCIENTIFIC-CONTACT(w,x))
                  ----------------------
                         deduce

                     => KNOWS(w,y)
                              |
                              |
                        KNOWS(z,LM)
```

Figure 3.

Suppose deductive processing stops at this point. This could occur if no rules deductively support the subgoal or if processing limits were exceeded. DADM distinguishes between these two possibilities. The former is determined if the predicate connection graph does not show any unifications with the subgoal in this proof. Otherwise, if unifications do exist, further deduction is possible.

When deductive support does not exist, the user may decide that deductive limitations exist in the set of rules. This may lead him to suggest additional rules concerning scientific contact. For example, he may insert a rule stating that people attending the same conference have scientific contact, or that people visiting the laboratory in which an individual works will likely have scientific contact with that individual, etc. Thus, such a partial proof may identify missing information in the rules and suggest the insertion of new rules.

If deductive support does exist in the rules, the user may elect to have the system continue developing this proof. Time limits may be expanded, and the particular proof specified for continued deductive processing. If the user feels this particular line of deductive reasoning is inappropriate for what he originally intended, he may be able to specify that this proof is of no interest to him and should be abandoned. The interactive process between the user and the deductive system would allow a user to aid in an incremental development and expansion of proofs. To this end, the user could specify very low processing limits to allow him to examine partial proofs as they are being constructed, and advise on directions for continued deductive processing.

5. SUMMARY

Most deductive systems output only complete proofs and answers in response to input queries. Partial proofs are usually ignored. Many partial proofs, however, are relevant to deducing an answer to a query but have not been completed because of missing information or because processing limits have been exceeded. We have argued that such partial proofs and the resulting partial answers can be of significance to users for the following reasons:

1. Identify what deductions the system has discovered to show a user how the system interpreted his original query.

2. Allow a user to participate and aid in the construction of proofs by letting him examine proofs under development and allowing him to advise and select potentialy fruitful deductive paths.

3. Identify incomplete knowledge about particular predicates either in the data base of facts or in the set of rules.

4. Identify what information is needed to complete existing partial proofs and to give complete answers.

The DADM system plans skeletal proofs in an effort to find potentially relevant rules needed to answer queries. Middle-term chaining focuses on finding deductive chains leading to the desired goal or query. Subgoals are set up to be resolved. When such subgoals cannot be resolved because of missing information, partial proofs and partial answers are given.

Partial proofs are not ignored or disregarded because they represent proofs in progress. (DADM aborts only those proofs that fail to verify.) When the deductive processing is completed, these partial proofs are accessible and often valuable. Deduction systems should give the user as much information as it can or as much as is desired. They should be transparent and user-oriented. The approach taken here is a step in that direction.

Acknowledgements

Charles Kellogg and Larry Travis have been instrumental in the design of the DADM system throughout its development. They have contributed substantially to the ideas presented here.

REFERENCES

Bledsoe, W.W. Non-resolution theorem proving. Artificial Intelligence, 9, 1977, 1-35.

Duda, R.O., Hart, P.E., Nilsson, N.J., and Sutherland, G.L. Semantic network representations in rule-based inference systems. In Pattern-Directed Inference Systems D.A. Waterman and F. Hayes-Roth (Eds.), Academic Press, New York, 1978, 203-221.

Kellogg, C., Klahr, P., and Travis, L. Deductive methods for large data bases. Fifth International Joint Conference on Artificial Intelligence, 1977, 203-209.

Kellogg, C., Klahr, P., and Travis, L. Deductive planning and pathfinding for relational data bases. To appear in Logic and Data Bases, H. Gallaire and J. Minker (Eds.), Plenum, New York, 1978.

Klahr, P. The deductive pathfinder: creating derivation plans for inferential question-answering. Ph.D. Dissertation, Computer Sciences Department, University of Wisconsin, Madison, Dec., 1975.

Klahr, P. Planning techniques for rule selection in deductive question-answering. In Pattern-Directed Inference Systems, D.A. Waterman and F. Hayes-Roth (Eds.), Academic Press, New York, 1978, 223-239.

Kowalski, R. A proof procedure using connection graphs. Journal of the ACM, 22, 1975, 572-595.

McSkimin, J.R. and Minker, J. The use of a semantic network in a deductive question-answering system. Fifth International Joint Conference Artificial Intelligence, 1977, 50-58.

Nilsson, N. A production system for automatic deduction. STAN-CS-77-618 Computer Science Department, Stanford University, 1977.

Reiter, R. On closed world data bases. To appear in Logic and Data Bases, H. Gallaire and J. Minker (Eds.), Plenum, New York, 1978.

Shortliffe, E.H. Computer-Based Medical Consultations: MYCIN. American Elsevier, New York, 1976.

Sickel, S. A search technique for clause interconnectivity graphs. IEEE Transactions on Computers, C-25, 1976, 823-835.

Travis, L., Kellogg, C., and Klahr, P. Inferential question-answering: extending CONVERSE. SP-3679, System Development Corporation, Santa Monica, 1973.

REPRESENTATIONAL ECONOMY IN A MECHANICAL THEOREM-PROVER

Philip T. Cox

Department of Computer Science
University of Toronto
Toronto, Ontario
Canada M5S 1A7

ABSTRACT. A structure is described for economically implementing a theorem-prover based on deduction plans. In this structure each well-formed expression is represented only once and is shared by the proof-building, unification, and backtracking mechanisms.

1. Introduction

Because the space searched by theorem-proving programs is so large, one of the major problems that has plagued mechanical theorem-provers is lack of storage space. In particular, the early implementations of the resolution principle [12] using "level saturation" search were swamped by the proliferation of data. Their failure led to the introduction of many strategies for limiting the size of the search space [5,11]. Unfortunately, little improvement was obtained.

In the conventional list or string representation for clauses in proofs, much information is repeated. For example in the resolution:

$$\begin{tikzcd} \underline{P(x)}, Q(x) \arrow[dr, dash] & & -Q(f(y)), \underline{\underline{R(y)}} \arrow[dl, dash] \\ & \underline{P(f(y))}, \underline{\underline{R(y)}} & \end{tikzcd}$$

the underscored literals P(x) and P(f(y)), although not identical as strings, are essentially the same logical entity, as are the doubly underscored literals R(y) and R(y). Structure-sharing schemes such as that proposed by Boyer and Moore [4], solve the storage problem associated with such repetition; however, they do not <u>eliminate</u> the repetition since the repeated literals exist virtually in such representations. Several deduction systems have been proposed in which literals are not repeated [2,3,13,14]. In each of these systems, however, unification is handled in the conventional way, so the representational economy of the proof structure is eclipsed by the wastefulness of the unification algorithm.

In [6,7] a procedure is described which constructs a proof by building a graph in which no literals are repeated. In this procedure no unifiers are calculated, no substitutions are applied, and when unification failure makes backtracking necessary, a tracing algorithm determines the source of unification failure so that backtracking is exact rather than exhaustive. Although [6,7] mention the economy of graphical proofs, they do not discuss the further economy which can be obtained in implementation by sharing structure between the proof-building, unification and backtracking mechanisms. In this paper, we describe such a shared structure: the key idea is that each well-formed expression is represented only once.

2. The proof procedure

In this section we describe the construction of proofs, unification, and backtracking. For the sake of brevity, this description is incomplete and informal: for details see [6,7,8].

Throughout this paper we use most words from the literature of theorem-proving and graph-theory with their standard meanings. We do, however, use "expression" or "well-formed expression" to refer to those objects usually called "terms", as well as those usually called "literals". We also use "term" to mean an expression which is not simply a variable, and "symbol" to mean either a predicate symbol or a function symbol.

2.1 Deduction plans

The underlying structure of a deduction plan for a set $\mathscr{S}$ of clauses is a rooted tree the root of which is a special vertex called TOP. Every vertex other than TOP is a variant of some literal in some clause of $\mathscr{S}$. This underlying rooted tree corresponds exactly to a linear resolution deduction from $\mathscr{S}$ in which the only inference rule is binary resolution [10]. The rule used in building such a tree is called "replacement". Replacement is incomplete by itself, so we have two further rules, "factoring" and "reduction" which ensure completeness.

2.1.1 <u>Example</u> Consider the set of clauses:

$$\mathscr{S} = \{\{P(x,y),\ P(x,f(x)),\ -Q(x)\},\\ \{Q(x),\ R(x,a),\ -S(x)\},\\ \{S(x),\ Q(x)\},\\ \{-R(x,y),\ -S(y)\},\\ \{-P(x,f(a))\}\quad\}$$

where a is a constant. Figure 1 is a plan for this set of clauses. The first step in constructing this plan consists of selecting a clause (the "top clause") from $\mathscr{S}$ and building the graph with TOP and the literals of the top clause as vertices, and

* This research was supported by Natural Sciences and Engineering Research Council of Canada grant A3025, and was performed while the author was with the Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

with arcs labelled "SUB" from TOP to each of the other vertices. SUB stands for "subproblem" and indicates literals to be removed by replacement, reduction or factoring. Each arc labelled "REPL" indicates one application of the replacement rule, and shows that we have selected a variant $\mathcal{C}$ of some clause from $\mathcal{S}$, and have performed a binary resolution on the subproblem at the tail of the arc, using the literal of $\mathcal{C}$ which appears at the head of the arc. The remaining literals of $\mathcal{C}$ are then introduced as new subproblem vertices at the head of new SUB arcs: the tail of each of these new SUB arcs is the head of the new REPL arc. In figure 1 the REPL arcs are numbered, indicating the order in which the underlying rooted tree is constructed. A linear proof may be constructed by performing the equivalent binary resolutions in this order. Arcs labelled "RED" and "FACT" indicate applications of the reduction and factoring rules. A subproblem is said to be "open" if there are no arcs leaving it. When using plans to establish unsatisfiability, the aim is to obtain a plan with no open subproblems: such a plan is said to be "closed", and is equivalent to the empty clause in a resolution proof.

So far, we have said nothing about unification, but it is clear that to validate an application of replacement or reduction, we must show that the literal at the head of the REPL or RED arc is unifiable with the negation of the literal at the tail. Similarly the literals at each end of a FACT arc must be unifiable. We call each such pair of expressions to be unified a "constraint". Of course, we cannot simply unify each constraint: we must show that there is a substitution that simultaneously unifies all the constraints arising from the construction of a plan.

2.2 Unification

The unification algorithm operates in two stages. The first stage consists of an algorithm which manipulates three sets: a set of constraints S which is initially C, the input set of constraints; a set F which is a partition of the set of all expressions occurring in C; and a set P which is a partition of the set of all terms occurring in C. F and P are initially the partitions in which each class has one member. The algorithm deletes a constraint from S, merges the classes in F which contain the expressions in this constraint, and merges P-classes in the newly created F-class if they contain terms beginning with the same symbol. Finally, for each pair of P-classes merged, one term from each is selected, say $f(t_1,\ldots,t_n)$ and $f(s_1,\ldots,s_n)$, and the constraints $\{t_1,s_1\},\ldots,\{t_n,s_n\}$ are added to S. These actions are repeated until S is empty, at which time the algorithm halts having produced partitions F_{OUT} and P_{OUT}. If p is an expression and q is a term, we denote by [p] and <q> respectively, the class in F_{OUT} containing p and the class in P_{OUT} containing q. It happens that each class of P_{OUT} is entirely contained in some class of F_{OUT}, and if [p] = [q] where p and q are terms beginning with the same symbol, then <p> = <q>.

2.2.1 Example Given the set of constraints:

$\mathcal{C}_1$: {G(v,F(y,y)), G(s,z)}

$\mathcal{C}_2$: {u, F(y,G(s,z))}

$\mathcal{C}_3$: {F(H(w), G(x,r)), u}

$\mathcal{C}_4$: {u,v}

$\mathcal{C}_5$: {v, F(y,y)}

The algorithm produces the partitions:

$$
\begin{array}{ll}
F_{OUT} = & \{\{y \\
& \quad G(s,z),\ G(v,F(y,y)),\ G(x,r) \\
& \quad H(w)\}, \\
& \{u,v,s,z,x,r \\
& \quad F(H(w),\ G(x,r)),\ F(y,G(s,z)),\ F(y,y)\} \\
& \{w\}\}
\end{array}
$$

$$
\begin{array}{ll}
P_{OUT} = & \{\{G(s,z),\ G(v,F(y,y)),\ G(x,r)\}, \\
& \{H(w)\}, \\
& \{F(H(w),\ G(x,r)),\ F(y,G(s,z)),\ F(y,y)\}\}
\end{array}
$$

In the second stage of unification, a directed graph U, called the "unification graph", is constructed, the vertices of which are the classes of F_{OUT}. Figure 2 shows the unification graph U for the set of constraints of example 2.2.1: the reader should investigate this graph to discover how the arcs of U are constructed.

The main result concerning F_{OUT}, P_{OUT} and U is that a set of constraints C is unifiable iff every class of F_{OUT} contains at most one class of P_{OUT} and U has no cycles. Note that the set of constraints of example 2.2.1 fails to be unifiable for both reasons.

2.3 Backtracking

Having found that the set C of constraints is nonunifiable, we need to determine which constraints must be removed in order to ensure unifiability of the remainder. To do this, we investigate a directed graph A called the "automaton for C". Figure 3 illustrates the automaton for the set of constraints of example 2.2.1. The vertices (states) of the automaton are the expressions occurring in C, and again we suggest that the reader determines how the arcs are constructed by inspecting figure 3. This graph can be interpreted as a nondeterministic pushdown automaton if we regard each undirected arc as a transition in either direction under the input symbol (constraint name) which labels the arc; and if we regard each directed arc as a transition under empty input in which the arc label is pushed on to the automaton's stack if the transition is in the direction of the arrow, or popped from the stack if the transition is against the arrow. If there is some word b (i.e. sequence of constraint names) under which the automaton can reach state q with empty stack starting from state p with ---ty stack, then p and q are said to be attached (by b). Also, if there is some word b under which the automaton can reach state p with nonempty stack starting from p with empty stack, then A is said to have a loop on p (with value b). The important connections between A, F_{OUT} and U are: [p] = [q] iff p and q are attached by some word; and U has a cycle containing arc ([p], f, [q]) iff for some term t in p beginning with f, there is a loop on t in A. Clearly, if p and q are attached by some word b, and some constraint named in b is removed from C, then p and q will no longer be attached by b in the automaton of the reduced constraint set. Similarly we can remove a cycle from U by deleting some constraint named in the value of the corresponding loop in the automaton. Using the automaton, we can therefore find all unifiable subsets of a set of constraints which are maximal with respect to unifiability. First we find all pairs of terms (p,q) in different classes of P_{OUT} but in the same class of F_{OUT}; then find all words which attach p and q in A. Next we find all cycles in U and the values of the corresponding loops in A. From each of these words we create the Boolean expression which is the sum of the constraint names occurring in the word. We then simplify the product of these sums to the sum of products in which no product subsumes another or contains a repeated constraint name. If we remove from the original set of constraints all those constraints named in one product of this "covering expression", the remaining set is unifiable and is maximal with respect to this property. For the set of constraints of example 2.2.1 we obtain:

Terms	Attached by
H(w), G(s,z)	$\ell_3\ell_4\ell_5\ell_4\ell_2$
	$\ell_3\ell_4\ell_5\ell_5\ell_4\ell_2$
H(w),G(v,F(y,y))	$\ell_3\ell_2\ell_5\ell_4\ell_2\ell_1$
	$\ell_3\ell_4\ell_5\ell_5\ell_4\ell_2\ell_1$
H(w), G(x,r)	$\ell_3\ell_2\ell_5\ell_4\ell_3$
	$\ell_3\ell_4\ell_5\ell_5\ell_4\ell_3$

Terms	Loops
F(y,y)	$\ell_1\ell_2\ell_4\ell_5$
F(H(w), G(x,r))	none
F(y,G(s,z))	$\ell_2\ell_4\ell_5\ell_1$
	$\ell_2\ell_4\ell_1$

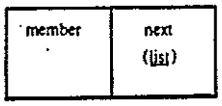
From these words we obtain the covering expression $\ell_4+\ell_1\cdot\ell_3+\ell_1\cdot\ell_5+\ell_2\cdot\ell_3+\ell_2\cdot\ell_5$ so the maximal unifiable subsets are: $\{\ell_1,\ell_2,\ell_3,\ell_5\}$, $\{\ell_2,\ell_4,\ell_5\}$, $\{\ell_2,\ell_3,\ell_4\}$, $\{\ell_1,\ell_4,\ell_5\}$, $\{\ell_1,\ell_3,\ell_4\}$.

If each constraint in a nonunifiable set of constraints arising from a deduction plan has associated with it those arcs in the plan which introduced it, then clearly for each maximal unifiable subset, we can determine how the plan must be pruned in backtracking.

3. Data structure

We now outline a data structure for implementing the above procedure. This is not intended to be a detailed description of a structure which may be directly implemented, but an informal description of the basic requirements. We are not concerned with the details of the algorithms for manipulating the structure, and hope that it is clear from the description given in section 2, how these algorithms might operate. When necessary we will discuss algorithms in general terms to clarify certain points about the structure.

In order to describe the elements of our structure, we use the common representation consisting of a string of boxes which corresponds to a set of contiguous memory locations in a machine. In defining an element we specify its type and for each box in the element, we give a name and specify the type of data the box may contain. When it is impractical to draw the contents of a box physically enclosed in the box, we will indicate its contents by a pointer; this is again a standard device which reflects the realisation of such structures in a machine. Having stated that a particular box in a particular element must contain data of a certain type, we relax this restriction for the element of type list which has the form:



The second box has the name "next" and must contain an element of type list: the first box has the name "member" and may contain any type of data; however, when a box A in some element contains a list constructed using list elements, the type of data in the "member" boxes of these list elements is specified in the definition of box A.

Throughout our explanation, we will introduce abbreviations for various structures in order to simplify subsequent diagrams. In these abbreviations some elements are collapsed to single cells: a pointer which originates in some box of such an element will be identified by being labelled with the name of the box. The single cell of a collapsed element will contain the type of the element.

As we have already mentioned, the important property of our structure is that each expression is represented only once. Consequently, an expression must be represented in such a way that it can be readily investigated by each part of the theorem-prover. To the plan-building mechanism the structure of an expression is largely irrelevant since expressions (literals) are merely names for vertices, and the applicability of a particular deduction rule is verified by the unification algorithm. In practice, of course, the plan builder should at least check the predicate symbols of the literals involved. When the algorithm which performs the first stage of unification merges two classes of P represented by $f(t_1,\ldots,t_n)$ and

$f(s_1,\ldots,s_n)$, it must have access to the expressions $t_1,\ldots,t_n$, $s_1,\ldots,s_n$ in order to build the new constraints $\{t_1,s_1\},\ldots,\{t_n,s_n\}$. It must also be able to find the classes in F and P to which an expression belongs. Finally, as states in the automaton, expressions must be connected by "push" transitions and by "input" transitions corresponding to the constraints introduced in plan-building.

To represent expressions by a structure satisfying these requirements, we define two data types as follows:

expression:

name	push	pop	trans	Fclass	Pclass
(symbol)	(push list)	(push list)	(trans list)	(Fclass)	(expression)

push:

label	head	tail
(symbol & integer)	(expression)	(expression)

For example, the representation of the expression $P(x,f(x),f(y))$ is shown in figure 4(a). Note that empty boxes in an element are crossed, and that the "name" box of an expression element corresponding to a variable is empty since names of variables are never used. The use of the "trans", "Pclass" and "Fclass" boxes of the expression element will be illustrated later.

We will henceforth use an abbreviated representation for the structure for an expression. The abbreviated form of figure 4(a) is shown in figure 4(b). In this and future abbreviated forms we adopt the conventions that an element corresponding to an arc of the plan, automaton or unification graph is represented by a dotted line connecting the elements which represent the ends of the arc, intersecting an oval containing the type of the element representing the arc.

The elements of type push correspond to the push transitions of the automaton, and may be traversed in either direction as the automaton is investigated. Also, during unification, the subexpressions of an expression may be located using the appropriate push transitions.

The input transitions of the automaton are represented in our structure by elements of type trans defined by:

trans:

label	end1	end2
(proofstep list)	(expression)	(expression)

Suppose, for example, that the automaton has two input transitions: one between $f(a)$ and $f(x)$ introduced by the construction of arc p_1 in the plan, and one between $f(a)$ and x introduced by the construction of arc p_1 and also by the construction of p_2. This is represented as in figure 5(a), and in abbreviated form in figure 5(b): in both diagrams, p_1 and p_2 stand for the proofstep elements which represent p_1 and p_2.

The "Fclass" and "Pclass" boxes of an expression element are used to maintain the partitions F and P. In the description in section 2, no variables occur in classes of P: however, in order that expressions may be treated uniformly in our structure, all the variables in a class of F are considered to comprise a class of P. The data type Fclass is defined as follows:

Fclass:

member	Uarcs
(expression list)	(arc list)

Classes in P are maintained as balanced trees using pointers from the "Pclass" boxes of expression elements [1]. For example, figure 6 illustrates a representation for the partitions

$$F = \{\{x,e_1,e_2,e_3\},\ \{y,e_4\}\}$$
$$P = \{\{e_1,e_2,e_3\},\ \{e_4\}\}$$

The expression elements are labelled with the expressions to which they correspond.

The "Uarcs" box of an Fclass element is used to point to the list of all arcs of U which leave the class of F. An arc of U is represented by an element of type arc:

tail	head	label
(Fclass)	(Fclass)	(symbol)

The unification graph of figure 7(a) for example has the representation given in 7(b) and 7(c).

Finally we define two data types for representing plans.

subproblem:

name	origin	dependents	solution
(expression)	(proofstep)	(proofstep list)	(proofstep)

proofstep:

type	tail	head	selected literal
('repl', 'red' or 'fact')	(subproblem)	(subproblem or subproblem list)	(expression)

For example, the representation of the plan of figure 8(a) is shown in figure 8(b), and its abbreviated form in figure 8(c). In these diagrams $e_1,e_2,\ldots$ etc. stand for the expression elements corresponding to these expressions. Note that there is no element corresponding to the TOP vertex of the plan, and that SUB arcs are not represented.

4. Concluding remarks

The data structure described above is a very simple version of one currently being developed for a theorem-proving program. Obviously, many things are missing from the above description: no mention has been made of constraints, for instance, nor of the form that input clauses should have. Also, many features of the structure as described are computationally inefficient; for example, the liberal use of lists built with the list element. We hope, however, that the structure described gives the reader an idea of the representational economy of the structure being implemented.

References

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1975).

[2] Andrews, P.B., *Refutations by Matings*, IEEE Transactions on Computers, vol. C-25, no.8, 801-807 (August 1976).

[3] Bibel, W. and Schreiber, J., *Proof Search in a Gentzen-like System of First-order Logic*, Proc. of International Computing Symposium, 205-212, North-Holland (1975).

[4] Boyer, R.S. and Moore, J.S., *The Sharing of Structure in Theorem-proving Programs* in Machine Intelligence 7, 101-116, John Wiley and Sons (1972).

[5] Chang, C. and Lee, R.C., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press (1973).

[6] Cox, P.T., *A Graphical Proof Procedure for First-order Logic*, Proceedings of a Conference on Theoretical Computer Science, 230-238. University of Waterloo, Waterloo, Ontario (August 1977).

[7] Cox, P.T., *Deduction Plans: a Graphical Proof Procedure for the First-order Predicate Calculus*, Ph.D. Thesis, Department of Computer Science, Research Report CS-77-28, University of Waterloo (1977).

[8] Cox, P.T., *Locating the Source of Unification Failure*, Proc. of Second Natl. Conf. of Canadian Soc. for Computational Studies of Intelligence, 20-29, Toronto (July 1978).

[9] Gotlieb, C.C., and Gotlieb, L.R., *Data Types and Structures*, Prentice-Hall (1978).

[10] Loveland, D., *A Unifying View of Some Linear Herbrand Procedures*, J. ACM 19, no.2, 366-384 (1972).

[11] Nilsson, N.J., *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill (1971).

[12] Robinson, J.A., *A Machine Oriented Logic Based on the Resolution Principle*, J.ACM 12, no.1, 23-41 (1965).

[13] Shostak, R.E., *Refutation Graphs*, Artificial Intelligence 7, 51-64 (1976).

[14] Sickel, S., *A Search Technique for Clause Interconnectivity Graphs*, IEEE Transactions on Computers, vol.C-25, no.8, 823-835 (August 1976).

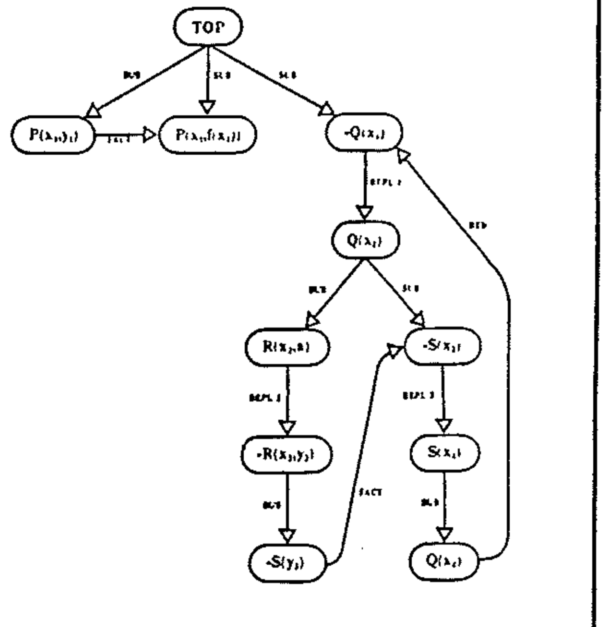
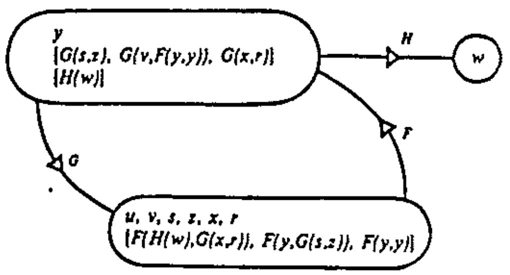


Figure 1



y
[G(s,z), G(v,F(y,y)), G(x,r)]
[H(w)]
H
w
F
G
u, v, s, z, x, r
[F(H(w),G(x,r)), F(y,G(s,z)), F(y,y)]

Figure 2

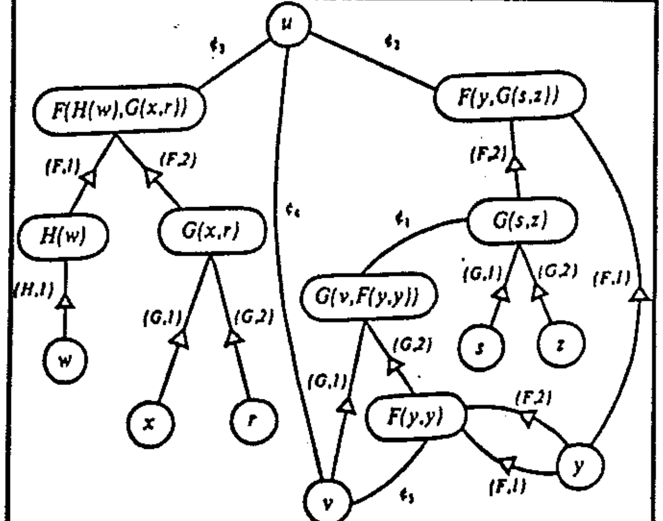


Figure 3

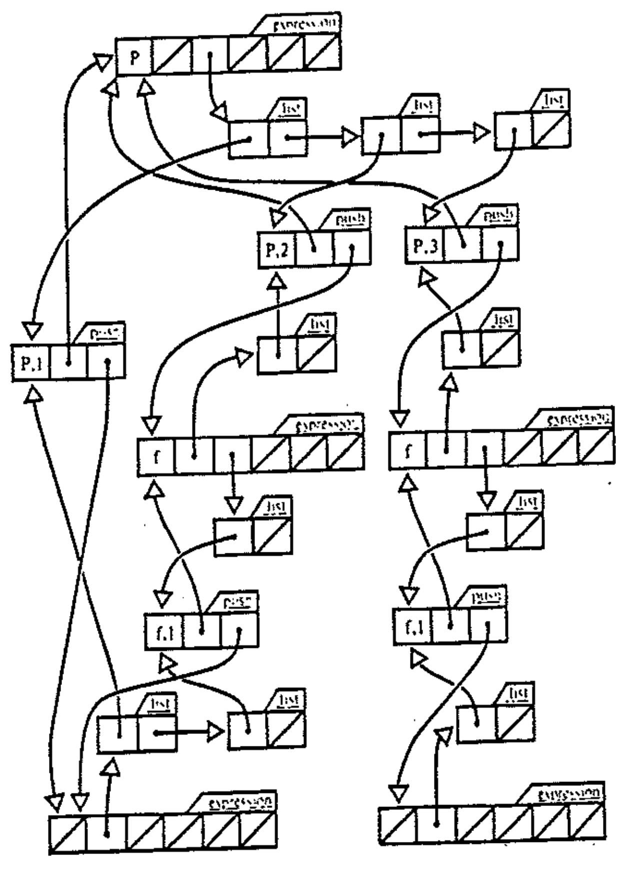


Figure 4(a)

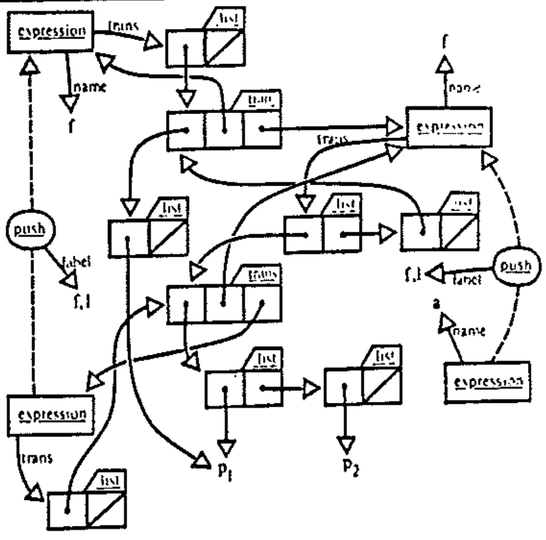


Figure 5(a)

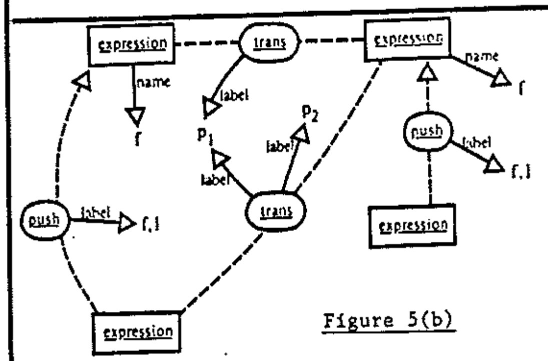


Figure 5(b)

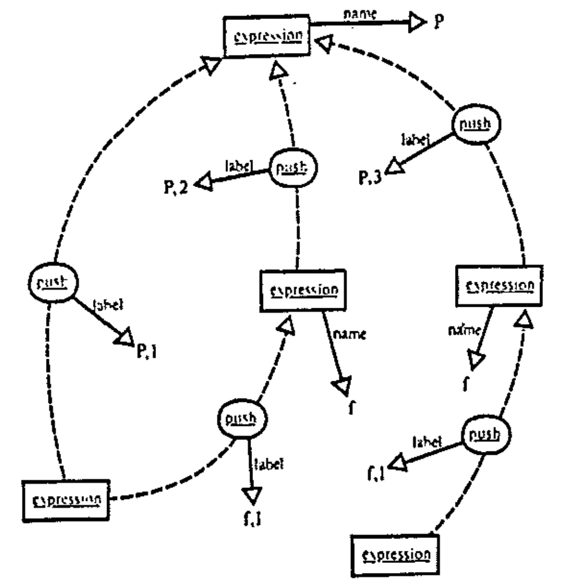


Figure 4(b)

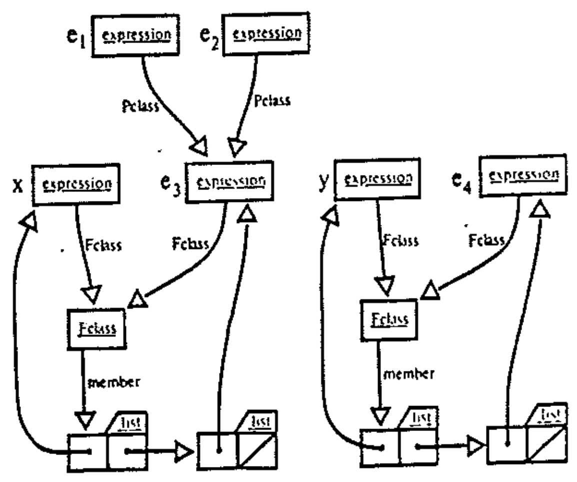


Figure 6

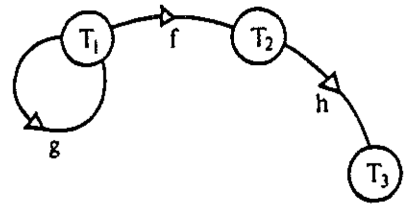


Figure 7(a)

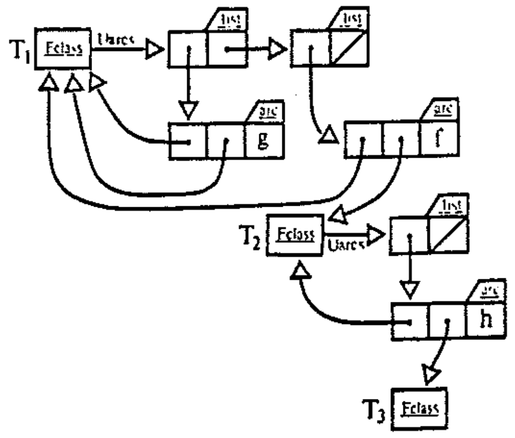


Figure 7(b)

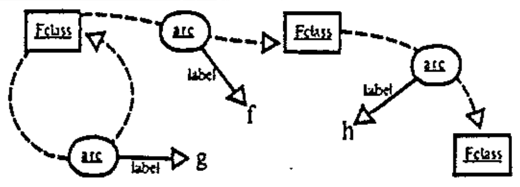


Figure 7(c)

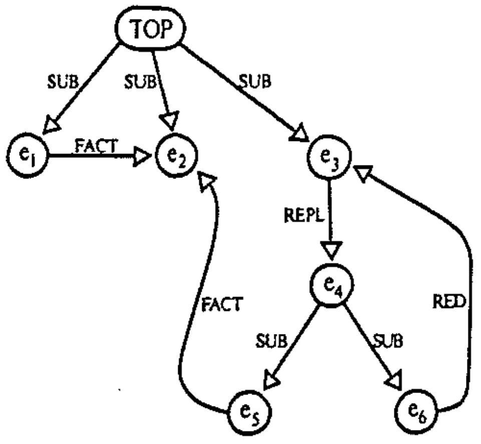


Figure 8(a)

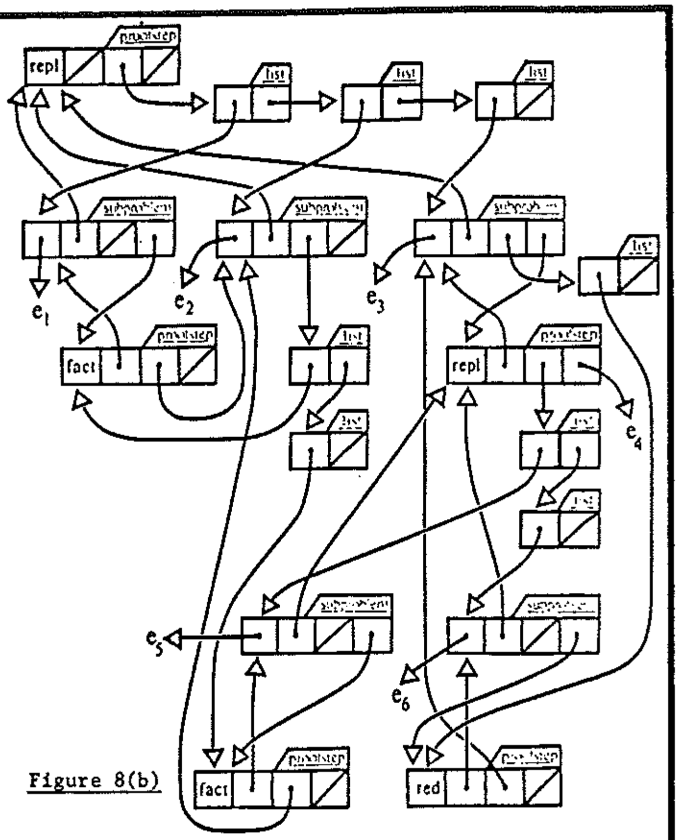


Figure 8(b)

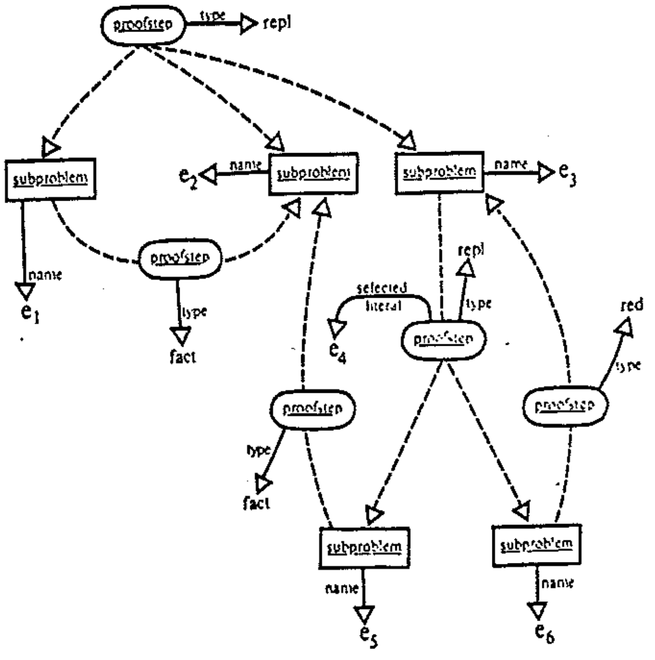


Figure 8(c)

A DEDUCTIVE APPROACH TO PROGRAM SYNTHESIS*

by

Zohar Manna and Richard Waldinger

Computer Science Dept.
Stanford University
Stanford, CA

Artificial Intelligence Center
SRI International
Menlo Park, CA

ABSTRACT

Program synthesis is the systematic derivation of a program from given specification. A deductive approach to program synthesis is presented for the construction of recursive programs. This approach regards program synthesis as a theorem-proving task and relies on a theorem-proving method that combines the features of transformation rules, unification, and mathematical induction within a single framework.

MOTIVATION

The early work in program synthesis relied strongly on mechanical theorem-proving techniques. The work of Green [1969] and Waldinger and Lee [1969], for example, depended on resolution-based theorem-proving; however, the difficulty of representing the principle of mathematical induction in a resolution framework hampered these systems in the formation of programs with iterative or recursive loops. More recently, program synthesis and theorem proving have tended to go their separate ways. Newer theorem proving systems are able to perform proofs by mathematical induction (e.g., Boyer and Moore [1975]), but are useless for program synthesis because they have sacrificed the ability to prove theorems involving existential quantifiers. Recent work in program synthesis (e.g., Burstall and Darlington [1977] and Manna and Waldinger [1977]), on the other hand, has abandoned the theorem-proving approach, and has relied instead on the direct application of transformation or rewriting rules to the program's specifications; in choosing this path, these systems have renounced the use of such theorem-proving techniques as unification or induction.

In this paper, we describe a framework for program synthesis that again relies on a theorem-proving approach. This approach combines techniques of unification, mathematical induction, and transformation rules within a single deductive system. We will outline the logical structure of this system without considering the strategic aspects of how deductions are directed. Although no implementation exists, the approach is machine-oriented and ultimately intended for implementation in automatic synthesis systems.

In the next section, we will give examples of specifications accepted by the system. In the succeeding sections, we explain the relation between theorem proving and our approach to program synthesis. This paper is an abbreviated version of a Stanford University and SRI International technical report that includes more detailed discussion and some complete examples to illustrate the application of the method.

SPECIFICATION

The specification of a program allows us to express the purpose of the desired program, without indicating an algorithm by which that purpose is to be achieved. Specifications may contain high-level constructs that are not computable, but are close to our way of thinking. Typically, specifications involve such constructs as the quantifiers *for all* ... and *for some*..., the set constructor {x: ...}, and the descriptor *find z such that*. . . .

For example, to specify a program to compute the integer square-root of a nonnegative integer n, we would write

$$\begin{aligned} sqrt(n) \Leftarrow\ & \textit{find } z \textit{ such that} \\ & integer(z) \textit{ and } z^2 \le n < (z+1)^2 \\ & \textit{where } integer(n) \textit{ and } 0 \le n. \end{aligned}$$

Here, the *input condition*

$$integer(n) \textit{ and } 0 \le n$$

expresses the class of legal inputs to which the program is expected to apply.

----

*This research was supported in part by the National Science Foundation under Grants MCS76-83655 and MCS78-02591, by the Office of Naval Research under Contracts N00014-76-C-0687 and N00014-75-C-0816, by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, and by a grant from the United States-Israel Binational Science Foundation.

The *output condition*

$$integer(z) \text{ and } z^2 \le n < (z+1)^2$$

describes the relation the output z is intended to satisfy.

Similarly, to describe a program to find the last element of a nonempty list l, we might write

```
last(l)  <=  find z such that
                for some y,  l = y<>[z]
   where islist(l) and l ≠ [].
```

Here, u<>v denotes the result of appending the two lists u and v; [u] denotes the list whose sole element is u; and [] denotes the empty list. (Thus, [A B C]<>[D] yields [A B C D]; therefore, by the above specification, *last*([A B C D]) = D.)

In general, we are considering the synthesis of programs whose specifications have the form

```
f(a)   <=  find z such that R(a, z)
   where P(a).
```

Thus, in this paper we limit our discussion to the synthesis of applicative programs, which yield an output but produce no side effects. To derive a program from such a specification, we attempt to prove a theorem of the form

for all a,
if $P(a)$
then for some z, $R(a, z)$.

The proof of this theorem must be constructive, in the sense that it must tell us how to find an output z satisfying the desired output condition. From such a proof, a program to compute z can be extracted.

BASIC STRUCTURE

The basic structure employed in our approach is the *sequent*, which consists of two lists of sentences, the *assertions* A_1, A_2, ..., A_m, and the *goals* G_1, G_2, ..., G_n. With each assertion or goal there may be associated an entry called the *output expression*. This output entry has no bearing on the proof itself, but records the program segment that has been constructed at each stage of the derivation (cf. the "answer literal" in Green [1969]).

The meaning of a sequent is that if all instances of each of the assertions are true, then some instance of at least one of the goals is true; more precisely, the sequent has the same meaning as its *associated sentence*

if for all $\bar{x}$, A_1 *and*
  for all $\bar{x}$, A_2 *and*
    .
    .
    .
  for all $\bar{x}$, A_m
then for some $\bar{x}$, G_1 *or*
    for some $\bar{x}$, G_2 *or*
    .
    .
    .
    for some $\bar{x}$, G_n

where $\bar{x}$ denotes all the free variables of the sequent. (In general, we will denote variables by $u, v, w, \ldots, z$ and constants by $a, b, c, \ldots, n$.) If some instance of a goal is true (or some instance of an assertion is false), the corresponding instance of its output expression satisfies the given specification. Note that the variable names in the sequent are "dummys;" we can systematically rename the variables in any of the assertions or goals, and the corresponding output expression, without changing the meaning of the sequent.

The distinction between assertions and goals is artificial, and does not increase the logical power of the deductive system. In fact, if we delete a goal from a sequent, and add its negation as a new assertion, we obtain an equivalent sequent; similarly, we can delete an assertion from a sequent, and add its negation as a new goal, without changing the meaning of the sequent. This property is known as *duality*. Nevertheless, the distinction between assertions and goals makes our deductions easier to understand.

If initially we are given the specification

$f(a)$ <= *find* z *such that* $R(a, z)$
where $P(a)$,

we construct the initial sequent

assertions	*goals*	*output*
$P(a)$	$R(a, z)$	z

In other words, we assume that the input condition $P(a)$ is true, and we want to prove that for some z, the goal $R(a, z)$ is true; if so, z represents the desired output. Quantifiers have been removed by the usual skolemization procedure (see, e.g., Nilsson [1971]).

The input condition $P(a)$ is not the only assertion in the sequent; typically, simple, basic axioms, such as $u = u$ or $u \cdot v \neq []$, are represented as assertions that are tacitly present in all sequents. Many properties of the subject domain, however, are represented by other means, as we shall see.

The system operates by causing new assertions and goals, and corresponding output expressions, to be added to the sequent without changing its meaning. Each time a substitution is made for a variable in an assertion or goal, the same substitution is applied to the corresponding output expression. The process terminates if the goal *true* (or the assertion *false*) is produced, whose corresponding output expression consists entirely of primitives from the target programming language; this expression is the desired program.

In the remainder of this paper we outline the deductive rules of our system.

SPLITTING RULES

The splitting rules allow us to decompose an assertion or goal into its logical components. For example, if our sequent contains an assertion of form F *and* G, we can introduce the two assertions F and G into the sequent without changing its meaning. We will call this the *andsplit rule* and express it in the following notation:

the *andsplit rule*

assertions	*goals*	*output*
F and G		t
═════		═════
F		t
G		t

We have an analogous *orsplit rule* for goals, and the *ifsplit rule*

assertions	*goals*	*output*
	if F *then* G	t
═════	═════	═════
F		t
	G	t

Note that the output entries for the consequents of the splitting rules are exactly the same as the entries for their antecedents.

TRANSFORMATION RULES

Transformation rules allow one assertion or goal to be derived from another. Typically, such rules are expressed as conditional rewriting rules:

$$r \Rightarrow s \quad \textit{if } P$$

meaning that a subexpression of form r can be replaced by the corresponding expression of form s, provided that the condition P holds. We never write such a rule unless r and s are equal terms or equivalent sentences, whenever condition P holds. For example, the transformation rule

$$u \in v \Rightarrow u = head(v) \textit{ or } u \in tail(v) \quad \textit{if } islist(v) \textit{ and } v \neq [\,]$$

expresses that an element belongs to a nonempty list if it equals the head of the list or belongs to its tail. The rule

$$u|0 \Rightarrow true \quad \textit{if } integer(u) \textit{ and } u \neq 0$$

expresses that every nonzero integer divides zero.

If a rule has the vacuous condition *true*, we write it with no condition; for example, the logical rule

$$Q \textit{ and } true \Rightarrow Q$$

and the list rule

$$head(u \cdot v) \Rightarrow u$$

may be applied to any subexpression that matches their left-hand sides.

A transformation rule

$$r \Rightarrow s \quad \textit{if } P$$

does not permit us to replace an expression of form s by the corresponding expression of form r when the condition P holds, even though these two expressions have the same values. For that purpose, we would require a second rule

$$s \Rightarrow r \quad \textit{if } P.$$

Assertions and goals are affected differently by transformation rules. Suppose

$$r \Rightarrow s \quad \textit{if } P$$

is a transformation rule and $F(r')$ is an assertion such that its subexpression r' is not within the scope of any quantifier. Suppose also that there exists a *unifier* for r and r', *i.e.*, a substitution θ such that $r\theta$ and $r'\theta$ are identical. Here, $r\theta$ denotes the result of applying the substitution θ to the expression r. We can assume that θ is a "most general" unifier (in the sense of Robinson [1965]) of r and r'. (We rename the variables of $F(r')$, if necessary, to insure that it has no variables in common with the transformation rule.) By the rule, we can conclude that if $P\theta$ holds, then $r\theta$ and $s\theta$ are equal terms or equivalent sentences. Therefore, we can add the assertion

$$\textit{if } P\theta \textit{ then } F(s)\theta$$

to our sequent.

For example, suppose we have the assertion

$$a \in l \textit{ and } a \neq 0$$

and we apply the transformation rule

$$u \in v \Rightarrow u = head(v) \textit{ or } u \in tail(v) \quad \textit{if } islist(v) \textit{ and } v \neq [\,],$$

taking r' to be $a \in l$ and θ to be the substitution $[\, u \leftarrow a;\ v \leftarrow l \,]$; then we obtain the new assertion

if islist(l) and l ≠ []
then (a = head(l) or a ∈ tail(l)) and a ≠ 0.

In general, if the given assertion $F(r')$ has an associated output entry t, the new output entry is formed by applying the substitution θ to t. For, suppose some instance of the new assertion "*if* $P\theta$ *then* $F(s)\theta$" is false; then the corresponding instance of $P\theta$ is true, and the corresponding instance of $F(s)\theta$ is false. Recall that $F(r)\theta$ and $F(r')\theta$ are identical. Then, by the transformation rule, the corresponding instance of $F(r)\theta$, *i.e.* of $F(r')\theta$, is false. We know that if any instance of $F(r')$ is false, the corresponding instance of t satisfies the given specification. Hence, because some instance of $F(r')\theta$ is false, the corresponding instance of $t\theta$ is the desired output.

In our deduction rule notation, we write

assertions	*goals*	*output*
$F(r')$		t
if $P\theta$ *then* $F(s)\theta$		$t\theta$

The corresponding dual deduction rule for goals is

assertions	*goals*	*output*
	$F(r')$	t
	$P\theta$ *and* $F(s)\theta$	$t\theta$

Transformation rules can also be applied to output entries in an analogous manner.

For example, suppose we have the goal

	$a \mid z$ *and* $b \mid z$	$z+1$

and we apply the transformation rule

$$u \mid 0 \Rightarrow \textit{true} \quad \textit{if integer}(u) \textit{ and } u \neq 0,$$

taking r' to be $a \mid z$ and θ to be the substitution $[\, z \leftarrow 0;\ u \leftarrow a \,]$. Then we obtain the goal

	(*integer*(a) *and* $a \neq 0$) *and* (*true and* $b \mid 0$)	$0+1$

which can be further transformed to

	integer(a) *and* $a \neq 0$ *and* $b \mid 0$	1

Note that applying the transformation rule caused a substitution to be made for the occurrences of the variable z in the goal and the output entry.

Transformation rules need not be simple rewriting rules; they may represent arbitrary procedures. For example, r could be an equation $f(x) = a$, s could be its solution $x = e$, and P could be the condition under which that solution applies. In general, efficient procedures for particular subtheories may be represented as transformation rules (see, e.g., Bledsoe [1977] or Nelson and Oppen [1978].) Transformation rules can also play the role of the "antecedent theorems" and "consequent theorems" of PLANNER (Hewitt [1971]).

RESOLUTION

The original resolution principle (Robinson [1965]) applied only to a sentence in conjunctive normal form. However, the ability to deal with sentences not in this form is essential if resolution and mathematical induction are to coexist happily within the same framework. The version of resolution we employ does not require the sentences to be in conjunctive normal form.

Assume our sequent contains two assertions of form $F(P_1)$ and $G(P_2)$, where P_1 and P_2 are subsentences of these assertions not within the scope of any quantifier. Suppose there exists a unifier for P_1 and P_2, i.e., a substitution θ such that $P_1\theta$ and $P_2\theta$ are identical. We can take θ to be the most general unifier. The *AA-resolution rule* allows us to deduce the new assertion

$F(true)\theta$ *or* $G(false)\theta$,

and add it to the sequent. (Here, $F(true)$ denotes the result of replacing P_1 by true in $F(P_1)$. Of course, we may need to do the usual renaming to ensure that $F(P_1)$ and $G(P_2)$ have no variables in common.) We will call θ the *unifying substitution* and $P_1\theta$ $(=P_2\theta)$ the *eliminated subexpression*; the deduced assertion is called the *resolvent*.

For example, suppose our sequent contains the assertions

if $(P(x)$ *and* $Q(b))$ *then* $R(x)$

and

$P(a)$ *and* $Q(y)$.

The two subsentences "$P(x)$ *and* $Q(b)$" and "$P(a)$ *and* $Q(y)$" can be unified by the substitution

$\theta = [\, x \leftarrow a;\ y \leftarrow b \,]$.

Therefore, the AA-resolution rule allows us to eliminate the subexpression "$P(a)$ *and* $Q(b)$" and derive the conclusion

(*if true then* $R(a)$) *or false*,

which reduces to

$R(a)$

by application of the appropriate transformation rules.

A "non-clausal" resolution rule similar to ours has been developed by Murray [1978]. Other such rules have been proposed by Wilkins [1973] and Nilsson [1977].

THE RESOLUTION RULES

We have defined the AA-resolution rule to derive conclusions from assertions:

the *AA-resolution rule*

assertions
$F(P_1)$
$G(P_2)$
$F(true)\theta$ *or* $G(false)\theta$

where $P_1\theta = P_2\theta$, and θ is most general.

By duality, we can regard goals as negated assertions; consequently, the following three rules are corollaries of the AA-resolution rule:

the *GG-resolution rule*

goals
$F(P_1)$
$G(P_2)$
$F(true)\theta$ *and* $G(false)\theta$

the *GA-resolution rule*

assertions	*goals*
$G(P_2)$	$F(P_1)$
	$F(true)\theta$ *and* (*not* $G(false)\theta$)

and a dual AG-resultion rule, where P_1, P_2, and θ satisfy the same condition as for the AA-resolution rule.

If at least one of the sentences to which a resolution rule is applied has a corresponding output expression, the resolvent will also have an output expression. If only one of the sentences has an output expression, say t, then the resolvent will have the output expression $t\theta$. On the other hand, if the two sentences $F(P_1)$ and $G(P_2)$ have output expressions t_1 and t_2, respectively, the resolvent will have the output expression

if $P_1\theta$ *then* $t_1\theta$ *else* $t_2\theta$.

For example, suppose we have derived the two goals

	$max(tail(l)) \geq head(l)$ *and* $tail(l) \neq []$	$max(tail(l))$
	not($max(tail(l)) \geq head(l)$) *and* $tail(l) \neq []$	$head(l)$

Then by GG-resolution, eliminating the subsentence $max(tail(l)) \geq head(l)$, we can derive the new goal

	(*true and* $tail(l) \neq []$) *and* (*not*(*false*) *and* $tail(l) \neq []$)	*if* $max(tail(l)) \geq head(l)$ *then* $max(tail(l))$ *else* $head(l)$

which can be reduced to

	$tail(l) \neq []$	*if* $max(tail(l)) \geq head(l)$ *then* $max(tail(l))$ *else* $head(l)$

A "polarity strategy" adapted from Murray [1978] restricts the application of the resolution rules.

MATHEMATICAL INDUCTION AND THE FORMATION OF RECURSIVE CALLS

Mathematical induction is of special importance for deductive systems intended for program synthesis, because it is only by the application of some form of the induction principle that recursive calls or iterative loops are introduced into the program being constructed. The induction rule we employ is a version of the principle of mathematical induction over a well-founded set, known in the computer-science literature as "structural induction."

We may express the principle as follows:

```
Let W be a set that is well-founded under an ordering $<$.
To show that a property $F(a)$ is true for every element $a$ of W,
prove that for an arbitrary element $a$ of W:
        if  for all $u$,
            if $u < a$ then $F(u)$
        then $F(a)$.
```

For example, if W is the set of nonnegative integers ordered by the relation $<$, the principle reduces to the familiar complete-induction principle.

Thus, in attempting to prove that a theorem of form $F(a)$ holds for each element a of a well-founded set, the well-founded induction principle permits us to assume an induction hypothesis

for all u, if $u < a$ then $F(u)$

in our effort to derive $F(a)$. In other words, we can assume inductively that the theorem holds for all u that are less than a in the well-founded ordering. In our system, the well-founded induction principle is represented by a separate deduction rule. We present only a special case of this rule here.

Suppose we are constructing a program whose specification is of form

```
$f(a)$ <=  find $z$ such that
           for some $y$, $R(a, y, z)$
  where $P(a)$.
```

Then our initial sequent is

assertions	*goals*	*output*
$P(a)$	$R(a, y, z)$	z

Then we can always add to our sequent a new assertion, the induction hypothesis

if $u < a$ *then if* $P(u)$ *then* $R(u, g(u), f(u))$		

Here, g is a new Skolem function corresponding to the variable y. The particular well-founded ordering $<$ to be employed in the proof has not yet been determined.

Let us paraphrase: We are attempting to construct a program f such that, for an arbitrary input a satisfying the input condition $P(a)$, the output $f(a)$ will satisfy the output condition $R(a, y, f(a))$, for some y. By the well-founded induction principle, we can assume inductively that for every u less than a in some well-founded ordering such that the input condition $P(u)$ holds, the output $f(u)$ will satisfy the same output condition $R(u, g(u), f(u))$, for some $g(u)$.

The above principle applies if the program f has more than one input. In this case, we consider a to be a tuple of inputs, and find an appropriate well-founded ordering over tuples.

In general, we could introduce an induction hypothesis corresponding to any subset of the assertions or goals in our sequent, not just the initial assertion and goal; most of these induction hypotheses would not be relevant to the final proof, and the proliferation of new assertions would obstruct our efforts to find a proof. Therefore, we introduce the following *recurrence strategy* for determining when to introduce an induction hypothesis.

Let us restrict our attention to the case where the induction hypothesis is derived from the initial assertion and goal. Suppose that $Q(a,\ y,\ z)$ is some subexpression of the initial goal; then that goal may be written

$$R(Q(a,\ y,\ z)).$$

Suppose further that at some point in the derivation an assertion or goal of form

$$S(Q(t,\ y',\ z'))$$

is developed, where t is an arbitrary term and y' and z' are distinct variables. In other words, the newly developed assertion or goal has a subexpression $Q(t,\ y',\ z')$ that is a precise instance of a subexpression $Q(a,\ y,\ z)$ of the initial goal. This recurrence motivates us to add the induction hypothesis

```
if $u < a$
then if $P(u)$
    then $R(Q(u,\ g(u),\ f(u)))$.
```

The rationale for introducing the induction hypothesis at this point is that now we can perform resolution between the induction hypothesis and the newly developed assertion or goal $S(Q(t,\ y',\ z'))$, eliminating the subexpression $Q(t,\ g(t),\ f(t))$.

Let us look at an example. Suppose we are constructing a program $rem(i,\ j)$ to compute the remainder of dividing a nonnegative integer i by a positive integer j; the specification may be expressed as

```
$rem(i,\ j)$ <= find $z$ such that
                for some $y$,
                $i = y \cdot j + z$ and $0 \le z$ and $z < j$
   where $0 \le i$ and $0 < j$.
```

(Note that, for simplicity, we have omitted type requirements such as *integer*(i).) Our initial sequent is then

assertions	*goals*	*outputs*
$0 \le i$ and $0 < j$	$i = y \cdot j + z$ and $0 \le z$ and $z < j$	z

Assume that during the course of the derivation we develop the goal

	$i-j = y_1 \cdot j + z$ and $0 \le z$ and $z < j$	z

This goal is a precise instance of the initial goal

$$i = y \cdot j + z \text{ and } 0 \le z \text{ and } z < j$$

obtained by replacing i by i-j. (The replacement of the dummy variable y by y_1 is not significant). Therefore, we add as a new assertion the induction hypothesis

if $(u_1, u_2) < (i,\ j)$ then if $0 \le u_1$ and $0 < u_2$ then $u_1 = g(u_1,\ u_2) \cdot u_2 + rem(u_1,\ u_2)$ and $0 \le rem(u_1,\ u_2)$ and $rem(u_1,\ u_2) < u_2$		

Here, g is a new Skolem function corresponding to the variable y, and $<$ is an arbitrary well-founded ordering. Note that $<$ is to be defined on pairs because the desired program f has a pair of inputs.

We can now apply GA-resolution between the goal

	$i-j = y_1 \cdot j + z$ and $0 \le z$ and $z < j$	z

and the induction hypothsis; the unifying substitution θ is

$$[\ u_1 \leftarrow i\text{-}j;\ u_2 \leftarrow j;\ y_1 \leftarrow g(i\text{-}j,\ j);\ z \leftarrow rem(i\text{-}j,\ j)\ ].$$

The new goal is

	true and *not* (*if* $(i-j,\ j) \prec (i,\ j)$ *then if* $0 \le i-j$ *and* $0 < j$ *then false*)	$rem(i-j,\ j)$

which reduces to

	$(i-j,\ j) \prec (i, j)$ *and* $0 \le i-j$ *and* $0 < j$	$rem(i-j,\ j)$

Note that the recursive call $rem(i-j,\ j)$ has been introduced into the output entry.

The particular well-founded ordering $\prec$ to be employed in the proof has not yet been determined. Ultimately, it is chosen to be the < ordering on the first component of the pairs, by application of the transformation rule

$$(u_1, u_2) \prec_{N1} (v_1, v_2) \Rightarrow \textit{true} \quad \text{if } u_1 < v_1 \textit{ and } 0 \le u_1 \textit{ and } 0 \le v_1.$$

A new goal

	$i-j < i$ *and* $0 \le i-j$ *and* $0 \le i$ *and* *true and* $0 \le i-j$ *and* $0 < j$	$rem(i-j,\ j)$

is produced; this goal ultimately reduces to

	$j \le i$	$rem(i-j,\ j)$

In other words, in the case that $j \le i$, the output $rem(i-j,\ j)$ satisfies the desired program's specification.

We will not discuss here the more general case, where a newly developed assertion or goal has a subexpression that is an instance of a subexpression not of the initial goal, but of some intermediate goal or assertion; this situation accounts for the introduction of "auxiliary procedures" to be called by the program under construction. We will also not discuss the case where the new subexpression is not a precise instance of the earlier subexpression, but where both are instances of a somewhat more general expression.

ACKNOWLEDGMENTS: We would like to thank Chris Goad and Neil Murray for valuable discussions.

REFERENCES:

Bledsoe, W. W. [1977], *Non-resolution theorem proving*, Artificial Intelligence Journal, Vol. 9, pp. 1-35.

Boyer, R. S. and J S. Moore [Jan. 1975], *Proving theorems about LISP functions*, JACM, Vol. 22, pp. 129-144.

Burstall, R. M. and J. Darlington [Jan. 1977], *A transformation system for developing recursive programs*, JACM, Vol. 24, No. 1, pp. 44-67.

Green, C. C. [May 1969], *Application of theorem proving to problem solving*, Proceedings of the International Joint Conference on Artificial Intelligence, Washington DC, pp. 219-239.

Hewitt, C. [Apr. 1971], *Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot*, Ph.D. thesis, MIT, Cambridge, MA.

Manna, Z. and R. Waldinger [Nov. 1977], *Synthesis: dreams ⇒ programs*, Technical Report, Computer Science Dept., Stanford University, Stanford, CA and Artificial Intelligence Center, SRI International, Menlo Park, CA.

Murray, N. [1978], *A proof procedure for non-clausal first-order logic*, technical report, Syracuse University, Syracuse, NY.

Nelson, G. and D. C. Oppen [Jan. 1978], *A simplifier based on efficient decision algorithms*, Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, Tuscon, AZ, pp. 141-150.

Nilsson, N. J. [1971], *Problem-solving methods in artificial intelligence*, McGraw-Hill Book Co., New York [pp. 165-168].

Nilsson, N. J. [Aug. 1977], *A production system for automatic deduction*, Technical Report, SRI International, Menlo Park, CA.

Robinson, J. A. [Jan. 1965], *A machine-oriented logic based on the resolution principle*, JACM, Vol. 12, No. 1, pp. 23-41.

Waldinger, R. J. and R. C. T. Lee [May 1969], *PROW: a step toward automatic program writing*, Proceedings of the International Joint Conference on Artificial Intelligence, Washington, DC, pp. 241-252.

Wilkins, D. [1973], *QUEST--a non-clausal theorem proving system*, M.Sc. thesis, University of Essex, England.

SYNTAX-DIRECTED, SEMANTICS-SUPPORTED PROGRAM-SYNTHESIS

by

Wolfgang Bibel

Institut für Angewandte Informatik und Formale Beschreibungsverfahren,
Universität Karlsruhe
on leave from Technische Universität München

Abstract. A number of strategies for the synthesis of algorithms from a given input-output specification of a problem are presented which are centered around a few basic principles. It has been verified for more than ten different algorithms that their uniform application in all cases results in a successful deductive synthesis. Many of these deductions are presented here including those of a spanning-tree algorithm, a graph-circuits algorithm, and a linear pattern-matching algorithm in strings.

Introduction

This paper is based on the view that (human or mechanical) programming is a deductive process which starts with a more or less detailed specification of the given problem in some representation language, and after some search guided by several strategies, eventually ends up with a deduction of the program which consists in the application of rules taken from logic, the underlying theory and from a knowledge base for programming skill. It also adopts the view that predicate logic (in a wider sense) is sofar still the most suitable representation language for studying this process, because of its conciseness, naturalness, flexibility extendibility - both descriptive problem specification and machine programs can be represented in logic - and in particular because of the well-understood deductive mechanisms for it.

Specifically, the topic of this paper is the search for such a deduction form a logical description of a problem to some equivalent f u n c t i o n a l form (be it in logic or in an Algol-like language). It is no secret that the search-tree for such a deduction is tremendous even for rather simple problems. Yet, it is a common observation that people are able to move in a rather direct way towards a solution even for problems that require knowledge not experienced before. Therefore, there must be some general guidelines or strategies.

Unfortunately (no, fortunately!), our human thinking is not organized such that these strategies could be easily boot-strapped from our behaviour. Therefore the only way to detect general strategies seems to be by a detailed comparative study of different deductions of different problems in order to detect common mechanisms. This has been done by the author for the well-known maximum problem and for Hoare's FIND problem and resulted in a few principles in the form of strategies [6,8] . The surprising experience a f t e r this study was that these strategies proved successful for a number of different other problems which are all discussed in more or less detail in this paper. They include such interesting algorithms such as a linear pattern-matching algorithm in strings [15], a spanning tree algorithm [1], and a graph-circuits algorithm [19].

The strategies center around four basic ideas on how to deal with a predicate logic formula describing a given problem: an $\exists$-quantifier together with a non-functional specification requires some kind of guessing (see GUESS in the next section); two (or more) consecutive $\exists$-quantifier require the evaluation of the dependency of the respective variables inherent in the specification (see DEPEND [6,8]); for both requirements the cardinality of certain problem inherent domains and other semantic knowledge provide helpful strategic information (see DOMAIN and CHVAR); the rewriting process can be strongly goal-oriented with several intermediate goals (see GET-G). This uniform and concentrated strategic concept seems to distinguish this approach from other work done in this field like [1o-14; 18; etc.].

The main concern in this paper is the synthesis of the l o g i c p a r t of the algorithms. Only for the maximum algorithm it is shown

(1) how a control [3] can easily be imposed upon it in order to obtain a recursive algorithm,

(2) how this recursive form can be transformed into a loop-form, and

(3) how the mathematically defined data can be implemented by a data structure

which is determined by the operations occurring in the algorithm. There are numerous separate studies for each of these subareas [17,4,20;9;2 etc.].

No formal system has been specified in which the natural deductions are to be formalized, assuming the reader to be aware of the fact that such systems are well-known. Rather he is expected to use his ability for natural reasoning in order to understand the deductions presented in a more or less informal way.

The main motivation of this paper is oriented towards a deeper understanding of the task of programming in general; but of course this in turn contributes to the automation of program synthesis. Therefore the reader should note that the given strategies leave a relatively small search space in each of the following examples so that it seems to be likely that a system which automatically synthesizes algorithms like ours will be realized in the near future. The text has been formulated with such a system in mind.

Basic strategies

A problem specification relates given input with the desired output; generally in such a specification the input i is restricted to those values satisfying a certain input-condition IC(i), while the output o is implicitly determined by an output-condition OC(i,o). The most appropriate corresponding logical structure seems to be a formula of the following form [3]: $\forall i \exists o(IC(i) \rightarrow OC(i,o))$. This form reflects the intention that any output is accepted for inappropriate input (for which IC(i) is false).

Example 1. Given this format a problem like the usual maximum (minimum etc.) problem reads

1.1. $\forall S \exists m(S \neq \emptyset \rightarrow m \in S \wedge S \leq m)$,

shortly $\forall S \exists m\ MAX(S) = m$,

where $\dot{\vee}$ denotes exclusive disjunction and $S \leq m :\leftrightarrow \forall x(x \in S \rightarrow x \leq m)$. Note that this definition in fact is not complete. Neither does it explicitly state the requirement of S being a set, but this implicitly is contained in $x \in S$ (assuming $\in$ as a known symbol); nor has it been stated that $<$ is an ordering relation on S, which again is implicitly contained in $S \leq m$ (assuming $\leq$ as a standard symbol).

Neither $m \in S$ nor $S \leq m$ (nor both) provide a way how to determine m for given S, not even partially. Therefore, some trial- and error method cannot be avoided. In this simple example we only can try for a hopefully correct m. Apparently it is wise to restrict such attempts somehow, specifically by requiring a subset of the conditions listed in the output-condition. The subset has to be proper since otherwise no reduction of the problem would be possible. Formally, this idea in the case of our example reads

1.2. $\forall S \forall m' \exists m(m' \in S \rightarrow$
$(S \neq \phi \rightarrow m \in S \wedge S \leq m \wedge (m \neq m' \dot{\vee} m = m')))$, or

1.3. $\forall S \forall m' \exists m(S \leq m' \rightarrow$
$(S \neq \phi \rightarrow m \in S \wedge S \leq m \wedge (m \neq m' \dot{\vee} m = m')))$

The underlying general strategy which is of a purely syntactic nature is the following.

GUESS: Tranform a problem of the form $\forall i \exists y(IC(i) \rightarrow OC(i,y))$ to $\forall i \forall y' \exists y(\text{domain-specification} \rightarrow (IC(i) \rightarrow OC(i,y) \wedge (y \neq y' \dot{\vee} y = y'))$ where domain-specification is a conjunction of a proper subset of the conjuncts in OC(i,y) with y' substituted for y.

Without any additional support the synthesis would now have to consider in our example the two (in general many) different cases, (1.2) and (1.3), possibly leading to different algorithms. In order to reduce the search space, a semantical argument in favour of one of those cases would be desirable. In our example such an argument can easily be seen, favouring (1.2) rather than (1.3) for the case of usual assumptions on the domain of S and its representation. Namely, if we keep in mind that m' has to be chosen to satisfy the domain-specification then apparently this choice can be performed immediately for $m' \in S$ while it would require a number of steps for $S \leq m'$. Moreover, if S is finite then in the first case the guess also is more likely to be correct than in the second one since $|S| < |\{m' | S \leq m'\}| = \infty$. These considerations can be generalized in the following strategy which requires semantic knowledge for its application.

For a problem as described in GUESS consider the set P of subsets of those conjuncts in OC(i,o) which specify y. By conjuncting the elements of each of these subsets, P becomes a set $\hat{P}$ of formulas. For each $C \in \hat{P}$ let $q_C := |\{y | C \text{ holds}\}|$, and s_C be the number of steps required to determine an arbitrary y such that C holds. Then for a given constant k and for $C,D \in \hat{P}$ let

$C <_k D$ iff $q_C \cdot s_C < k \cdot q_D \cdot s_D \wedge q_D \cdot s_D \nless k \cdot q_C \cdot s_C$.

With this priority-relation which captures the considerations illustrated above in a general form, the strategy reads as follows.

DOMAIN: For domain-specification in GUESS choose a minimal $C \in \hat{P}$ w.r.t. $<_k$ (which has not been considered sofar).

Apparently, all the information necessary to apply DOMAIN has to be provided to the system from outside (interactively or via a knowledge base). The weight-factor k has been included as a caution for cases where q and s behave in an opposite way in the sense of $q_C < q_D$ but $s_C > s_D$; but in all problems studied sofar this has never occurred, so the reader may well assume k=1. - Of course, application of DOMAIN to (1.1) yields $m' \in S$ as domain-specification, thus excluding (1.3) in a first attempt to synthesize MAX.

In the following we will see that (1.2) implicitly already contains an algorithmic solution for MAX which only has to be unfolded by rewriting the formula in a rather constrained goal-oriented way. The general strategic scheme and three of its instances are the following ones.

GET-G (goal-oriented equivalence transformation with goal G).
Rewrite a given formula according to domain and goal dependent equivalence transformation rules until goal G is achieved.

GET-DNF: the goal is that the resulting formula is in disjunctive normal form (alternative cases).

GET-REC: the goal is to find some kind of recursion; if necessary, generalize the problem.

GET-EP: the goal is that predicates become (more easily) evaluable.

In a program we need alternative cases. Therefore GET-DNF will be applied to the conclusion in (1.2) which yields

1.4. $m' \in S \rightarrow ((S \neq \emptyset \rightarrow m \in S \wedge S \leq m \wedge m \neq m')$
$\dot{\vee} (S \neq \emptyset \rightarrow m \in S \wedge S \leq m \wedge m = m'))$

Next we look for some kind of recursion for the first alternative (in the second one the result is already determined). The formula talks on two objects m,m' and on a set S. A knowledge-base would have to provide the information about possible forms of recursion for that kind of objects, which is the cardinality of S in this example. Therefore at this point of the process the system would merge the newly derived information $(m \neq m')$ with the clauses in the original output-condition $(m \neq S,\ S \leq m)$ on the basis of the input-condition in order to reduce the cardinality. This might start with the application of the rule $m' \in S \rightarrow (m \in S \wedge m \neq m' \leftrightarrow m \in S \setminus m' \wedge m \neq m')$. So the goal is met for the first clause by substituting $S \setminus m'$ for S. This leaves the question whether this substitution can be applied throughout the whole problem. Because of $m' \in S \rightarrow (S \leq m \wedge m \neq m' \leftrightarrow S \setminus m' \leq m \wedge m' \leq m \wedge m \neq m' \leftrightarrow S \setminus m' \leq m \wedge m' < m)$, this turns out to be true, only with the additional substitution of $m' < m$ for $m \neq m'$ resulting in

1.5. $m' \in S \rightarrow ((S \setminus m' \neq \emptyset \,\dot{\vee}\, S = \{m'\} \rightarrow m \in S \setminus m'$
$\wedge S \setminus m' \leq m \wedge m' < m) \,\dot{\vee}\, MAX(S) = m \wedge m = m')$.

Distributing the two alternatives in the input-condition leads to a contradiction for the second one which leaves

1.6. $m' \in S \rightarrow (S \setminus m' \neq \phi \rightarrow m \in S \setminus m'$
$\wedge S \setminus m' \leq m \wedge m' < m) \wedge S \neq \{m'\} \,\dot{\vee}\, MAX(S) = m$
$\wedge m = m'$

or shortly

1.7. $m' \in S \rightarrow (MAX(S \setminus m') = m \wedge m' < m \wedge S \neq \{m'\}$
$\dot{\vee}\, MAX(S) = m \wedge m = m')$

This completes the efforts of GET-REC.

Since (1.1) and (1.7) are equivalent formulas a solution for one of them will also solve the other. Now, we claim that (1.7) does represent a recursive solution for MAX. It is obtained by imposing a control [3] onto it which determines in which sequence the clauses of the formula are to be interpreted for given S and m' matching the input-condition. Let us read (1.7) in such a controlled, but informal way:
Since we assume that the problem is well-stated and has a solution, (1.7) necessarily is a true formula; if the clause $S \neq \{m'\}$ is false then only the second alternative can provide the truth, i.e. $m = m'$ is a solution; else recursively assume that m_o is a solution for $MAX(S \setminus m')$; if (after application of GET-EP) $m' < m_o$ is false then then again $m = m'$ solves the problem else $m = m_o$ provides the truth of the first alternative. Formally,

```
1.8. procedure  max(S) = m;
     choose m' such that m' ∈ S;
     if S = {m'} then return m';
     $m_o \leftarrow max(S \setminus m')$;
     if $m_o \leq m'$ then return m'
     else return $m_o$;
```

In order to improve the efficiency of this algorithm a loop-mechanism is preferable rather than a stack-mechanism for implementing this recursion. This can be easily achieved by applying a well-known result from recursion theory [9, pp. 56ff] saying that a primitive recursive form of recursion can be executed bottom-up rather than top-down, which leads to

```
1.9. procedure max'(S) = m; result ← - ∞;
     loop |S|;choose m' such that m' ∈ S;
     if result < m' then result ← m';
     S ← S\m'; end;
```

The last step in this program-development has to provide an appropriate data-structure for implementing S. This is determined by the operations involving S which occur in the algorithm. A rich knowledge base seems to be the appropriate vehicle for this part [2]. Apparently, the following would be an appropriate solution.

```
1.1o. procedure $max_o(S) = m$; $r \leftarrow -\infty$; $i \leftarrow 1$;
        loop $|S|$;
        if $r < S[i]$ then $r \leftarrow m'$; $i \leftarrow i+1$; end;
```

<u>Variants of the basic ∀∃-case</u>

The reader sofar may be convinced of the naturalness of the strategies from section 1 but still may have doubts whether a different problem would not require completely different strategies. It is the purpose of the remaining sections of this paper to verify for a number of different problems that these strategies in fact provide the basis for the synthesis of algorithms for a l l of them in the following sense: if the given problem is of the same syntactic structure like MAX i.e. if there is exactly one output-variable (of the type of objects) and if no further ∃-quantifier is created during the derivation then the algorithm results by applying e x a c t l y t h e s a m e sequence of strategies as in the previous section. Otherwise an adequate generalization or extension of this sequence has to be provided in order to reduce the problem to the basic case described sofar. The detailed presentation in the previous section will allow us to proceed in a much more concentrated way from now on.

<u>Example 2.</u> Apparently, the most difficult part in our synthesis is the GET-REC part. For a full mechanization (for a limited number of problems) more detailed research has to be invested into it. In particular, if we recall the transition from (1.4) to (1.5) we see that it requires
(1) a kind of "creative" theorem proving technique where the desired theorem may not yet be known during the rewriting process (a rich knowledge base certainly will be helpful),
(2) a rewriting technique for extending a partial match to a full match (techniques described in [16] seem to be helpful),
(3) considerable search among the formula parts (especially for larger problems) which of them to handle first (refinements of GET-REC could be helpful).
In order to demonstrate that this search mantioned in (3) may lead to various results even in our simplest problem, we mention another variant of a MAX-algorithm.

The first alternative could also be rewritten such that GET-REC requires a slight generalization of the problem eventually leading to

2.1. $m = MAX1(S,m') \wedge S \setminus \{\bar{m} | \bar{m} \in S \wedge \bar{m} \leq m'\} \neq \phi$
$\quad \dot{\vee} \; MAX(S) = m \wedge m = m'$

where $MAX(S) = MAX1(S,-\infty) = m$ and the test-predicate is assumed being easily evaluable. A correspronding algorithm approaches the maximum from below improving the intermediate result in each step.

<u>Example 3</u>. In order to illustrate the effect of strategy DOMAIN let us assume that for the MAX-problem (1.1) the situation is such that $S \leq m'$ is testable in an easier way than $m \in S$. Such a situation can be quite realistic, e.g. by assuming that all elements m' with $m' \geq S$ are labelled and that S is characterized by a complicated predicate. In this case DOMAIN would lead to formula (1.3) rather than (1.2). The main change in the subsequent process is during GET-REC where under the new condition $m \in S \wedge S \leq m \wedge m \neq m'$ can now be rewritten to $m \in S \wedge S \leq m \wedge m < m'$.
Eventually this leads to the recursion

3.1. $S \leq m' \rightarrow m = MAX2(S,m') \wedge \{\bar{m} | S \leq \bar{m} < m'\} \neq;$
$\quad \dot{\vee} \; m = MAX(S) \wedge m = m'$

where $MAX(S) = MAX2(S,\infty)$. This represents an algorithm which approaches the maximum from above. In the situation assumed for this case such an algorithm certainly would be more desirable than that from the previous section, a result which demonstrates the flexibility of this approach.

We conclude the discussion of this ∀∃-type where ∃ binds an object-variable, with referring the reader to [8, example 3] for a further example of this type where the deduction of a binary search algorithm has been given on the basis of our strategies determining a given element in an array.

<u>Example 4</u>. In [6;8] we synthesized a partitioning algorithm which partitions a given set S into two disjoint subsets S_1, S_2 such that $S_1 < a < S_2 = S \setminus (S_1 \cup a)$ for an $a \in S$. This problem is of the same syntactic structure $\forall i \exists S_1(\ldots)$ as MAX except that the output-variable S_1 varies over sets rather than objects. Apparently, in an algorithm a set can be displayed only by constructing its elements. Therefore the basic strategies GUESS and DOMAIN require a straightforward adaption to cover such a case. E.g. GUESS transforms the given formula to
$\forall i \forall y' \exists S_1(\ldots \wedge (y' \notin S_1 \,\dot{\vee}\, y' \in S_1))$, everything else remaining unchanged. This case will be exemplified in the rest of this section by two more interesting problems resulting in a spanning-tree and a circuits algorithm.

The minimum-cost spanning tree problem requires the determination of a spanning tree of minimal cost in a connected, undirected graph with a cost-function; i.e. there is given a set N of nodes, a set E of edges, and a function c: E → <reals> (see [1] for more details). Let G(N,E,c) abbreviate the formula describing the properties which characterize such a graph; similarly, let $T(N,E)$ denote the property of N and E forming a tree; let $c(E) := \sum_{e \in E} c(e)$. Then the problem formally reads

$$4.1.\quad \forall N \forall E \forall c \forall e \exists E_o (G(N,E,c) \to E_o \subseteq E \wedge T(N,E_o) \wedge \forall E_o' (E_o' \subseteq E \wedge T(N,E_o') \to c(E_o) \leq c(E_o')))$$

shortly $\forall N \forall E \forall c \exists E_o (MST(N,E,c) = E_o)$

GUESS and DOMAIN yield

$$4.2.\quad \forall N \forall E \forall c \forall e \exists E_o (e \in E \to (IC \to OC \wedge (e \in E_o \dot\vee e \notin E_o)))$$

After application of GET-DNF rewriting the first case with the new information $e \in E_o$ according to GET-REC gives

$$4.3.\quad G(N,(E\setminus e) \cup e, c) \to E_o \setminus e \subseteq E \setminus e \wedge T(N,(E_o \setminus e) \cup e) \wedge \forall E_o' (E_o' \subseteq (E \setminus e) \cup e \wedge T(N,E_o') \to c(E_o \setminus e) + c(e) \leq c(E_o')) \wedge E_o = (E_o \setminus e) \cup e$$

This together with (4.6) below suggests the recursion on a slightly more general problem, namely

$$4.4.\quad \forall N \forall E \forall E_1 \forall E_2 \forall c \exists E_o (G(N,E \cup E_1 \cup E_2,c) \to E_o \subseteq E \wedge T(N,E_o \cup E_1) \wedge \forall E_o' (E_o' \subseteq E \cup E_1 \cup E_2 \wedge T(N,E_o') \to c(E_o) + c(E_1) \leq c(E_o')))$$

shortly $\forall N \forall E \forall E_1 \forall E_2 \forall c \exists E_o (MST'(N,E,E_1,E_2 c) = E_o)$

Apparently, $MST(N,E,c) = MST'(N,E,\phi,\phi,c)$. With that abbreviation (4.3) reads

$$4.5.\quad MST(N,E,c) = MST'(N,E\setminus e,e,\phi,c) \cup e$$

The second alternative from (4.2) with the new information $e \notin E_o$ according to GET-REC gives

$$4.6.\quad G(N,(E\setminus e) \cup e,c) \to E_o \subseteq E \setminus e \wedge T(N,E_o) \wedge \forall E_o' (E_o' \subseteq (E \setminus e) \cup e \wedge T(N,E_o') \to c(E_o) \leq c(E_o')) \wedge \forall n_1 \forall n_2 (e = (n_1,n_2) \to \exists P (P(P,n_1,n_2) \wedge \forall e' (e' \in P \to e' \in E_o \wedge c(e') \leq c(e))))$$

where $P(P,n_1,n_2)$ expresses the fact that P is a path from n_1 to n_2.
The only non-trivial step of GET-REC in generating (4.6) is the merging of $e \notin E_o$ with $TR(N,E_o)$ which results in the fact that there must be a cycle in the graph since e connects two nodes of the tree without belonging to it; merging this knowledge with the minimality property in the output-condition adds the "$c(e') \leq c(e)$"-clause, since otherwise if $c(e') > c(e)$ for some $e' \in E_o$ then one could substitute e for e' in E_o resulting in a lower cost. Althogether this whole conjunct embodies a new problem namely to determine circuits in the graph [19] which here serves as a predicate for deciding between the alternatives (4.5) and (4.6). First of all, GET-EP dislikes the occurring E_o and recognizes that it can be substituted equivalently by $E\setminus e$ in the given context (for any such path not in E_o there must be one in E_o with the same properties since (N,E_o) is spanning and minimal). With this substitution (4.6) now shortly can be written as

$$4.7.\quad MST(N,E,c) = MST'(N,E\setminus e,\phi,e,c) \wedge CRC(N,E\setminus e,n_1,n_2,c) = P$$

where the circuit problem after application of GUESS and DOMAIN reads

$$4.8.\quad \forall N \forall E \forall c \forall e \forall n_1 \forall n_2 \forall \bar{e} \exists P (G(N,E,c) \wedge e = (n_1,n_2) \wedge \bar{e} \in E \setminus e \wedge c(\bar{e}) \leq c(e) \to P(P,n_1,n_2) \wedge \forall e' (e' \in P \to e' \in E \setminus e \wedge c(e') \leq c(e)) \wedge (\bar{e} \in P \dot\vee \bar{e} \notin P)$$

GET-DNF and GET-REC yields the following recursive solution of the circuit problem.

$$4.9.\quad \forall \bar{n}_1 \forall \bar{n}_2 (\bar{e} \in E \setminus e \wedge c(\bar{e}) \leq c(e) \wedge \bar{e} = (\bar{n}_1,\bar{n}_2) \to CRC = CRC(N,E\setminus\{e,\bar{e}\},n_1,m_1',c) \cup CRC(N,E\setminus\{e,\bar{e}\},n_1,m_2',c) \wedge n_1' \neq n_1 \neq n_2' \neq n_2 \neq n_1' \dot\vee CRC = CRC(N,E\setminus\{e,\bar{e}\},m_1,m_1',c) \cup \bar{e} \wedge m_2 = m_2' \wedge m_1 \neq m_1' \dot\vee n_1 = m_1' \wedge n_2 = m_2' \dot\vee CRC = CRC(N,E\setminus\{e,\bar{e}\},m_1,m_1',c))$$

where CTC is short for $CRC(N,E\setminus e,n_1,n_2,c)$ and (m_1,m_2) comprises the cases (n_1,n_2) and (n_2,n_1), and (m_1',m_2') the cases (n_1',n_2') and (n_2',n_1'). So the conclusion in (4.9) in fact consists of 9 alternative cases which represent the combinatorially possible locations of the chosen edge $\bar{e}$ with respect to the other chosen edge e, and the circuit in (N,e) consisting of path P and edge e.

Now, combining all these results gives the following solution for MST.

4.1o.
$$(4.5)\ \dot\vee\ MST(N,E,c) = MST'(N,E\setminus e,\phi,e,c) \wedge (4.9)$$

As with all our results (4.1o) is nondeterministic w.r.t. the choice of $e,\bar{e}$. But it contains the hint how to proceed best. Namely, any algorithm derived from (4.1o) by adding a control to it is forced to keep working on (4.9) and to apply (4.5) only on failure of (4.9) since (4.5) does not contain any test predicate. Because of $c(\bar{e}) \leq c(e)$ in (4.9) one would favorably choose the $\bar{e}$'s in ascending order in a bottom-up interpretation (which we always prefer) since otherwise edges would have to be chosen several times. With such a control (4.1o) represents Kruskal's algorithm [1, algorithm 5.1].

<u>Analysis of the general case</u>

In the previous sections all deductions started with a description formula involving only o n e existential quantifier.

This restriction will be dropped from now on. The simplest such case implicitly has been already mentioned at the beginning of example 4 where the partition problem actually asks for two subsets S_1, S_2. But from the problem-description it was immediately clear that one of them can be expressed by the other and the input. Such an attempt to reduce the number of output-variables would certainly preceed any further manipulation. As a strategy this represents an instance of GET.

GET-RNV: the goal is a reduced number of output-variables by expressing one of them by the others.

In [6,8] three more strategies, GET-SOC, CHVAR, and DEPEND have been introduced in order to handle the case of more than one existential quantifier where GET-RNV does not apply anymore. There they have been applied to the well-known FIND problem and produced Hoare's algorithm for it. Two more examples of this type are the duplicate problem [13] and the pattern-matching problem in strings. Only the latter one can be discussed in the space left. Knowledge of those three strategies will be required in particular for readers who not only want to understand the derivation steps but are curious to know w h y those particular steps have been chosen.

Example 5. Let us consider a string S as a function $c_S : N^{|S|} \to$ <characters>. Further assume fct as a built-in test for functionality, dom as the function which denotes the domain of a function. $c_S|_D$ be the restriction of c_S to domain D and N_m^n the natural numbers k with $m \leq k \leq n$. Then the definition of the pattern-matching problem can be stated as

5.1. $\forall c_S \forall c_P \exists k (\mathrm{fct}\, c_S \wedge \mathrm{dom}\, c_S = N_1^n \wedge \mathrm{fct}\, c_P \wedge \mathrm{dom}\, c_P = N_1^m \wedge \forall k \in N_o^{n-m} \wedge \forall j (j \in N_1^m \wedge c_P(j) = c_S(j+k))$

shortly $\forall c_S \forall c_P \exists k (PM(c_S, c_P) = k)$

In a moment we will see that this in fact is a problem of the present general type. GUESS and DOMAIN gives

5.2. $\forall c_S \forall c_P \forall k' \exists k (k' \in N_o^{n-m} \to (IC \to OC \wedge (k \neq k' \dot{\vee} k = k'))$

which is equivalent with

5.3. $\forall c_S \forall c_P \forall k' \exists k (k' \in N_o^{n-m} \to (IC \to OC \wedge k \neq k' \wedge \exists \bar{j} (\bar{j} \in N_1^m \wedge c_P(\bar{j}) \neq c_S(\bar{j}+k'))$
$\dot{\vee} (IC \to OC \wedge k = k'))$

Formula (5.3) has two existential quantifiers. In a first attempt to solve this problem GET-SOC [8] would notice that no single literal in the range of $\exists \bar{j}$ does contain k. Therefore the $\exists \bar{j}$-subproblem can be solved independently with our basic mechanism. After insertion of this solution into (5.3) the problem now can be solved in turn with the mechanism producing the simplest and well-known algorithm which moves the pattern in any sequence (e.g. from left to right) to all possible positions, and for each of them independently checks whether the match succeeds or fails which requires a quadratic number of comparisons.

If we want to obtain a smarter result, strategy GET-SOC has to be switched off activating the general subproblem mechanism [6;8] . As will be seen in the following derivation, the solution of the $\exists \bar{j}$-subproblem provides valuable information for the original problem, the use of which causes the improvement.

The $\exists \bar{j}$-quantifier in (5.3) has been created by the first one; therefore this is not the situation for which CHVAR is required since naturally $\bar{j}$ depends from k. DEPEND gives the subproblem

5.4. $\forall c_S \forall c_P \forall k' \forall j \forall k \exists \bar{j} (IC1 \to \bar{j} \in N_1^m \wedge c_P(\bar{j}) \neq c_S(\bar{j}+k')),$

$IC1 \equiv IC \wedge k' \in N_o^{n-m} \wedge j \in N_1^m \wedge k \in N_o^{n-m} \wedge c_P(j) = c_S(j+k) + k \neq k'$

GUESS and DOMAIN yields

5.5. $\forall c_S \forall c_P \forall k' \forall j \forall k \forall \bar{j}' \exists \bar{j} (\bar{j} \in N_1^m \to (IC1 \to c_P(\bar{j}) \neq c_S(\bar{j}+k') \wedge (\bar{j} \neq \bar{j}' \dot{\vee} \bar{j} = \bar{j}'))$

In the first alternative apparently we can add $c_P(\bar{j}') = c_S(\bar{j}'+k')$. Also note that

5.6. $c_P(j) \neq c_P(\bar{j}') \wedge j \neq \bar{j} \wedge c_P(j) = c_S(j+k) \wedge c_P(\bar{j}') = c_S(\bar{j}'+k') \leftrightarrow$
$c_P(j) \neq c_P(\bar{j}') = c_S(\bar{j}'+k') = c_S(j+k'+\bar{j}-j) \wedge c_P(j) = c_S(j+k) \wedge k \neq k'+\bar{j}'-j \wedge j \neq \bar{j}'$

5.7. $c_P(j) = c_P(\bar{j}') \wedge c_P(j) = c_S(j+k) \wedge k \neq k' \wedge c_P(\bar{j}) \neq c_S(\bar{j}+k') \wedge \bar{j}' = \bar{j} \wedge j \neq \bar{j}' \leftrightarrow$
$c_P(\bar{j}') = c_P(j) = c_S(j+k) = c_S(\bar{j}'+k+j-\bar{j}') \neq c_S(\bar{j}'+k') \wedge k \neq k'+\bar{j}'-j \wedge \bar{j}' = \bar{j} \wedge j \neq \bar{j}'$

Applying these equivalences GET-REC yields

5.8. $\bar{j}' \in N_1^m \to (IC1(c_S, c_P|_{N_1^m \setminus \bar{j}'}, k', j, k) \to$
$c_P(\bar{j}) \neq c_S(\bar{j}+k') \wedge \bar{j} \neq \bar{j}' \wedge k \neq k'+\bar{j}'-j \wedge c_P(\bar{j}') = c_S(\bar{j}'+k') \wedge c_P(j) \neq c_P(\bar{j}') \wedge \bar{j}' \neq j$
$\dot{\vee} (IC1(c_S, c_P|_{N_1^m \setminus \bar{j}'}, k', j, k) \to$
$c_P(\bar{j}) \neq c_S(\bar{j}+k') \wedge \bar{j} \neq \bar{j}' \wedge c_P(\bar{j}') = c_S(\bar{j}'+k') \wedge c_P(j) = c_P(\bar{j}')$
$\dot{\vee} (IC1 \to c_P(\bar{j}) \neq c_S(\bar{j}+k') \wedge \bar{j} = \bar{j}' \wedge c_P(j) \neq c_P(\bar{j}') \wedge j \neq \bar{j}')$
$\dot{\vee} (IC1 \to c_P(\bar{j}) \neq c_S(\bar{j}+k') \wedge \bar{j} = \bar{j}' \wedge c_P(j) = c_P(\bar{j}') \wedge j \neq \bar{j}' \wedge k \neq k'+\bar{j}'-j)$

$$\dot{\vee}(IC1 \rightarrow c_p(\bar{j}) \neq c_S(\bar{j}+k') \wedge \bar{j} = \bar{j}'$$
$$\wedge c_p(j) = c_p(\bar{j}') \wedge j = \bar{j}')$$

So we see that the rewriting process manipulates also the domain of possible values for k. This suggests to consider a function $MISM(c_S, c_p, k', j, k, PSH) = (\bar{j}, PSH')$ which results in a pair consisting of the argument on which the mismatch occurs and the new domain of potential shifts. With this abbreviation we get

5.9. $\bar{j}' \in N_1^m \rightarrow MISM(c_S, c_p, k', j, k, PSH) =$
$MISM(c_S, c_p|_{N_1^m \setminus \bar{j}'}, k', j, k, PSH \setminus \{k'+j'-j\})$
$\wedge c_p(j) \neq c_p(\bar{j}') \wedge j \neq \bar{j}'$
$\dot{\vee} MISM(c_S, c_p, k', j, k, PSH) =$
$MISM(c_S, c_p{}_{N_1^m \setminus \bar{j}'}, k', j, k, PSH)$
$\wedge c_p(j) = c_p(\bar{j}')$
$\dot{\vee} MISM(c_S, c_p, k', j, k, PSH) = (\bar{j}', PSH)$
$\wedge (c_p(j) \neq c_p(\bar{j}') \wedge j \neq \bar{j}' \dot{\vee}$
$c_p(j) = c_p(\bar{j}') \wedge j = \bar{j}')$
$\dot{\vee} MISM(c_S, c_p, k', j, k, PSH) =$
$(\bar{j}', PSH \setminus \{k'+\bar{j}-j\}) \wedge c_p(j) = c_p(\bar{j}') \wedge j \neq \bar{j}'$

Let us agree on the notation $expr._i$ to obtain the i-th component of an n-tuple expr. Then we have

5.1o. $k' \in N_o^{n-m} \rightarrow (IC \rightarrow$
$k \in N_o^{n-m} \wedge \forall j (j \in N_1^m \rightarrow c_p(j) = c_S(j+k)) \wedge k \neq k'$
$\wedge \forall j (MISM(c_S, c_p, k', j, k, N_o^{n-m}) = (\bar{j}, PSH)$
$\rightarrow c_p(\bar{j}) \neq c_S(\bar{j}+k') \wedge k \in PSH))$
$\dot{\vee} (IC \rightarrow OC \wedge k = k')$

Again the manipulation on the domain of possible values of k suggests to embed our problem into a more general one with PSH as an additional variable $GPM(c_S, c_p, PSH) = (k, PSH')$. From the previous formula we get with this abbreviation

5.11. $k' \in PSH \rightarrow GPM(c_S, c_p, PSH) =$
$GMP(c_S, c_p, MISM(c_S, c_p, k', j, k, PSH)._2)$
$\wedge (MISM(c_S, c_p, k', j, k, PSH)._1 = \bar{j}$
$\rightarrow c_p(\bar{j}) \neq c_S(\bar{j}+k'))$
$\dot{\vee}\ GMP(c_S, c_p, PSH) = (k, PSH')$

5.12. $PM(c_S, c_p) = GPM(c_S, c_p, N_o^{n-m})._1$

As with (1.7) for MAX these formulas can now be easily interpreted in a recursive way. Also as in MAX, the resulting procedures for MISM and GPM can be easily transformed into loops.

Again the resulting form leaves open the sequence of choosing $k', \bar{j}'$ and j. Therefore we are free to choose $\bar{j}'$ and j such that $c_p(j) \neq c_p(\bar{j}')$ as long as the pattern does not consist of a one-character string for which case there is a simple linear algorithm anyway. With that provision PSH decreases at each step of the algorithm which proves the linearity of the algorithm. Unfortunately, we have not the space to go into a more detailed study on different sequences of choosing $k', \bar{j}'$ and j, and on the connection with the algorithms in [15]. So we conclude just with the remark that the sequence $k' = 0, \bar{j}' = 1, \bar{j}' = 1,2,3,4;\ k' = 2, \ldots$ leads exactly to the same shifts as in [15] for the example discussed there in detail in section 1.

Summary

This paper presented the successful application of a few basic strategies to a number of interesting programming problems. They take into account both the syntactic structure of the problem description and relevant semantic information, with syntax being of primary importance. Because of the uniform structure of the derivations there is some hope that this approach might be also a first step towards a classification of algorithms.

Apparently, these strategies seem to considerable cut down the search space for an algorithm without predetermining the outcome in an unfavourable or uncontrolled way. The resulting derivations guarantee the correctness of the derived algorithms (favouring the synthesis rather than the verification approach to programming [4, section 5]). An intermediately resulting representation of the algorithm describes its logic which still can be interpreted with different controls which is in accordance with the logic programming philopsophy [3;17]. In this paper in fact we essentially restricted our intentions to obtain such a logic representation of the resulting algorithm.

This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG).

Typescript: G. Weiher

References

[1] Aho,A.V., Hopcroft,J.E., Ullman,J.D., The design and analysis of computer algorithms, Addison-Wesley,Reading,1974

[2] Barstow,D.R., Automatic construction of algorithms and data structures using a knowledge base of programming rules, Ph.D.thesis, Stanford AIM-3o8, 1977.

[3] Bibel,W., Programmieren in der Sprache der Prädikatenlogik, Techn.Universität München, 1975; short version: Prädikatives Programmieren, Lect.Notes Comp. Sc.33, Springer,Berlin-New York, 274-283, 1975.

[4] Bibel,W., Synthesis of strategic de-

finitions and their control, Report 761o, Techn.Universität München, FB Mathematik, 1976.

[5] Bibel,W., A uniform approach to programming, Report 7633, Techn.Universität München, FB Mathematik, 1976.

[6] Bibel,W., On strategies for the synthesis of algorithms, Proc.AISB/GI conf.on Artif.Intell., Hamburg, D. Sleeman (ed.), Leeds University, U.K., 22-27, 1978.

[7] Bibel,W., Deduktion von Algorithmen, forthcoming report, Universität Karlsruhe.

[8] Bibel,W., Furbach,U., Schreiber J., Strategies for the synthesis of algorithms, Proc. 5th GI-conf.on Programming Languages, K.Alber (ed.), Informatik Fachberichte 12, Springer, Berlin-New York, 97-1o9, 1978.

[9] Brainerd,W.S., Landweber,L.H., Theory of computation, Wiley, 1974.

[1o] Clark,K., Predicate Logic as an computational formalism, Ph.D.thesis, Imperial College London, forthcoming.

[11] Darlington,J.L., A synthesis of several sorting algorithms, to appear in Acta Informatica.

[12] Green,C., A summary of the PSI program synthesis system, 5IJCAI, Cambridge, 38o, 1977.

[13] Hogger,C.J., Program synthesis in predicate logic, Proc. AISB/GI conf. on Artif.Intell., Hamburg, D.Sleeman (ed.), Leeds Univ., U.K.,22-27, 1978.

[14] Hogger,C.J., Derivation of logic programs, Ph.D.thesis, forthcoming.

[15] Knuth,D.E., Morris,J.H., Pratt,V.R., Fast pattern matching in strings, SIAM J. Comput. 6, 323-35o, 1977.

[16] Kodratoff,Y., Fargues,J., A sane algorithm for the synthesis of LISP functions from example problems, Proc. AISB/GI conf.on Artif.Intell., Hamburg, D. Sleeman (ed.), Leeds Univ., U.K., 169-175, 1978.

[17] Kowalski,R., Algorithm = Logic+control, Report, Imperial College, 1976.

[18] Manna,Z., Waldinger,R., The automatic synthesis of recursive programs, Proc. Symp.on Artif.Intell. and Programming Languages, Rochester, ACM, 29-36, 1977.

[19] Mateti,P., Deo,N., On algorithms for enumerating all circuits of a graph, SIAM J.Comput., 5, 9o-99, 1976.

[2o] Warren,D., Implementing Prolog, Report 39, Dept. Artif.Intell., Edinburgh, 1977.

A Proof Procedure for Higher-Order Modal Logic

GRAHAM WRIGHTSON

INSTITUT FÜR INFORMATIK I

UNIVERSITÄT KARLSRUHE

75 KARLSRUHE 1

FEDERAL REPUBLIC OF GERMANY

Abstract A mechanizable proof procedure for higher-order modal logic is presented. The procedure is based upon consistency trees (also called tableau by Beth and Hintikka). Deduction rules, a routine and examples are presented.

Introduction

One form of intensional logic is the modal sort in which notions of necessity and possibility are treated. Apart from the philosophical [14] and linguistic [12] interest in modality there has been a considerable increase in its applications during recent years, particularly within the information sciences e.g. in computational linguistics [1,4] and program verification [6,9,13]. Some of these systems use higher-order modal logic but almost without exception there is no mechanized proof procedure - neither for first-order nor for higher-order modal logic. In this paper such a procedure is presented after introducing the higher-order modal logic upon which it is founded. There then follow the rules which are used in the procedure, a routine which forms a type of strategy for applying the rules and a few examples to illustrate the method.

Higher-Order Modal Logic

The version of higher-order modal logic ML is similar to that presented in [3], pp.67-75. One of [5,10] should be consulted for an intuitive explanation of the modal notions of necessity (the necessity operator is notated as $\Box$) and possibility (the possibility operator is notated as $\Diamond$). A resumé of the essential features relevant to the proof procedure is now given.

The Syntax of ML

The types of ML include only those for individuals of type e and predicates (relations) at various levels of type $(\sigma_0,\sigma_1,\dots,\sigma_{n-1})$ with n arguments of which the first is an object of type σ_0, the second an object of type σ_1, etc. Hence the set P of predicate types is the smallest set such that

(i) $e \in P$,

(ii) $\sigma_0,\sigma_1,\dots,\sigma_{n-1} \in P$ imply

$$(\sigma_0,\sigma_1,\dots,\sigma_{n-1}) \in P.$$

Primitive Symbols. For each $\sigma \in P$ we have a denumerable list of *variables*

$$x^0_\sigma, x^1_\sigma, x^2_\sigma, \dots$$

and non-logical *constants*

$$c^0_\sigma, c^1_\sigma, c^2_\sigma, \dots$$

of type σ together with the improper symbols $\sim, \rightarrow, \wedge, \vee, \leftrightarrow, \forall, \exists, \Box, \Diamond, [,]$. A symbol s_σ of type σ is a variable or constant of type σ.

An *atomic formula* of ML is an expression of the form

$$ss^0s^1\dots s^{n-1},$$

where s is of type $\sigma = (\sigma_0,\sigma_1,\dots,\sigma_{n-1})$ and s^k is a symbol of type σ_k for $k < n$.

The *formulas* of ML are generated in the usual way from the atomic formulas by the connectives $\sim, \rightarrow, \wedge, \vee, \leftrightarrow$, the quantifiers $\forall x_\sigma, \exists x_\sigma$, where x_σ is an arbitrary variable, and the modal connectives $\Box$ and $\Diamond$.

The Semantics of ML

Let D and I be non-empty sets. A *frame* for ML based on D and I is an indexed family $(M_\sigma)_{\sigma \in P}$ of sets, where

(i) $M_e = D$,

(ii) For each type $\sigma = (\sigma_0,\ldots,\sigma_{n-1})$, M_σ is a non-empty subset of $\mathcal{P}(M_{\sigma_0} \times \ldots \times M_{\sigma_{n-1}})^I$.

A *general model* of ML based on D and I is a system $M = (M_\sigma, m)_{\sigma \in P}$ such that

(i) $(M_\sigma)_{\sigma \in P}$ is a frame for ML based on D and I,

(ii) The mapping m assigns to each constant c_σ an element of M_σ.

As(M) denotes the set of all *assignments* over the general model M, i.e. all functions a on the set of variables such that $a(x_\sigma) \in M_\sigma$ for each variable x_σ. For an assignment a, let $\bar{a}$ be the extension of a to the set of all constants, such that $\bar{a}(c_\sigma) = m(c_\sigma) \in M_\sigma$.

A formula A is *satisfied* in a model M with respect to an index $i \in I$ and assignment $a \in As(M)$ (written M,i,a sat A) when, by recursive definition on the formula A

(i) M,i,a sat $ss^0\ldots s^{n-1}$ iff $(\bar{a}(s^0),\ldots,\bar{a}(s^{n-1}))$ is an element of $\bar{a}(s)(i)$,

(ii) the usual satisfaction clauses for $\sim, \rightarrow, \wedge, \vee, \leftrightarrow$, $\forall x_\sigma$ and $\exists x_\sigma$,

(iii) M,i,a sat $\Box$A iff M,j,a sat A for all $j \in I$,

(iv) M,i,a sat $\Diamond$A iff M,j,a sat A for some $j \in I$.

Thus from (iii) the modal logic presented here is S5, where the binary accessibility relation R between two indicies i and j is reflexive, symmetric and transitive.

The *modal axioms* for S5 are the following

(i) $\Box A \rightarrow A$,

(ii) $\Box[A \rightarrow B] \rightarrow [\Box A \rightarrow \Box B]$,

(iii) $\sim \Box A \rightarrow \Box \sim \Box A$.

In what follows the expresion *tree* will often be used. A formal definition is not given as it would lead away from the main theme, however, the idea will become clearer from the examples.

Those who wish to have a more explicit elaboration of its meaning should consult one of [7,11,16]. The procedure given here is similar to [15].

Some Definitions for the Procedure

In the rules and the routine for applying the rules the following definitions hold:

A *formula* is a formula of ML.

A *forest* is a particular configuration of formulas,trees,paths and alternative-world trees which has been constructed by applications of the routine.

A *path* is a sequence of formulas in a tree such that the origin (the formula at the top of the tree) is in every path which is in the tree and such that every formula which is below the origin is a successor of some previous formula.

A path p has *access to* an alternative-world tree t just in case both

(i) the origin of t is a formula of the form A only if p contains a formula of the form $\Diamond$A and

(ii) for every formula of the form $\Box$B which is contained in p, there is a formula of the form B whichis contained in every path of t.

A *path is closed* just in case it contains both a formula of the form A and a formula of the form $\sim$A.

A *tree is closed* just in case either every path in that tree is closed or every open path in that tree has access to at least one closed tree.

A *path is open* if it is not closed.

In forests, paths do not extend into any alternative-world trees to which the paths have access,i.e. if p is a path in a tree t, then p is not a path in any tree to which p has access.

$(\ldots n_\sigma \ldots)$ is the formula which results from replacing all occurrences of x_σ which are free in the formula $(\ldots x_\sigma \ldots)$ with an occurrence of n_σ which is free in $(\ldots n_\sigma \ldots)$.

The Rules for the Procedure

1.Always take the highest (closest to the origin) formula when applying the rules.

2. *Denials*

(a) Erase "$\sim\sim$" (double negation) whereever it appears in unchecked formulas in open paths.

(b) Check formulas of the form $\sim\forall x_\sigma(\ldots x_\sigma\ldots)$ in open paths and rewrite them as $\exists x_\sigma\sim(\ldots x_\sigma\ldots)$ at the bottom of each open path in which the checked formula occurs.

(c) Check formulas of the form $\sim\exists x_\sigma(\ldots x_\sigma\ldots)$ in open paths and rewrite them as $\forall x_\sigma\sim(\ldots x_\sigma\ldots)$ at the bottom of each open path in which the checked formula occurs.

(d) Check formulas of the form $\sim\Box A$ and $\sim\Diamond A$ in open paths and rewrite them as $\Diamond\sim A$ resp. $\sim\Box A$ at the bottom of each open path in which the checked formula occurs.

3. *Truth-Functional Connectives*

Apply the following rules to any formula having the form AoB or $\sim$ AoB, where o is either $\to, \wedge, \vee$ or $\leftrightarrow$, in open paths; check the formula and write the result of applying the rule at the bottom of each open path in which the formula occurs.

(a) ✓
$$\begin{tikzcd} & A \to B \arrow[dl, dash] \arrow[dr, dash] & \\ \sim A & & B \end{tikzcd}$$

(b) ✓
$$\begin{tikzcd} & A \vee B \arrow[dl, dash] \arrow[dr, dash] & \\ A & & B \end{tikzcd}$$

(c) ✓
$$\begin{array}{c} A \wedge B \\ \\ A \\ B \end{array}$$

(d) ✓
$$\begin{tikzcd} & A \leftrightarrow B \arrow[dl, dash] \arrow[dr, dash] & \\ A & & \sim A \\ B & & \sim B \end{tikzcd}$$

(e) ✓
$$\begin{array}{c} \sim(A \to B) \\ \\ A \\ \sim B \end{array}$$

(f) ✓
$$\begin{array}{c} \sim(A \vee B) \\ \\ \sim A \\ \sim B \end{array}$$

(g) ✓
$$\begin{tikzcd} & \sim(A \wedge B) \arrow[dl, dash] \arrow[dr, dash] & \\ \sim A & & \sim B \end{tikzcd}$$

(h) ✓
$$\begin{tikzcd} & \sim(A \leftrightarrow B) \arrow[dl, dash] \arrow[dr, dash] & \\ \sim A & & A \\ B & & \sim B \end{tikzcd}$$

4. *Necessities*

For each formula of the form $\Box A$ in an open path, check the formula and write A at the bottom of each open path in which $\Box A$ occurs.

5. *Alternative-World Necessity Rule*

For every alternative-world tree which is being formed at the bottom of an open path in which a formula of the form $\Box A$ occurs, write A at the bottom of every open path in those alternative-world trees, unless it is already in the path, and erase the check which is beside $\Box A$.

6. *Possibilities*

For each formula of the form $\Diamond A$ in an open path begin to form an alternative-world tree at the bottom of every open path in which $\Diamond A$ occurs by writing A at the origin of each of those alternative-world trees. Check $\Diamond A$.

7. *Universal Quantifiers*

Given an open path in which a formula of the form $\forall x_\sigma(\ldots x_\sigma\ldots)$ occurs, then for each constant n_σ that appears free anywhere in the path or in a path in an alternative-world tree to which the path has access, write the formula $(\ldots n_\sigma\ldots)$ at the bottom of the path in which $\forall x_\sigma(\ldots x_\sigma\ldots)$ occurs unless that formula already occurs in the path. If no constant of type σ occurs free in the path or in some open path in an alternative-world tree to which the path has access, choose some constant n_σ and write $(\ldots n_\sigma\ldots)$ at the bottom of the path. Do not check $\forall x_\sigma(\ldots x_\sigma\ldots)$.

8. *Existential Quantifiers*

Given an open path in which a formula of the form $\exists x_\sigma(\ldots x_\sigma\ldots)$ occurs, inspect the path to see if it contains a formula of the form $(\ldots n_\sigma\ldots)$, where n is some constant of type σ. If it does not contain some formula of this sort then choose a constant n_σ that is not free anywhere in the path and write the formula $(\ldots n_\sigma \ldots)$ at the bottom of the path. When this has been done for every open path in which $\exists x_\sigma(\ldots x_\sigma \ldots)$ occurs, check that formula.

9.*S5 Necessity*
For every formula of the form $\Box A$ which is in t', where t' is an alternative-world tree to tree t, if $\Box A$ is not a member of that path of t which has access to t', write $\Box A$ at the bottom of that path and write A at the bottom of every open path in t.

The Routine for the Procedure

```
1.List at the origin of the tree the premises and
  the negation of the conclusion, go to 2.
2.Is there a formula unchecked in an open path in
  the forest to which one of the rules for denial
  applies?
  Yes: Apply it and go to 2.
  No: Close all paths containing a formula and
      its denial and go to 3.
3.Are all paths in the original tree closed?
  Yes: Stop. The argument is valid.
  No: Go to 4.
4.Is there a formula in an open path to which the
  S5 necessity rule applies?
  Yes: Apply it, go to 4.
  No: Go to 5.
5.Is there an unchecked formula in an open path
  to which the rule for necessities applies?
  Yes: Apply it and go to 2.
  No: Go to 6.
6.Is there an unchecked formula in an open path
  to which one of the rules for truth-functional
  connectives applies?
  Yes: Apply it and go to 2.
  No: Go to 7.
7.Is there an unchecked formula in an open path
  to which the rule for existential quantifiers
  applies?
  Yes: Apply it and go to 7.
  No: Go to 8.
8.Is there a formula in an open path to which the
  rule for universal quantifiers applies?
  Yes: Apply it and go to 9.
  No: Go to lo.
9.Has there been any change in the forest since
  last entering 8?
  Yes: Go to 8.
  No: Go to lo.
lo.Has there been any change in the forest since
  last entering stage 2?
  Yes: Go to 2.
  No: Go to 11.
11.Is there a formula unchecked in an open path
  to which the rule for possibilities applies?
  Yes: Apply it and go to 12.
  No: Go to 12.
12.Is there a checked formula in an open path to
  which the alternative-world necessities rule
  applies?
  Yes: Apply it and go to 12.
  No: Go to 13.
13.Have all formulas of the form □A been checked
  in every open path at the bottom of which an
  alternative-world tree is being formed?
  Yes: Go to 14.
  No: Check them and go to 13.
14.Has there been any change in the forest since
  last entering stage 2?
  Yes: Go to 2.
  No: Go to 15.
15.Does every open path have access to at least
  one alternative-world tree which is closed?
  Yes: Stop. The argument is valid.
  No: Stop. The argument is invalid.
```

Some Explanations

An explanation of the modal aspects of the procedure will be given, while the non-modal aspects, as mentioned before, can be reviewed in one of

[7,11,16]. Formal proofs of the completeness and soundness of the procedure, which are along the lines of [2,8], are extremely long (see [17] for proofs) and space limits do not permit them here.

By *change in the forest* is meant the introduction of a new path or alternative-world tree or addition of a formula to a path.

Whileever a formula is *checked* a rule cannot be applied to it.

If t'' is an *alternative-world tree being formed at the bottom of an open path* of tree t' and if t' is an alternative-world tree which has been formed at the bottomof an open path of tree t, then t'' is analternative-world tree which is being formed at the bottom of an open path of t.

The denial rule for $\sim \Box A$ and $\sim \Diamond A$ is simply an application of the definition of the possibility operator from the necessity operator, namely, to say that a formula is necessarily true is equivalent to saying that it is not possible that the formula is false. Hence,

$$\Box A \leftrightarrow \sim \Diamond \sim A \text{ and}$$
$$\Diamond A \leftrightarrow \sim \Box \sim A.$$

The necessities rule comes from the definition that $\Box A$ is satisfied in world i, iff A is satisfied in all $j \in I$, hence A is satisfied in i.

The alternative-world necessity rule reflects the definition that the accessibility relation R is transitive.

The possibilities rule is justified by the definition of satisfaction for $\Diamond A$ in a world i. There must be another world j in which A is satisfied.

To say that a tree is closed is to say that there is no assignment in any world such that either (i) all paths are closed i.e. a formula is either satisfied or not satisfied in each path or (ii) each path has access to at least one closed tree i.e. access to a world $j \in I$.

Examples

1. Show that the following instance of the Barcan formula $\forall x \Box P(x) \rightarrow \Box\forall x\, P(x)$ is valid in S5.

1.✓	$\sim(\forall x \Box P(x) \rightarrow \Box\forall x\, P(x))$
2.	$\forall x \Box P(x)$
3.✓	$\sim \Box\forall x\, P(x)$
4.✓	$\Diamond \sim\forall x\, P(x)$
5.✓	$\Box P(a)$
6.✓	$\sim\forall x\, P(x)$
7.	$P(a)$
8.✓	$\exists x \sim P(x)$
9.	$\sim P(a)$
	closed

Line 1 is the negation of the conclusion. Lines 2 and 3 result from line 1 by rule 3e. Line 4 results from line 3 by rule 2. Line 5 comes from line 2 by the universal quantifiers rule and line 6 from line 4 by the possibilities rule. Line 7 is from line 5 by the necessities rule and line 8 from line 6 by the denials rule 2b. Line 9 is from line 8 by existential quantification.

As can be seen, only first-order has been used and so the types of the variables and constants were not noted. An example illustrating higher-order is now given.

2. Show that $\forall S\ (q(S) \rightarrow S(m))$, $\forall I\ (k(I) \rightarrow q(I))$ are premises for the conclusion $\forall P\ (k(P) \rightarrow P(m))$, whereby m is a constant of type e, I,S,P are variables of type (e) and k,q are constants of type ((e)).

1.	$\forall S\ (q(S) \rightarrow S(m))$		
2.	$\forall I\ (k(I) \rightarrow q(I))$		
3.✓	$\sim\forall P\ (k(P) \rightarrow P(m))$		
4.✓	$\exists P \sim(k(P) \rightarrow P(m))$		
5.✓	$\sim(k(p) \rightarrow p(m))$		
6.✓	$q(p) \rightarrow p(m)$		
7.✓	$k(p) \rightarrow q(p)$		
8.	$k(p)$		
1o.	$\sim p(m)$		
11.	$\sim k(p)$	$q(p)$	
12.		$\sim q(p)$	$p(m)$

All branches are closed. Constant p is of type (e).

3. Show the validity of the formula in line 1, whereby r' and r'' are constants of type e, p' and p'' constants of type (e), q a constant of type ((e)), r and p are variables of types e and (e) respectively.

1.√ $\sim(\Diamond\exists p\ q(p) \vee \Diamond\exists p\exists r\ p(r)) \leftrightarrow \Diamond\exists p\exists r\ (q(p) \vee p(r))$

2.√ $\Diamond\exists p\ q(p) \vee \Diamond\exists p\exists r\ p(r)$

3.√ $\sim\Diamond\exists p\exists r\ (q(p) \vee p(r))$

6.√ $\Box\sim\exists p\exists r\ (q(p) \vee p(r))$

7.√ $\sim\exists p\exists r\ (q(p) \vee p(r))$

8. $\forall p\sim\exists r\ (q(p) \vee p(r))$

9.√ $\Diamond\exists p\ q(p)$

19.√ $\sim\exists r\ (q(p') \vee p'(r))$

23. $\forall r\ \sim(q(p') \vee p'(r))$

26.√ $\sim(q(p') \vee p'(r))$

29. $\sim q(p')$

30. $\sim p'(r')$

33.√ $\exists p\ q(p)$

34.√ $\sim\exists p\exists r\ (q(p) \vee p(r))$

35. $\forall p\sim\exists r\ (q(p) \vee p(r))$

36. $q(p'')$

37.√ $\sim\exists r\ (q(p'') \vee p''(r))$

38. $\forall r\ \sim(q(p'') \vee p''(r))$

39. $\sim(q(p'') \vee p''(r''))$

40. $\sim q(p'')$

41. $\sim p''(r'')$

closed

10.√ $\Diamond\exists p\exists r\ p(r)$

20.√ $\sim\exists r\ (q(p') \vee p'(r))$

24. $\forall r\ \sim(q(p') \vee p'(r))$

27.√ $\sim(q(p') \vee p'(r))$

31. $\sim q(p')$

32. $\sim p'(r')$

42.√ $\exists p\exists r\ p(r)$

43.√ $\sim\exists p\exists r\ (q(p) \vee p(r))$

44. $\forall p\sim\exists r\ (q(p) \vee p(r))$

45.√ $\exists r\ p''(r)$

46. $p''(r'')$

47.√ $\sim\exists r\ (q(p'') \vee p''(r))$

48. $\forall r\ \sim(q(p'') \vee p''(r))$

49. $\sim(q(p'') \vee p''(r))$

50. $\sim q(p'')$

51. $\sim p''(r'')$

closed

4.√ $\sim(\Diamond\exists p\ q(p) \vee \Diamond\exists p\exists r\ p(r))$

5.√ $\Diamond\exists p\exists r\ (q(p) \vee p(r))$

11.√ $\sim\Diamond\exists p\ q(p)$

12.√ $\sim\Diamond\exists p\exists r\ p(r)$

13.√ $\Box\sim\exists p\ q(p)$

14.√ $\Box\sim\exists p\exists r\ p(r)$

15.√ $\sim\exists p\ q(p)$

16.√ $\sim\exists p\exists r\ p(r)$

17. $\forall p\ \sim q(p)$

18. $\forall p\ \sim\exists r\ p(r)$

21. $\sim q(p')$

22.√ $\sim\exists r\ p'(r)$

25. $\forall r\ \sim p'(r)$

28. $\sim p'(r')$

52.√ $\exists p\exists r\ (q(p) \vee p(r))$

53.√ $\sim\exists p\ q(p)$

54.√ $\sim\exists p\exists r\ p(r)$

55. $\forall p\ \sim q(p)$

56. $\forall p\sim\exists r\ p(r)$

57.√ $\exists r\ (q(p'') \vee p''(r))$

58.√ $q(p'') \vee p''(r)$

59. $\sim q(p'')$

60.√ $\sim\exists r\ p''(r)$

61. $\forall r\ \sim p''(r)$

62. $q(p'')$

closed

63. $p''(r'')$

64. $\sim p''(r'')$

closed

References

[1] Brown,F.M. and C.B.Schwind: Outline of an Integrated Theory of Natural Language Understanding, D.A.I.Research Report No.5o, Uni. of Edinburgh, March 1978.

[2] Deussen, P.: Analytische Tableau, Manuskript, Inst. für Informatik I, Uni. Karlsruhe,1975.

[3] Gallin,D.: Intensional and Higher-Order Modal Logic, North-Holland/American Elsivier, 1975.

[4] Hobbs,J.R. and S.J.Rosenschein: Making Computational Sense of Montague's Intensional Logic, Artificial Intelligence, Vol.9,No.3, Dec.1977.

[5] Hughes,G.E. and M.J.Cresswell: An Introduction to Modal Logic, Methuen, 1968.

[6] Janssen,T.M.V. and P.van Emde Boas: On the Proper Treatment of Referencing,Dereferencing and Assignment, Preprint ZW 94/77, Dept. of Pure Mathematics, Uni. of Amsterdam, 1977.

[7] Jeffrey,R.C.: Formal Logic:Its Scope and Limits, McGraw-Hill,1967.

[8] Kripke,S.A.: Semantical Analysis of Modal Logic I,Normal Modal Propositional Calculi, Zeitschrift für math. Logik und Grundlagen der Mathematik,Bd.9,S.67-96 (1963).

[9] Kröger,F.: LAR:A Logic of Algorithmic Reasoning, Acta Informatica 8, 243-266 (1977).

[10] Leblanc,H.: Truth-Value Semantics, North-Holland, 1976.

[11] Leblanc,H. and W.A.Wisdom: Deductive Logic, Allyn and Bacon, 1972.

[12] Montague,R.: Formal Philosophy, R.Thomason (ed.), Yale Uni. Press, 1974.

[13] Pratt,V.R.: Semantical Considerations on Floyd-Hoare Logic, 17.Symposium Foundations of Computer Science, IEEE, Oct. 1976.

[14] Prior,A.: Modal Logic, in Encyclopaedia of Philosophy, P.Edwards (ed.), Macmillan,1967.

[15] Slaght,R.L.: Modal Tree Constructions,Notre Dame Jour. Formal Logic,Vol.XVIII,No.4,1977.

[16] Smullyan,R.M.: First-Order Logic,Springer, 1968.

[17] Wrightson,G.: Completeness and Soundness of of a Higher-Order Modal Proof Procedure, Int. Bericht, Fakultät Informatik,Uni.Karlsruhe,1979.

A Theorem Prover for Meta theory

Frank Brown
Dept. of Computer Science
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

We describe an automatic theorem prover for meta theory which is capable of proving the completeness of quantificational logic from intuitively true assumptions.

I. INTRODUCTION

This is a report of some of our research carried out mainly during the spring of 1978. It describes an implementation of a deductive system for meta theory which is capable of proving the completeness of quantificational logic. This deductive system is similar to earlier theorem provers developed by the author [1,2,3]. This meta-theory is based on a modal logic and a theory of meaning which have recently been developed by the author [4,5,6]. The meta theory and its deductive system are described in section 2. Finally, in section 3 we discuss an example theorem which the system has proven, namely the completeness of quantificational logic.

2. DESCRIPTION OF THE THEOREM PROVER

Our theorem prover consists of an interpreter for mathematical expressions and many items of mathematical knowledge. This interpreter is a fairly complex mechanism, but it may be viewed as applying items of mathematical knowledge of the form: $\phi \leftrightarrow \psi$ or $\phi = \psi$ to the theorem being proven, in the following manner. The interpreter evaluates the theorem recursively in a call-by-need manner. That is, if $(fa_1 \ldots a_n)$ is a sub-expression being evaluated, then the interpreter tries to apply its items of knowledge to that sub-expression before evaluating the arguments $a_1 \ldots a_n$. For each sub-expression that the interpreter evaluates, in turn it tries to match the ϕ expression of an item to that sub-expression. If, however, during the application process an argument a_i does not match the corresponding argument of the ϕ expression, then a_i is evaluated, and the system then tries to match the result of that evaluation. If ever the interpreter finds a sub-expression $\phi\theta$ which is an instance of ϕ of some item, then it replaces that expression by the corresponding instance $\psi\theta$ of ψ. At this point all memory of the sub-expression $\phi\theta$ is immediately lost and the interpreter now evaluates $\psi\theta$. If no items can be applied to a sub-expression then the sub-expression is not evaluated again but is simply returned.

Sometimes it will be the case that our interpreter will need to use items which are valid only in certain domains Π. In such a case we could represent the item as a conditional item of the form:

$$\Pi x \rightarrow (\phi x \leftrightarrow \psi x)$$
or
$$\Pi x \rightarrow (\phi x = \psi x)$$

The interpreter handles conditional items in the same way in which it handles non-conditional items until it has found a $\phi\theta$ which matches the sub-expression being evaluated. At this point on a conditional item, the interpreter tries to match each element in the conjunction Πx with some expression which it believes to be true. If such matches are found with substitution $\theta\sigma$ then $\psi\theta\sigma$ is returned. Otherwise the interpreter tries to apply another item as previously described.

However, in this theorem prover we will not bother to represent such items as conditional items, but simply use items of the form $\phi x \leftrightarrow \psi x$ and $\phi x = \psi x$. We will trust the theorem prover itself not to confuse the various domains and in particular not to instantiate any variable x to some term representing an entity in the wrong domain.

The rationale behind this trust is that if the domains are reasonably disjoint then since the theorem prover can only instantiate variables by a process of matching expressions, each variable can only be instantiated to something of the right domain.

There is certainly some sort of moral here and we think it is this: Don't worry about the details, just get on with the proof. We are not here arguing that this is a deep theoretical issue (although perhaps it is) but this approach to domain dependent items is advantageous in that it allows the theorem prover to obtain the proofs more quickly, and more importantly, the proofs obtained are shorter.

It may be argued that proofs obtained by such a theorem prover are worthless, since it would be very difficult to prove that a theorem prover using this method is sound, but as we have argued in [1], the real issue is not whether the theorem prover is sound but merely whether any particular proof is correct or not.

2.1 Logical Knowledge

Our theorem prover has knowledge about twelve logical symbols which are listed below with their English translations:

$\wedge$	and
$\vee$	or
$\sim$	not
■	true
□	false
$\rightarrow$	implies
$\leftrightarrow$	iff
$\exists$	there exists
$\forall$	for all
$=$	equal
$\rightarrow$	implies (This symbol is called a sequent arrow)

and   and (This symbol is used to form an implicit conjunction of sequents)

The sequent arrow may be defined as follows:

$p_1, \ldots, p_n \rightarrow q_1, \ldots, q_m =_{df} (p_1 \wedge \ldots \wedge p_n) \rightarrow (q_1 \vee \ldots \vee q_m)$

where p_i and q_j are sentences. Thus a sequent may be thought of as being a database of statements $p_1, \ldots, p_n$ called assertions which occur before the sequent arrow, and statements $q_1, \ldots, q_m$ called goals which occur after the sequent arrow. The implicit conjunction of different sequents may be thought of as being a group of different databases.

The items of logical knowledge, which are all schemata because they involve ellipses (i.e. dots representing arbitrary expressions), are listed below:

2.1.1 Assertion schemata:

$\blacksquare \rightarrow$: $(\ldots \blacksquare \ldots \rightarrow \ldots) \leftrightarrow (\ldots \ldots \rightarrow \ldots)$

$\square \rightarrow$: $(\ldots \square \ldots \rightarrow \ldots) \leftrightarrow \blacksquare$

$\sim \rightarrow$: $(\ldots \sim p \ldots \rightarrow \ldots) \leftrightarrow (\ldots \ldots \rightarrow p \ldots)$

$\wedge \rightarrow$: $(\ldots p \wedge q \ldots \rightarrow \ldots) \leftrightarrow (\ldots p,q \ldots \rightarrow \ldots)$

$\vee \rightarrow$: $(\ldots p \vee q \ldots \rightarrow \ldots) \leftrightarrow (\ldots p \ldots \rightarrow \ldots)$ and $(\ldots q \ldots \rightarrow \ldots)$

$\supset \rightarrow$: $(\ldots p \rightarrow q \ldots \rightarrow \ldots) \leftrightarrow (\ldots \ldots \rightarrow p \ldots)$ and $(\ldots q \ldots \rightarrow \ldots)$

$\leftrightarrow \rightarrow$: $(\ldots p \leftrightarrow q \ldots \ldots) \leftrightarrow (\ldots p,q \ldots \rightarrow \ldots)$ and $(\ldots \ldots \rightarrow p,q \ldots)$

$\exists \rightarrow$: $(\ldots \exists x\ \phi x \ldots \rightarrow \ldots) \leftrightarrow (\ldots \phi(f{*}_1 \ldots {*}_n) \ldots \rightarrow \ldots)$
where f is a new skolem function and ${*}_1 \ldots {*}_n$ are all the unification variables which occur in ϕx.

$\forall \rightarrow$: $(\ldots \forall x\ \phi x \ldots \rightarrow \ldots) \leftrightarrow (\ldots \forall(x*)\phi x,\ \phi* \ldots \rightarrow \ldots)$
where * is a new unification variable.

$= \rightarrow$: $(\Pi a \ldots \left\{\begin{matrix} a=t \\ t=a \end{matrix}\right\} \ldots \Gamma a \rightarrow \phi a \ldots \psi a) \leftrightarrow (\Pi t \ldots \Gamma t \rightarrow \phi t \ldots \psi t)$
where a is of the form $(f{*}_1 \ldots {*}_n)$ and f is a skolem function not occurring in t. This is our version of the law of Leibniz.

2.1.2 Goal schemata:

$\rightarrow \blacksquare$: $(\ldots \rightarrow \ldots \blacksquare \ldots) \leftrightarrow \blacksquare$

$\rightarrow \square$: $(\ldots \rightarrow \ldots \square \ldots) \leftrightarrow (\ldots \rightarrow \ldots \ldots)$

$\rightarrow \sim$: $(\ldots \rightarrow \ldots \sim p \ldots) \leftrightarrow (\ldots p \rightarrow \ldots \ldots)$

$\rightarrow \wedge$: $(\ldots \rightarrow \ldots p \wedge q \ldots) \leftrightarrow (\ldots \rightarrow \ldots p \ldots)$ and $(\ldots \rightarrow \ldots q \ldots)$

$\rightarrow \vee$: $(\ldots \rightarrow \ldots p \vee q \ldots) \leftrightarrow (\ldots \rightarrow \ldots p,q \ldots)$

$\rightarrow \supset$: $(\ldots \rightarrow \ldots p \rightarrow q \ldots) \leftrightarrow (\ldots p \rightarrow \ldots q \ldots)$

$\rightarrow \leftrightarrow$: $(\ldots \rightarrow \ldots p \leftrightarrow q \ldots) \leftrightarrow (\ldots p \rightarrow \ldots q \ldots)$ and $(\ldots q \rightarrow \ldots p \ldots)$

$\rightarrow \forall$: $(\ldots \rightarrow \ldots \forall x\ \phi x \ldots) \leftrightarrow (\ldots \rightarrow \ldots \phi(f{*}_1 \ldots {*}_n) \ldots)$
where f is a new skolem function and ${*}_1 \ldots {*}_n$ are all the unification variables which occur in ϕx.

$\rightarrow \exists$: $(\ldots \rightarrow \ldots \exists x\ \phi x \ldots) \leftrightarrow (\ldots \rightarrow \ldots \exists (x*)\phi x,\ \phi* \ldots)$
where * is a new unification variable.

2.1.3 Replica Creation Schemata

$\forall() \rightarrow$: $(\ldots \forall(x\ldots)\phi x \ldots \rightarrow \ldots) \leftrightarrow (\ldots \forall(x\ldots*)\phi x, \phi* \ldots \rightarrow \ldots)$
where * is a new unification variable and no more than one unification variable occurs in $(x\ldots)$.

$\rightarrow \exists()$: $(\ldots \rightarrow \ldots \exists(x\ldots)\phi x \ldots) \leftrightarrow (\ldots \rightarrow \ldots \exists(x\ldots*)\phi x, \phi* \ldots)$
where * is a new unification variable and no more than one unification variable occurs in $(x\ldots)$.

The items $\forall() \rightarrow$ and $\rightarrow \exists()$ are used to create additional replicas: $\phi*$, a universally quantified assertion $\forall(x\ldots)\phi x$, and an existentially quantified goal $\exists(x\ldots)\phi x$. The replica $\phi*$ is exactly like the original formula except that the initial quantifier is deleted and the bound variable associated with that quantifier is replaced by a new free unification variable.

A Unification Variable is a free variable which is created by $\forall \rightarrow$, $\rightarrow \exists$, $\forall() \rightarrow$, or $\rightarrow \exists()$ items, and which may later be instantiated to some term by the unification item (see section 2.1.4). Unification variables are written as a star sign: * possibly with numeric subscripts, such as: ${*}_1$, ${*}_2$, ${*}_3$.

In these four items we have seen formulae of the form $\forall(x\ldots)\phi x$ and $\exists(x\ldots)\phi x$ which are not usually thought of as being well formed sentences of logic. Such formulae should be interpreted as respectively $\forall x \phi x$ and $\exists x \phi x$ which are well formed sentences of logic. The ... list, which is called the replica instance list, is used merely to store certain pragmatic information used by the deductive system. This information is basically the list of unification variables (or more precisely the list of instantiations of the unification variables, see section 2.1.4) that were produced from this quantifier by applications of the $\forall \rightarrow$, $\rightarrow \exists$, $\forall() \rightarrow$, and $\rightarrow \exists()$ items.

2.1.4 The Unification schema:

Unify: $[(\ldots p_1 \ldots \rightarrow \ldots q_1 \ldots)$ and .. and .. $(\ldots p_n \ldots \rightarrow \ldots q_n \ldots)] \leftrightarrow$
$[(\ldots p_i \ldots \rightarrow \ldots q_i \ldots)$ and .. and .. $(\ldots p_n \ldots \rightarrow \ldots q_n \ldots)]\ \Theta$

where $1 \leq i \leq n$ and Θ is any one of the sets of substitutions of terms for unification variables which satisfy both the forcing restriction and the instantiation restriction. These two restrictions are described below.

The forcing restriction is the requirement that the substitution makes tautologous the greatest number of sequents starting with the first sequent and progressing towards the nth sequent.

In the case that there actually is some substitution which will make all the sequents tautologous, without further unification variables being created by the $\forall \rightarrow$ and $\rightarrow \exists$ items, then Θ will be one such substitution. As a minor point, if Θ makes all the sequents tautologous, then the unification schema is defined to return $\blacksquare$.

The instantiation restriction is the requirement that no unification variable be instantiated to a term which already occurs in the replica instance list of the quantifier of the given sequent which contains the unification variable. The rationale behind this restriction is that if a term t occurs in the replica instance list of a quantifier such as ∀ in (.. ∀xϕx .. → ..) then the sub-formulae of ϕt must already occur in some sequent which must be proven in order to prove the theorem.

2.1.5 Other-logical schemata

atom: (.. p .. → .. p ..) ↔■
and: (.. and ■ and ..) ↔(.. and ..)

The logical items are not all used at the same time. In particular the ∀()→, →∃(), and unify items are used in a special way. Initially, the interpreter evaluates each sequent trying to apply the items in the following order:

(1) Non splitting assertion items:
■→, □→, ∼→, ∧→, ∃→, = →
(2) Non splitting goal items:
→■ , →□, →∼, →∨, → ⊃, →∀
(3) Non logical items
(4) The atom and "and" items
(5) Splitting goal items: →∧, → ↔
(6) Splitting assertion items:
∨→, ⊃→, ↔ →
(7) →∃
(8) ∀→

After the above items have been applied as many times as possible, the interpreter then tries to apply the unify item to the resulting conjunction of sequents.

If the application of the unification item results in ■ then the processes terminates because the theorem has been proven. But, if the application of the unification item does not result in ■ , then the interpreter applies the ∀() → and →∃() items to certain formulas, and then repeats the whole process starting at step (1).

However, because the proof described in this paper is obtained without using the ∀() → and →∃() items we will not bother to describe the exact way they are used. The way they are used, along with more details about the Unify item, has however, been described in previous publications [2, 3].

One major difference between this theorem prover and our previous sequent logic theorem provers [2, 3] is that the →∃ and ∀→ have been inserted into the initial evaluation procedure as steps 7 and 8, and thus one instance of every quantifier is initially created before the Unify rule is ever applied. The reason for this is due to the vast numbers of trivial quantifiers produced by the modal sequent logic described in section 2.2. It was found that unless instances of these quantifiers were produced before unification takes place, many important bindings would not be found quickly enough and irrelevant bindings would be produced by the forcing restriction, due to the fact that the relevant formulae would be available for matching.

2.2 Theory of Modality

Our theory of modality is based on a very strong modal logic which is described in [4, 5]. It consists of a single primitive unary symbol: ⊢ which is interpreted as logical truth. This modal logic is stronger than S5 and can be described by the following minimal set of inference rules and axioms:

RO: from p infer ⊢ p
A1: ⊢p → p
A2: ⊢(p → q) → (⊢p → ⊢q)
A3: ⊢p ∨⊢∼ ⊢ p
A4: (∀q World*q → ⊢* q p) → ⊢p

The inference rule RO and the axioms A1, A2 and A3 are essentially the S5 modal logic. The last axiom A4 expresses Leibniz's intuition that something is logically true only if it is true in all worlds. The World* and ⊢ * symbols have the same meaning as respectively the World and the binary ⊢ symbols. World and binary ⊢ are both defined in section 2.2.2.

Our automatic theorem prover does not use the above axioms but is based on the sequent calculus derived from these axioms which is described in [5]. We describe this modal sequent calculus in section 2.2.1 and then list some definitions of modal concepts used by the theorem prover in section 2.2.2. Finally, in section 2.2.3 we discuss the possibility problem of modal logic.

2.2.1 Rules of the Sequent Calculus

We list below thirteen theorems of our modal logic of the form p → q or r → (p ↔ q) which may be used as rewrite rules replacing p by q in any context in which r is a hypothesis. The symbols World* and ⊢ * have the same meaning respectively as World and ⊢ , but are never to be replaced by their definitions in a proof procedure using these rules. Furthermore any initial theorem given to such a proof procedure must not itself contain the World* or ⊢ * symbols, although it could of course contain the World and ⊢ symbols.

⊢ :	(⊢ p) ↔∀w(World* w) →⊢*w p
⊢∧:	(World* w) → ((⊢*w(p ∧ q)) ↔ (⊢*w p∧⊢*w q))
⊢∨:	(World* w) → ((⊢*w(p ∨ q)) ↔ (⊢*w p∨⊢*w q))
⊢→ :	(World* w) → ((⊢*w(p → q)) ↔ (⊢*w p →⊢*w q))
⊢↔ :	(World* w) → ((⊢*w(p ↔ q)) ↔ (⊢*w p ↔⊢*w q))
⊢∼ :	(World* w) → ((⊢*w(∼ p)) ↔ ∼ ⊢* w p)
⊢■ :	(World* w) → ((⊢*w ■) ↔ ■)
⊢□ :	(World* w) → ((⊢*w □) ↔ □)
⊢∀ :	(World* w) → ((⊢*w(∀x ϕx)) ↔ (∀x ⊢*w ϕx))
⊢∃ :	(World* w) → ((⊢*w(∃x ϕx)) ↔ (∃x ⊢*w ϕw))
⊢a :	(World* w) → ((⊢*w(∀p ϕp)) ↔ (∀p ⊢*w ϕp))
⊢e :	(World* w) → ((⊢*w(∃p ϕp)) ↔ (∃p ⊢*w ϕp))
⊢⊢ :	(World* w) → ((⊢*w(⊢ p)) ↔ ⊢p)

The ⊢∀ and ⊢∃ theorems pertain to quantifiers of object language variables whereas the ⊢a, ⊢e theorems pertain to quantifiers for propositional variables. The ⊢∀ and ⊢∃ theorems are equivalent to the fact that something is an object iff it is logically true that it is an object. All these theorems hold regardless of whether propositions are objects or not.

In order to try to prove a theorem ψ with a proof procedure using these rules, sometimes it must actually try to prove $\vdash\psi$ instead. There is a deep and beautiful reason for this which is basically that this initial $\vdash$ inserted before ψ is a symbol of the metalanguage of this logic as are all the $\vdash$* and World* symbols. Essentially $\vdash\psi$ is the statement in the methatheory that ψ is logically true, and it is this rather than ψ itself which we are trying to prove.

Our theorem prover also contains the following special laws dealing with equality.

$= \vdash \leftrightarrow$: $\quad \vdash(p \leftrightarrow q) \leftrightarrow p = q$
$= \vdash \sim$: $\quad \vdash \sim p \quad \leftrightarrow p = \square$
$= \vdash$: $\quad \vdash p \quad \leftrightarrow p = \blacksquare$

These laws are used only on assertions in a sequent, and only in the case that either p or q is a skolem function.

These three laws are not derivable from the minimal axiomatization described earlier, but are derivable if the following axiom is added:

A5: $\vdash(p \leftrightarrow q) \rightarrow p = q$

2.2.2 <u>Modal Concepts</u>

$D\vdash$: $\vdash p\ q \leftrightarrow$
$\vdash(p \rightarrow q)$ "p entails q"
$D\equiv$: $p \equiv q \leftrightarrow$
$\vdash(p \leftrightarrow q)$ "p is synonomous to q"
$D\Diamond$: $\Diamond p \leftrightarrow$
$\sim\vdash\sim p$ "p is possible"
Ddet : (Det p q) $\leftrightarrow$
$\vdash p\ q \vee \vdash p \sim q$ "p determines q"
Dcom : (Complete p) $\leftrightarrow$
$\forall q$(Det p q) "p is complete"
Dwor : (World p) $\leftrightarrow$
$\Diamond p \wedge$ (Complete p) "p is a world"
Dval : (Valid p) $\leftrightarrow$
$\forall q$(World q) $\rightarrow \vdash q\ p$ "p is valid"
Dsat : (Sat p) $\leftrightarrow$
$\exists q$(World q) $\wedge \vdash q\ p$ "p is satisfiable"
Duniq : (Uniq <u>p</u> ϕp) $\leftrightarrow$
$\forall p \forall q\ \phi p \wedge \phi q \rightarrow p \equiv q$ "p is unique in ϕ"
Done : (One <u>p</u> ϕp) $\leftrightarrow$
$\exists q\ \forall r(\phi r \leftrightarrow q \equiv r)$ "p is one in ϕ"
Dcat : (Cat p) $\leftrightarrow$
(One q(World q $\wedge \vdash$ q p)) "p is categorical"
Dmaxp : (Maxpos p) $\leftrightarrow$
(One q($\Diamond$q $\wedge \vdash$ q p))"p is maximally possible"
Dmax : (Max p) $\leftrightarrow$
(One q($\vdash$ q p)) "p is maximal"

2.2.3 <u>The Possibility Problem</u>

The possibility problem of modal logic is that from the logical axioms of modal logic we cannot prove certain elementary facts about the possibility of conjunctions of distinct possibly negated atomic expressions consisting of nonlogical symbols. For example, if we have a theory formulated in our modal logic which contains the nonlogical atomic expression (ON A B) then since $\sim$(ON A B) is not logically true, it follows that (ON A B) must be possible. Yet $\Diamond$(ON A B) is not a theorem of our modal logic.

Thus, for any theory expressed in modal logic, a certain number of axioms dealing with possibility should also be added.
For example, in the case of the propositional logic, or in the case of the quantificational logic over a finite domain since it reduces to propositional logic, one sufficient but inefficient axiomatization would be to assert the possibility of all consistent disjunctions of conjunctions of literals as additional nonlogical axioms:

$\Diamond(\bigvee(\bigwedge$Literals)

A computationally better axiomatization which is obtained by noting that the possibility of a disjunction of sentences is implied by the possibility of any one of those sentences:

$\Diamond x \rightarrow \Diamond(x \vee y)$

is to assert only the possibility of all consistent conjunctions of literals:

$\Diamond$ ($\bigwedge$literals)

Using the meaning function M defined in section 2.4 this may be done in a finite manner by adding the single axiom:

Prop Pos Ax: (Conj S) $\wedge$ (Consist S) $\rightarrow \Diamond$(M S)

where Conj and Consist are recursive functions defined as follows:

(Conj S)= df (Lit S) $\vee \exists T \exists R$ (S=[T '$\wedge$ R] $\wedge$
(Lit T) $\wedge$ (Conj R))
(Lit S) = df ($\exists T$ S=['$\sim$T] $\wedge$ (Atomicsent T)) $\vee$
(Atomicsent S)
(Consist [ ]) = df $\blacksquare$
(Consist [S.L]) = df (Consist2 S L) $\wedge$
(Consist L)
(Consist2 S [ ])= df $\blacksquare$
(Consist2 S[T.L]) = df $\sim$(Opp S T) $\wedge$
(Consist2 S L)
(Opp S T) = df S=['$\sim$T] $\vee$ T=['$\sim$S]

The methods for representing object language expressions in our logic and for obtaining their meanings are defined in sections 2.3 and 2.4.

It is important to note that it is unlikely that this finite recursive axiomatization of the possibility problem for propositional logic, could be extended to even first order quantificational logic. The reason is that if it could then it would provide a positive solution to Hilbert's Einscheingunts problem. However, if we are willing to forgo the requirement of recursiveness then by letting S in PropPosAx be an infinite conjunction of variable free literals then it is possible to produce an axiomatization for a first order quantificational logic object language. See section 2.7.

2.3 <u>Theory of Syntax</u>

Our theory of syntax contains two basic primitive symbols, a binary symbol: Cons and a zeroary symbol: Nil. It also contains a number of other zeroary symbols
'$\wedge$, '$\vee$, '$\rightarrow$, '$\leftrightarrow$, '$\sim$, '$\blacksquare$, '$\square$, '$\forall$, '$\exists$, 'x, 'p
each of which may be thought of as being a name of the symbol obtained from it by deleting the accent sign. The accented symbols may, however, be thought of as being defined in terms of Cons and Nil and thus need not be viewed as additional primitive symbols.

Our theory of syntax currently consists of only one definition scheme:

Dlist: $[x_1 \ldots x_n]$ = df (Cons $x_1 \ldots$
(Cons x_n Nil)...)

Our theory of syntax is described in more detail in [6].

2.4 The Theory of Meaning

The syntax of our theory of meaning currently consists of one binary primitive symbol: m, such that (m S A) is interpreted as the meaning of an expression S in the association list A. It also consists of one unary defined symbol: M, which stands for the meaning of an expression in a null association list.

We assume that the definition and recursive axioms for meaning are all logically true.

Our theory of meaning is based on the recursive meaning function m described in [6]. This function consists of the following recursive axioms:

M :	(M S) = df (m S Nil)
m '∧ :	(m[S '∧ T]A) ↔ (m S A) ∧ (m T A)
m '∨ :	(m[S '∨ T]A) ↔ (m S A) ∨ (m T A)
m ' → :	(m[S '→ T]A) ↔ (m S A) → (m T A)
m ' ↔:	(m[S '↔ T]A) ↔ (m S A) ↔ (m T A)
m ' ∿ :	(m['∿ S]A) ↔ ∿(m S A)
m ' ■ :	(m '■ A) ↔ ■
m ' □ :	(m '□ A) ↔ □
m '∀ :	(m '∀ V S]A) ↔∀X(m S[[V.X].A])
m '∃ :	(m['∃ V S]A) ↔∃X(m S[[V.X].A])
m ' a :	(m['∀ U S]A) ↔∀p(m S[[U.p].A])
m ' e :	(m['∃ U S]A) ↔∃p(m S[[U.p].A])
m V :	(m V A) ↔ (Val V A)
m U :	(m U A) ↔ (Val U A)

m' :(m['ϕS_1...S_n]A) ↔ (ϕ(mS_1A)...(mS_nA))

for each non-logical symbol ϕ of the object language.

In these rules the notation [α_1...α_n] is an abbreviation for (Cons α_1...(Cons α_n Nil)) and the notation [α.β] is an abbreviation for (Cons α β). Thus, for example the m '∿ law actually is:

(m(Cons '∿(Cons S Nil))A) ↔ ∿(m S A)

S and T range over expressions, usuallv sentences. A ranges over association lists. V ranges over object variables and U ranges over propositional variables.

The Val function is defined as follows:

V1: (Val X[[X.Z].A])=Z
V2: (∿X=Y) → (Val X[[Y.Z].A])=(Val X A)

2.5 Model Theory

Our Model Theory consists of the following definitions:

D$Com ($Complete p q̲ ϕq) ↔ (∀q ϕq → (Det p q))
"p is complete for ϕ"
D$Wor($World p q̲ ϕq) ↔◇p∧($Complete p q̲ϕq)
"p is a world for ϕ"
D$Val($Valid p q̲ ϕq)↔∀q ($World q r̲ ϕ r) → ⊢q p
"p is valid for ϕ"
D$Sat($Sat p q̲ ϕq) ↔ ∃q ($World q r̲ ϕr)∧⊢ q p
"p is satisfiable for ϕ"
D$Cat(Cat p q̲ ϕq) ↔(One q̲ ($World q r̲ ϕr)∧⊢q p
"p is categorical for ϕ"
D$M (Model p) ↔ ($World p q̲ (∃S (M S) ≡ q))
"p is a model"
D$MSat(Modelsat p) ↔∃q (Model q)∧⊢q p
"p is satisfied by some model"
D$MVal (Modelval p)↔∀q (Model q) → ⊢q p
"p is true in all models"
D$MCat (Modelcat p)↔ (One q̲ Model q∧ ⊢ q p)
" p is true in exactly one model"

Note that a model is essentially nothing more than a $World with respect to the subdomain of concepts which are expressible in the object language.

2.6 Proof Theory

Our proof consists of a single unary primitive symbol: ⊩ such that ⊩S is interpreted to mean that the sentence S is a theorem, or rather that S is provable.

Our proof theory also contains several defined symbols which are listed below with their definitions and interpretations.

D⊩	:⊩T S ↔⊩[T → S]
	"S is a theorem of T"
D◇	:◇ S ↔ ∿⊩['∿ S]
	"S is consistent"
D$◇_o$	:$◇_o$ S ↔ ∿⊩['∿ S]
	"S is consistent"
Ddec	:(Dec T S) ↔ ⊩T S ∨ ⊩ T['∿ S]
	"T decides S"
DThCom	:(ThComplete T)↔ ∀S(Dec T S)
	"T is a complete theory"
DThWor	:(ThWorld T) ↔ $◇_o$T ∧ (ThComplete T)
	"T is a world theory"
DThWor$_o$	:(ThWorld$_o$ T)↔ $◇_o$T ∧ (ThComplete T)
	"T is a world theory"
DthWExt	:(ThWorldext S)↔ ∃T (ThWorld T)∧ ⊩ T S
	"T has a world theory extension"

The purpose of the $◇_o$ and ThWorld$_o$ definitions is pragmatic and will be explained in section 2.7.

Furthermore, the variables S and T in the last definition (DThWExt) are not of the same sort. The sort distinctions have been omitted in accordance with this theorem prover's basic rational for handling sorts as discussed in section 2. In this last definition the variable S ranges over finite sentences whereas the variable T ranges over both finite and infinite sentences. Infinite conjunctions of finite sentences, however, are probably sufficient.

2.7 Miscellaneous Lemmas

The following lemmas are assumed as extra rewrite rules in various proofs. In particular they are necessary for a proof of the completeness theorem.

Sound: ⊩S → ⊢(M S)
LIND: ◇S →(ThWorldext S)

It should not be thought that the axiom LIND is either logically false or even that it is false in a meta theory which is strong enough to derive Godel's incompleteness theorem because LIND claims only that there is a complete consistent extention, and not that this extention is a finite sentence.

PosAx: (ThWorld S) →◇(M S) ∧ (ThWorld$_o$ S)

The PosAx theorem is equivalent to:
(ThWorld S)→ ◇(MS)

The intuitive justification for this axiom lies in the fact that a Thworld may be thought of as being an infinite conjunction of variable free literals. Viewed in this manner we see that PosAx is nothing more than a solution to the possibility problem for those concepts which are expressible in the object language.

The above three axioms which are only implications are used as rewrite rules only on assertions in a sequent. Thus, for example, the sequent:

$\nVdash S \rightarrow \phi$

would become:

$\vdash (M\ S) \rightarrow \phi$

by application of the soundness lemma: Sound, to the assertion $\nVdash S$.
The Soundness lemma would not be applied to the sequent

$\phi \rightarrow \nVdash S$

because $\nVdash S$ is a goal.

It is easy to see that if unrestricted Lindenbaum Lemma: LIND could be applied an infinite number of times, since $\Diamond S$ is rewritten as (ThWorldext S) which in turn may, by using various definitions, be rewritten as:
$(\exists U\ \ \Diamond U \wedge (\text{ThComplete } U) \wedge \nVdash U\ S)$, thus producing a new formulae $\Diamond U$ to which LIND can again be applied. For this reason we restrict LIND to being applied only to $\Diamond S$ formulas not generated by previous applications of LIND. This restriction is implemented by simply defining ThWorldext S to produce $(\exists U\ \ \Diamond_o U \wedge (\text{ThComplete } U) \wedge \nVdash U\ S)$
where $\Diamond_o$ is a distinct symbol from $\Diamond$ even though it has exactly the same meaning.

It is also easy to see that PosAx would have the same sort of problem, and for this reason it uses the symbol ThWorld_o.

One final point worth noting about these lemmas is that even though they are all implications and not equivalences, they would be true even if they were equivalences. The reason we do not make Sound and PosAx equivalences is that we don't need to know these facts in order to prove, for example, the completeness theorem, and furthermore for pragmatic reasons we would never want to apply them to goal formulas anyway. In the case of the soundness theorem, the reason we do not assume an equivalence is that the equivalence is itself a version of the completeness theorem, and it obviously makes no sense to assume the completeness theorem while trying to prove it.

3. THE COMPLETENESS THEOREM

The theorem prover for metatheory is capable of proving the completeness of quantificational logic from intuitively true assumptions. Specifically the theorems described in section 2 are assumed as axioms. The completeness theorem which states that the meaning of every consistent theory is possible is written as follows:

$\Diamond S \ \ \rightarrow \Diamond (MS)$

This theorem was proven by our theorem prover on a DEC KI10 computer in 38 seconds. The number of sequents produced was 326 octal. See [7].

This proof, by the way, is probably the first proof of a completeness theorem relating provability to the modal concept of logical truth. The notion of truth unlike set theoretic characterizations [8,9] can be shown to satisfy Tarski's famous criteria for truth in a world [10] namely that (in that world) something is a true sentence in a world iff it is the case.

$\vdash w\ (\vdash w (MS) \leftrightarrow (MS))$

(in w) (S is a true sentence iff (MS)).
One final advantage of this completeness proof is that it is a proof purely from the laws of logic (plus a few recursive functions which are probably definable in second order logic) alone, and assumes no axioms of set theory such as the axiom of choice. So in fact it has shown that not only does completeness follow from certain strong set theoretic principles, but that it is logically true which is a much more general result.

Acknowledgements

It has taken me over one half of a decade to develop the meta theory on which the automatic theorem prover is based. I thank Professor B. Meltzer for supporting this research over this long period, and for commenting on various drafts of this theory, the first of which was written five and one half years ago. I also thank the many other people who have interacted on various aspects of this theory, including Dr. R. Kowalski, Dr. K. Clark, Dr. B. Richards, Dr. W. Bibel, Dr. S. Tarnlund, and especially Dr. C. Schwind.

References

1. Brown, F.M. "An investigation into the goals of research in Automatic Theorem Proving as related to Mathematical Reasoning", DAI Research Report 49, 1978.
2. Brown, F.M. "A Theorem Prover for Elementary Set Theory", IJCAI5 Conf. Proc., MIT, 1977.
3. Brown, F.M. "Towards the Automation of Set Theory and its Logic", DAI Research Report 34, 1977.
4. Brown, F.M. "A Theory of Meaning", DAI Working Paper, 17, 1976.
5. Brown,.F.M. "A Sequent Calculus for Modal Quantificational Logic", 3rd AISB/GI Conf. Proc., Hamburgh, 1978.
6. Brown, F.M. "The Theory of Meaning", DAI Research Report 35, 1977.
7. Brown, F.M. "An Automatic Proof of the Completeness of Quantificational Logic" DAI Research Report 52, 1978.
8. Henkin, L. "The Completeness of the First Order Functional Calculus", JSL, Vol. 14, 1949.
9. Mendelson, E. Introduction to Mathematical Logic, van Nostrand Reinhold Company, New York, 1964.
10. Tarski, A. "The Concept of Truth in Formalized Languages" (1931) Logic, Semantics, Mata mathematics, trans by J.H. Woodger, Oxford, Clarendon Press, 1956.

First-Order Unification in an Equational Theory

by

Michael J. Fay

November 11, 1978

Some kind of matching rule lies at the heart of any automatic deduction procedure. The most prominent matching rule is the well-known unification algorithm of Robinson [Ro 65]. This algorithm decides whether two terms have a common instance, and if so, provides a substitution for the variables that produces a most general common instance. The application of Robinson's algorithm is limited to first-order logic with universally quantified variables.

Several researchers have argued that more powerful matching rules could lead to more efficient automatic theorem provers. Plotkin [Pl 72] presented an informal argument indicating that his associative unification algorithm could avoid redundant computations carried out by a paramodulation-based theorem prover that used only standard first-order unification. Slagle [Sl 74] also proposed a theorem-proving system in which certain common properties, such as associativity, commutativity and various cancellation and identity rules were built in (although not directly as part of the unification algorithm). Procedures have been designed for unification of terms containing functions that are associative [Pl 72, Ne 74, Sl 74, Si 75], commutative and associative [Ne 74, St 75, Li&Si 76], commutative [Ne 74, Sl 74, Si 76], idempotent [RS 78], and idempotent, associative, and commutative [Li&Si 77]. To be complete, these procedures generally must find more than one most general unifying substitution, unlike ordinary first-order unification, in which all unifiers will be an instance of a single most general unifier. The general problem of unification in the presence of a theory T is called "T-unification".

Notice that all of the above procedures are special-purpose, tailored to perform unification in the presence of particular properties. In this report, I present a procedure that is general enough to unify terms in the presence of any theory that can be expressed using only a set of equations, such that the equations form a "complete set of reductions" as defined by Knuth and Bendix [Kn&Be 70]. Such theories include elementary group theory, associativity, and idempotency. In recent generalizations of the Knuth-Bendix methods [La&Ba 77abc, St&Pe 77, Hu 77, La 78a], "permutative" equality axioms such as commutativity may be included. These methods allow further generalization of the T-unification procedure. An important theory for which the Knuth-Bendix method fails is the theory of an associative, idempotent function. Consequently, the method of this paper does not attack the (open) problem of unification in such a theory [RSSU 78].

The purposes of this research are 1) to generalize the methods used in the special-purpose unification procedures, 2) to examine the relationships between the application of rewriting rules and the substitution of terms for variables, and 3) to develop procedures that are more efficient than those appearing in the literature. This latter goal has not been achieved, but some progress has been made on the first two fronts.

We will use the Greek letters θ, Φ, and Υ to denote substitutions; p, q, r, t, and u to denote terms; and x, y, and z to denote variables. Also, we use the notation $t \sim_T u$ to mean that $t = u$ can be inferred from the theory T. Often we will be able to omit the T without loss of clarity.

Suppose that $t\theta \sim u\theta$. Then we say that θ is a <u>T-unifier</u> of t and u. If $t \sim u\theta$, then t is said to be a <u>T-instance of a term</u> u. If θ and Φ are substitutions, $\theta \sim \Phi$ means that for every variable x, $x\theta \sim x\Phi$. A substitution Φ is a <u>T-instance of a substitution</u> θ (alternately, θ is a <u>T-generalization</u> of Φ) if and only if $\Phi \sim \theta\Upsilon$ for some substitution Υ.

In general, two terms will have more than one "most general T-unifier". Let t and u be terms, and S be a (possibly infinite) set of substitutions. Then, with respect to t and u and a theory T, S is

1. correct if and only if $t\theta \sim u\theta$ for every θ in S
2. complete if and only if for every Φ such that $t\Phi \sim u\Phi$, there is a θ in S and Ψ such that $\Phi \sim \theta\Psi$
3. independent if and only if for every pair of distinct substitutions θ and Φ in S, there is no Ψ such that $\theta\Psi \sim \Phi$.

If S is correct, complete, and independent, we say that S is a most general set of T-unifiers of t and u in T. Given a theory T, a procedure P that generates a set S of substitutions for any pair of terms t and u is said to be (correct, complete, independent) if and only if for every t and u, S is (correct, complete, independent) with respect to t, u and T. These definitions are a simplification of those in Plotkin [Pl 72].

To illustrate the T-unification process, suppose T is the theory of an associative funciton f. T can be expressed by the equation $f(f(x,y),z) = f(x,f(y,z))$, where x, y, and z are universally quantified varibles. Now, suppose we wish to find all T-unifiers of the terms $t = f(a,y)$ and $u = f(x,b)$. The obvious T-unifier is the (ordinary) unifier $\theta = \{a/x,\ b/y\}$. Since $t\theta$ and $u\theta$ are identical, they are equal in the theory T. But there is another T-unifier Φ, not a T-instance of θ, such that $t\Phi \sim_T u\Phi$; it is $\Phi = \{f(a,z)/x,\ f(z,b)/y\}$. Observe that

$$t\Phi = f(a,f(z,b)) \sim_T f(f(a,z),b) = u\Phi .$$

Φ is discovered by observing that the associative rule must apply to at least one of the subterms of $t\Phi$ or $u\Phi$ (including the terms themselves).

Figure 1 shows how the associativity rule can be used to discover the T-unifier Φ. As we shall see, only the left side of the associativity equation need be used in this process. Also, the terms t and u can be put into right-associative form (as shown) before the computation starts.

The T-unifier computed in Figure 1 is

Φ_1 composed with Φ_2, which is $\{f(a,y')/x,$

$b/z',\ f(y',b)/y\}$. Since z' does not appear in either t or u, the element b/z' may be dropped.

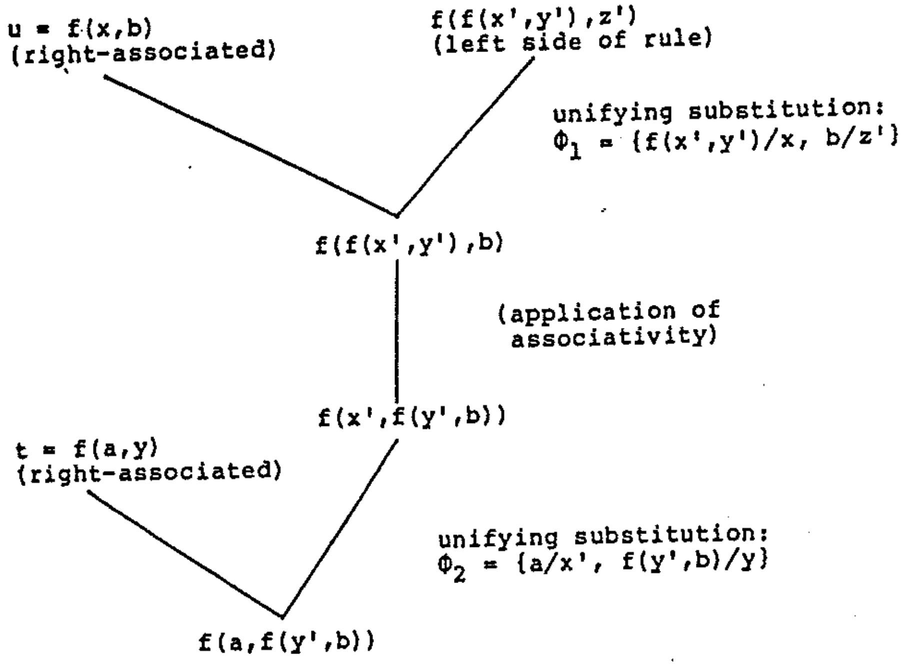


Figure 1
Discovering a T-unifier for associative terms

In the above example, associativity is effectively used as a one-way rewriting rule of the form p --> q. To apply the rule to a term t, instantiate p so that $p\theta$ equals a subterm of t (possibly t itself). Then replace $p\theta$ in t by $q\theta$. Symbolically, we write t(t'| p --> q), where t' is the subterm being rewritten.

The associative rule, when used alone, has special properties. It can be applied on subterms of t, in any order, producing a unique normalized result: the right-associated form of t. Let N(t) denote this result. Then, t ~ u if and only if N(t) = N(u).

Generalizing this idea, suppose that a theory T, consisting of a set of equations, can be expressed by a _complete set of reductions_ [Kn&Be 70]. That is, suppose there exists a set R of rewriting rules of the form p --> q, where p and q are terms, with the property that:

> R has finite termination, and no two distinct irreduceable terms are equal in the equational theory corresponding to R.

Then it is easy to decide, for any terms t and u, whether $t \sim_T u$. Simply apply reductions from R to subterms of t and u, in any order, until no more reductions apply. This process will terminate; we denote the resulting _normalized_ terms by N(t) and N(u). (For complete sets of reductions, N(t) and N(u) are unique, regardless of the order of application of reductions.) Then $t \sim_T u$ if and only if N(t) = N(u). A substitution θ is said to be normalized if for every variable x, $x\theta$ is normalized.

Associativity by itself is a complete set containing one reduction. Knuth and Bendix discovered several interesting complete sets of reductions, including a set of ten reductions for group theory.

Referring back to Figure 1, we see that, in general, T-unifiers of t and u can be formed by _unifying subterms_ of t and u _with left-hand sides of reductions_, and then testing the resulting terms for T-equality by normalizing them. This process is iterated, while at the same time attempts at ordinary unification are made.

Let the notation mgu(t",p) represent a most general unifier of t" and p, if one exists. Also, for any substitution θ and set of variables V, define "θ restricted to V" (written $\theta \restriction V$) to be $\{t/x \mid t/x \in \theta$ and $x \in V\}$. Finally, Var(t) is the set of distinct variables occurring in the term t.

Let L(t,R) be the set of all substitutions that allow any non-variable subterm of the normalized term t to be rewritten. Then

$$L(t,R) = \{\Phi \mid \Phi = \mathrm{mgu}(t'',p)\restriction \mathrm{Var}(t) \text{ where } p \dashrightarrow q \text{ is in } R, \text{ and } t'' \text{ is a non-variable subterm of } t\}$$

(Incidentally, t" cannot be a ground term, because if it were both ground and could be unified with p, then t would not be normalized.) We use this definition to formulate the TU2 procedure shown in Figure 2, which generates a complete (but generally not independent) set of T-unifiers ($\underline{U}$ means union, $\in$ denotes the empty substitution). The SELECT statement is nondeterministic, and is equivalent to HALT if the selection set is empty.

The requirement that $\theta = N(\theta\Phi)$ did not appear in the original version of TU2 [Fa 78] and is due to Lankford [La 78b]. This restriction, which eliminates one important source of exponential growth, results from the following observations:

1. If $\theta\Phi$ is normalized, then θ must also be normalized.

```
TU2 PROCEDURE(t, u: term) RETURNS(θ: substitution)
BEGIN
    θ := ∈;  t := N(t);  u := N(u)
    REPEAT
        IF t = u THEN RETURN(θ)
        SELECT Φ FROM L(t,R) U L(u,R) U {mgu(t,u)}
        IF θΦ ≠ N(θΦ) THEN HALT
        t := N(tΦ);  u := N(uΦ)    [illegible]
    ENDLOOP
END TU2
```

Figure 2

2. By Theorem 2 (below), $t\theta \sim u\theta$ implies $tN(\theta) \sim uN(\theta)$. Therefore, for any complete set of T-unifiers S of t and u, there is a normalized complete set $S' \equiv \{N(\theta) \mid \theta \in S\}$. Since every element of S is a T-instance of an element of S', we can require all final substitutions returned by TU2 to be normalized.

The proof that TU2 will in fact find sufficient substitutions to generate a complete normalized set is given in Lemmas 2 and 3, and Theorem 4.

It is interesting to note that the iterations within TU2 correspond to applications of certain inference rules in other automatic deduction systems (e.g. [La 75]). By retaining substitutions that get built during inferences, such systems can eliminate redundant searches by verifying that these substitutions are normalized, in the same manner that TU2 does.

In terms of efficiency, TU2 is superior to general paramodulation, because (1) it employs the above efficiency measure; (2) it uses equations in only one direction; and (3) if does not attempt to unify non-normalized terms. But it compares unfavorably with the special-purpose T-unification procedures of Raulefs and Siekmann [RS 78] and Plotkin [Pl 72]. For example, Plotkin's associative unification procedure is independent; TU2 is not. (Note: a decision algorithm for associative unification has since been constructed by the Russian mathematician Makanin [Ma 77].) Also, TU2 explores many infinite paths that do not produce any T-unifiers (more so than Plotkin's procedure). Much of the inefficiency of TU2 is due to the fact that it composes the same disjoint substitutions in different orders; this could perhaps be eliminated by imposing some order on how the substitutions Φ are chosen.

The following simple theorems and corollaries are used in demonstrating the completeness of TU2. Proofs can be found in [Fa 78]. N is a normalizing function with respect to a complete set of reductions. Thus, $N(t) = N(u)$ is equivalent to $t \sim u$.

Lemma 1: The effect of rewriting followed by substitution is the same as the effect of substitution followed by rewriting. Symbolically, if t' is a subterm of t, then $t(t' \mid p \rightarrow q)\theta = t\theta(t'\theta \mid p \rightarrow q)$. We assume that the variables in $p \rightarrow q$ are renamed and unique for each application.

Theorem 1: $N(t)\theta \sim t\theta$.
Corollary 1: $N(\theta)\Phi \sim \theta\Phi$.
Corollary 2: $t \sim u$ implies $t\theta \sim u\theta$.

In general, it is not true that $N(t\theta) = N(t)\theta$. As a counterexample, let $t = f(x,y)$, $\theta = \{e/y\}$, and R be the complete set of reductions $\{f(x,e) \rightarrow x\}$. Then $N(t\theta) = x$, but $N(t)\theta = f(x,e)$.

Theorem 2: $tN(\theta) \sim t\theta$.
Corollary 3 $\theta N(\Phi) \sim \theta\Phi$.

We now proceed with the proofs of the completeness and correctness of TU2.

Lemma 2: Let t and u be unequal normalized terms, and θ a normalized substitution, such that $t\theta \sim u\theta$. Then there is a normalized substitution Φ that is a generalization of θ, such that either: (1) $\Phi \in L(t,R) \underline{U} L(u,R)$, or (2) $\Phi = mgu(t,u)$.

Proof: First note that if such a Φ exists, then Φ is normalized, since $\Phi\Upsilon = \theta$ for some Υ, and θ is normalized.

We show that if condition (1) is false, then (2) holds. Suppose that there is no substitution Φ in $L(t,R) \underline{U} L(u,R)$ that is a generalization of θ. Then by the definition of L(t,R), no rewrite rule applies to any subterm $t"\theta$ of $t\theta$, except possibly to $x\theta$, where x is a variable. But since θ is normalized, this latter case is excluded; consequently, $t\theta$ is normalized. Similarly, $u\theta$ is normalized. Now $t\theta = N(t\theta) = N(u\theta) = u\theta$, so θ is an instance of mgu(t,u), by the unification proof of Robinson [Ro 65]. ⌘

The next lemma proves some inductive assertions associated with TU2.

Lemma 3: Suppose $t\theta' \sim u\theta'$, θ' is normalized, and TU2 is invoked with t and u. Let t_i, u_i, and θ_i be the values of the variables t, u, and θ in TU2 just before the (i+1)th time the condition "t=u" is tested. Then the following two properties are true for all i, $0 \le i \le n$, where the loop is executed at least n times:

(1) $t\theta_i \sim t_i$,

(2) $u\theta_i \sim u_i$, and

t_i, u_i, and θ_i are normalized. In addition, there is a particular path of execution such that for every i, $0 \le i \le n$, there is a normalized substitution Υ_i (a "remainder") such that

(3) $\theta_i\Upsilon_i = \theta'$, and

(4) $t_i\Upsilon_i \sim u_i\Upsilon_i$.

<u>Proof</u>: We prove this by induction on i. As the base case, set $i = 0$, noting that the condition "t=u" must be tested at least once. Observe that in TU2, just before the loop is entered, the values of the program variables are such that $t_0 = N(t)$, $u_0 = N(u)$, and $\theta_0 = \epsilon$. Clearly $t\epsilon \sim N(t)$ and $u\epsilon \sim N(u)$, so properties (1) and (2) are satisfied. Letting $\Psi_0 = \theta'$, we see that properties (3) and (4) are also satisfied: $\epsilon\theta' = \theta'$, and $t_0\theta' \sim u_0\theta'$ (the latter by Theorem 1).

Suppose properties (1) through (4) hold for all i, $0 \leq i \leq k$, and that TU2 does not terminate the loop on the (k+1)th test or the "t=u" condition. Since t_k, u_k, and Ψ_k are normalized, and (4) holds, we can apply Lemma 2: there must be a generalization Φ of Ψ_k such that either $\Phi \in L(t_k,R) \cup L(u_k, R)$, or $\Phi = mgu(t_k,u_k)$. The SELECT statement of TU2 must choose such a Φ. Let Ψ_{k+1} be a most general substitution such that $\Phi\Psi_{k+1} = \Psi_k$. (Note that such a Ψ_{k+1} must be normalized.) TU2 assigns the following values:

$$\theta_{k+1} := \theta_k\Phi$$

$$t_{k+1} := N(t_k\Phi)$$

$$u_{k+1} := N(u_k\Phi)$$

Thus,

$$(5) \quad \theta_{k+1}\Psi_{k+1} = \theta_k\Phi\Psi_{k+1} = \theta_k\Psi_k = \theta'$$

$$(6) \quad t_{k+1}\Psi_{k+1} = t_k\Phi\Psi_{k+1} = t_k\Psi_k$$

$$(7) \quad u_{k+1}\Psi_{k+1} = u_k\Phi\Psi_{k+1} = u_k\Psi_k$$

θ_{k+1} is normalized because it is a generalization of θ'. (6) and (7), with property (4) of the induction hypothesis, imply

$$(8) \quad t_{k+1}\Psi_{k+1} \sim u_{k+1}\Psi_{k+1}$$

and finally,

$$(9) \quad t\theta_{k+1} = t\theta_k\Phi \sim t_k\Phi \sim t_{k+1}$$

$$(10) \quad u\theta_{k+1} = u\theta_k\Phi \sim u_k\Phi \sim u_{k+1} .$$

The opposite ends of the equations (5), (8), (9), and (10) correspond to properties (3), (4), (1), and (2), respectively, for $i = k + 1$. ẍ

<u>Theorem 3</u>: TU2 is correct. That is, if TU2 is given terms t and u and halts with θ, then $t\theta \sim u\theta$.

<u>Proof</u>: We use the notation of Lemma 3. Suppose TU2 halts with $\theta = \theta_k$. Then, using properties (1) and (2) from Lemma 3 plus the fact that $t_k = u_k$, we have

$$t\theta = t\theta_k \sim t_k = u_k \sim u\theta_k = u\theta .$$

So any θ produced by TU2 T-unifies t and u. ẍ

<u>Theorem 4</u>: TU2 is complete, for any complete set of rewrite rules R. Specifically, for any terms t and u, if $t\theta'' \sim u\theta''$, then TU2 generates a T-generalization θ of θ''. That is, there is some Ψ such that $\theta\Psi \sim \theta''$.

<u>Proof</u>: Let $\theta' = N(\theta'')$. We will show that TU2 generates θ such that for some Ψ, $\theta\Psi = \theta'$; thus, θ is a T-generalization of θ''.

Given Lemma 3, we must now show that any path of execution of TU2 that "follows" θ' must terminate; we then apply property (3). Observe that each time through the loop, TU2 selects a substitution Φ such that Φ either allows some subterm of t_i or u_i to become an instance of some p, where p --> q is in R, or Φ directly unifies t and u, thereby substituting at least one subterm of t (or u) for a variable of u (or t). In any case, Φ must eliminate at least one variable name from all of t_i and u_i, although it may introduce some new variable names from a rewrite rule. Since the rewrite rules have the finite termination property, let A be the maximum number of applications of rewrite rules that can be applied to normalize both $t\theta'$ and $u\theta'$. Let B be the maximum number of different variables appearing in any p, where p --> q $\in$ R. Let $C = |Var(t) \cup Var(u)|$. Then the total number of different variables contained in $t\theta'$ and $u\theta'$ is bounded by AB + C. This is a bound on the number of times that the loop of TU2 can be executed while following θ'. Thus, for some k, $k \leq AB + C$, the condition "t=u" must become true, and $t_k = u_k$. So TU2 returns $\theta = \theta_k$, and by property (3) of Lemma 3, $\theta\Psi_k = \theta'$, thus TU2 and this proof are complete. ẍ

As mentioned earlier, TU2 can be generalized to accommodate "permutative reductions", where each rule in R is of the form E(p) --> E(q). E(p) and E(q) represent equivalence classes of terms equal in E, where the equational theory E contains only "permuters" [La 78a]. The appropriate changes to TU2 include the following:

1. L(t,R) ≡ {mgu(t",p)| E(p) --> E(q) ∈ R, and t" is a non-variable subterm of t}. (All elements of the class E(p) are used.)

2. Instead of testing equality of t and u, the revised procedure returns θ if E(t) ≡ E(u).

3. Normalization is by application of permutative reductions as in [La&Ba 77b].

Two interesting questions remain open: (1) Can TU2 be modified so as to be independent? (2) Can a decision procedure exist for T-unification in a decidable theory? Past experience shows that answers to problems related to T-unification do not come easily.

Acknowledgements: I would like to thank my M. S. advisor, Sharon Sickel, for her unflagging encouragement; the referees for their helpful comments and additional references; and Dallas Lankford for his lively correspondence.

References

[Fa 78] Fay, M. J. First-order unification in an equational theory. M. S. Thesis, Information Sciences Board, University of California at Santa Cruz, May 1978.

[Hu 77] Huet, G. Confluent reductions: abstract properties and applications to term rewriting systems. Foundations of Computer Science Conference, Providence, 1977.

[Kn&Be 70] Knuth, D. E., and Bendix, P. B. Simple word problems in universal algebras. In J. Leech (ed.) Computational Problems in Abstract Algebra. Pergamon Press, New York, 1970, pp. 263-297.

[La 75] Lankford, D. S. Canonical inference. Technical Report, Mathematics Dept., University of Texas at Austin, December, 1975.

[La&Ba 77abc] Lankford, D. S., and Ballantyne, A. M. Decision procedures for simple equational theories with a (commutative, permutative, associative) axiom: complete sets of (commutative, permutative, associative) reductions. Technical Reports, Mathematics Dept., University of Texas at Austin, March, April, and August 1977.

[La 78a] Lankford, D. S. The refutation completeness of permutative narrowing and blocked resolution. (Appears elsewhere in this proceedings.)

[La 78b] Lankford, D. S. Private communication.

[Li&Si 75] Livesey, M., and Siekmann, J. Termination and decidability results for string unification. Essex University, Computer Centre Memo CSM-12, 1975.

[Li&Si 77] Livesey, M., and Siekmann, J. Unification of Sets. Internal Report 3/76, Institut fur Informatik I, Universitat Karlsruhe, West Germany, 1977.

[Ma 77] Makanin, G. S. The problem of solvability of equations in a free semigroup. Akad. Nauk. SSSR, TOM 233, no. 2, 1977.

[Ne 74] Nevins, A. J. A human oriented logic for automatic theorem-proving. Journal of the ACM 21, 4 (October 1974), pp. 606-621.

[Pl 72] Plotkin, G. D. Building-in equational theories. In Machine Intelligence 7, B. Meltzer and D. Michie, Eds.

[Ro 65] Robinson, J. A. A machine-oriented logic based on the resolution principle. Journal of the ACM 12, 1 (January 1965), pp. 23-41.

[RS 78] Raulefs, P., and Siekmann, J. Unification of idempotent functions. To appear in J. of Theor. Comp. Sci., 1978.

[RSSU 78] Raulefs, P., Siekmann, J., Szabo, P., and Unvericht, E. A short survey on the state of the art in matching and unification problems. SEKI Project, Institut fur Informatik I, Universitat Karlsruhe, West Germany, 1978.

[Si 75] Siekmann, J. String unification. Essex University Memo CSM-7, 1975.

[Si 76] Siekmann, J. Unification of commutative terms. Internal Report 2, Universitat Karlsruhe, West Germany, 1976.

[Sl 74] Slagle, J. R. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM* *21*, 4 (October 1974), pp. 622-642.

[St 75] Stickel, M. E. A complete unification algorithm for associative-commutative functions. *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*. C/o Artificial Intelligence Laboratory, Massachusetts Institute of Technology, August, 1975, pp. 71-76.

[St&Pe 77] Stickel, M. E., and Peterson, G. E. Complete sets of reductions for equational theories with complete unification algorithms. Technical Report, Dept. of Computer Science, University of Arizona, September 1977.

THE REFUTATION COMPLETENESS OF BLOCKED PERMUTATIVE NARROWING AND RESOLUTION

D. S. Lankford
Louisiana Tech University
Math Department
Ruston, Louisiana 71272

A. M. Ballantyne
University of Texas
Math Department
Austin, Texas 78712

ABSTRACT In this article we establish the refutation completeness of a new kind of canonical inference named blocked permutative narrowing and resolution. In the appendix we show that every equational theory with a complete set of permutative reductions has a complete unification procedure which is implicitly included in the narrowing procedure. The analysis of this connection leads to a refinement of the notion of blocked narrowing and to a general method for constructing (finite) complete unification algorithms.

1. INTRODUCTION

In this article we show how to construct a refutation complete procedure which combines complete sets of permutative reductions [Lan&Bal 77A,B,C] and resolution [Rob 65]. The method we use is an extension of the method developed for ordinary complete sets of reductions by [Lan 75] and [Sla 74].

At the ground level the method is a realization of a ground normal form theorem [Plo 72] which says if $\mathcal{E}$ is a decidable equational theory, η is a normal form function, and $\mathcal{C}$ is a set of clauses with no equations and for which $\mathcal{E} \cup \mathcal{C}$ is equality unsatisfiable, then $\eta(\mathcal{C})$ is unsatisfiable. At the general level the method implicitly includes a procedure which generates a complete set of $\mathcal{E}$-unifiers. An advantage of the method is that it does not require construction of an $\mathcal{E}$-unification procedure in advance, and it is applicable to equational theories $\mathcal{E}$ which do not have a complete $\mathcal{E}$-unification algorithm, such as commutative ring theory.

The methods developed here can be extended to the term rewriting systems of [Hue 77] and [Sti&Pet 77], and also to the cases when the set of permutative reductions is not complete and equations occur in non-unit clauses as was done by [Lan 75]. The methods can also be extended to other refutation complete systems, such as the chaining and narrowing system of [Lan&Mus 78]. And, of course, these methods are applicable to human-oriented and natural deduction systems like those of [Ble&Bru 74] and [Nev 74].

2. BLOCKED PERMUTATIVE NARROWING AND RESOLUTION

As a prelude to the definitions of blocked permutative narrowing and resolution we summarize some relevant facts about permutative reduction [Lan&Bal 77A,B,C]. Let $\mathcal{P}$ be a finite set of equations and let $\approx_\rho$ be the congruence relation defined by $t \approx_\rho u$ iff $\vdash_\rho t = u$. Such a set $\mathcal{P}$ is called a set of permuters in case for each term t, the $\approx_\rho$-equivalence class of t, denoted $\approx_\rho(t)$, is finite. We do not know if it is decidable whether an arbitrary $\mathcal{P}$ is a set of permuters. A set of permutative rewrite rules relative to $\mathcal{P}$ is a finite set $\mathcal{R}$ of expressions $\approx_\rho(L) \longrightarrow \approx_\rho(R)$ where L and R are terms and each variable symbol which occurs in a member of $\approx_\rho(R)$ also occurs in each member of $\approx_\rho(L)$. We say $\approx_\rho(u)$ is an immediate $\mathcal{P}$-reduction of $\approx_\rho(t)$ relative to $\mathcal{R}$ in case there is some $\approx_\rho(L) \longrightarrow \approx_\rho(R) \in \mathcal{R}$, $t' \in \approx_\rho(t)$, $u' \in \approx_\rho(u)$, $L' \in \approx_\rho(L)$, $R' \in \approx_\rho(R)$, and a substitution θ such that u' is the result of replacing one occurrence of $L'\theta$ in t' by $R'\theta$. When $\mathcal{P}$ is a set of permuters and $\mathcal{R}$ is a set of permutative rewrite rules relative to $\mathcal{R}$ we say the pair $\mathcal{P},\mathcal{R}$ is Noetherian in case there is no infinite sequence $\approx_\rho(t_1)$, $\approx_\rho(t_2)$, $\approx_\rho(t_3)$, ... with $\approx_\rho(t_{i+1})$ an immediate $\mathcal{P}$-reduction of $\approx_\rho(t_i)$ relative to $\mathcal{R}$, $i = 1,2,3,\ldots$, and we say the pair $\mathcal{P},\mathcal{R}$ is Church-Rosser in case for each $\approx_\rho(t)$ and each pair of immediate $\mathcal{P}$-reductions $\approx_\rho(u)$ and $\approx_\rho(v)$ of $\approx_\rho(t)$ relative to $\mathcal{R}$, there is some $\approx_\rho(w)$ and finite sequences $\approx_\rho(u_1)$, ..., $\approx_\rho(u_m)$ and $\approx_\rho(v_1)$, ..., $\approx_\rho(v_n)$ such that $\approx_\rho(u_m) = \approx_\rho(v_n) = \approx_\rho(w)$ where $\approx_\rho(u_{i+1})$ is an immediate $\mathcal{P}$-reduction of $\approx_\rho(u_i)$ relative to $\mathcal{R}$, $i = 1,\ldots,m-1$, and $\approx_\rho(v_{j+1})$ is an immediate $\mathcal{P}$-reduction of $\approx_\rho(v_j)$ relative to $\mathcal{R}$, $j = 1,\ldots,n-1$. A complete set of permutative reductions is a pair $\mathcal{P},\mathcal{R}$ where $\mathcal{P},\mathcal{R}$ is Noetherian and Church-Rosser.

It is undecidable whether $\mathcal{P},\mathcal{R}$ is Noetherian, see, for example, [Hue&Lan 78] and [Lip&Sny 77]. However, practical Noetherian procedures have been developed, see, for example [Der&Man 78], [Gua... 69], [Hue&Lan 78], [Knu&Ben 70], [Lan 75], [Man&Nes 70], and [Lip&Sny 77]. It is decidable for ordinary Noetherian $\mathcal{R}$ (empty $\mathcal{P}$) whether $\mathcal{R}$ is Church-Rosser, see [Knu&Ben 70] and [Lan 75]. For certain other $\mathcal{P}$ it is also decidable whether Noetherian $\mathcal{P},\mathcal{R}$ is Church-Rosser, see [Lan&Bal 77A,B,C] and [Sti&Pet 77]. Also, very general Church-Rosser tests have been developed by [Hue 77]. But in general it is not known whether it is decidable if Noetherian $\mathcal{P},\mathcal{R}$ is Church-Rosser. For $\mathcal{P}$ containing just one associative equation, [Sti&Pet 77] have shown that a Church-Rosser decision procedure for Noetherian $\mathcal{P},\mathcal{R}$ would decide the word equation problem, a difficult problem in associative calculi which has only recently been solved [Mak 77].

Throughout the remainder of this paper let $\mathcal{P},\mathcal{R}$ be a complete set of permutative reductions. It should be evident that an algorithm * can be described so that if t is any term then t^* is the result of reducing t by $\mathcal{R}$ as far as possible. We say that $\approx_\rho(t)$ is $\mathcal{P},\mathcal{R}$-irreducible in case there is no immediate $\mathcal{P}$-reduction of $\approx_\rho(t)$ relative to $\mathcal{R}$. A $\mathcal{P},\mathcal{R}$-clause is like an ordinary clause, except that it is composed of $\mathcal{P},\mathcal{R}$-literals, expressions of the form $\pm P(\approx_\rho(t_1),\ldots,\approx_\rho(t_n))$ where all the $\approx_\rho(t_i)$ are $\mathcal{P},\mathcal{R}$-irreducible. A $\mathcal{P},\mathcal{R}$-substitution is like an ordinary substitution, except that it has the form $\theta = \{\approx_\rho(t_1)/v_1,\ldots,\approx_\rho(t_m)\}$ where the $\approx_\rho(t_i)$ are $\mathcal{P},\mathcal{R}$-irreducible. Because $\approx_\rho$ is a congruence relation the substitution instance of a $\mathcal{P},\mathcal{R}$-literal $\pm P(\approx_\rho(t_1),\ldots,\approx_\rho(t_n))$ by a $\mathcal{P},\mathcal{R}$-substitution $\phi = \{\approx_\rho(u_1)/v_1,\ldots,\approx_\rho(u_m)/v_m\}$ may be defined as $\pm P(\approx_\rho(t_1\phi'),\ldots,\approx_\rho(t_n\phi'))$ where $\phi' = \{u_1/v_1,\ldots,u_m/v_m\}$. The definition of substitution instance extends to $\mathcal{P},\mathcal{R}$-clauses in the obvious way. A substitution instance of a $\mathcal{P},\mathcal{R}$-clause is $\mathcal{P},\mathcal{R}$-blocked in case the instance is a $\mathcal{P},\mathcal{R}$ clause.

An <u>immediate $\mathcal{P}$-inference</u> of a $\mathcal{P},\mathcal{R}$-clause $\mathcal{C}$ is obtained as follows. Let $\approx_{\mathcal{P}}(t)$ be an argument of some predicate of $\mathcal{C}$, let $t' \in \approx_{\mathcal{P}}(t)$, let $p = q \in \mathcal{P}$, let u be a <u>paramodulant</u> [Rob&Wos 69] of t' by $p = q$ on a subterm of t' which is not a variable, let μ be the unifier of that paramodulation (it is assumed that $p = q$ was standardized away from $\mathcal{C}$), and let $\mathcal{C}'$ be obtained from $\mathcal{C}\mu$ by replacing $\approx_{\mathcal{P}}(t)\mu$ by $\approx_{\mathcal{P}}(u)$. The immediate $\mathcal{P}$-inferences of $\mathcal{C}$ are the clauses $\mathcal{C}'$ as described above. An immediate $\mathcal{P}$-inference is called <u>$\mathcal{P},\mathcal{R}$-blocked</u> in case it is $\mathcal{P},\mathcal{R}$-irreducible. The refutation completeness of the restriction of paramodulation to subterms which are not variables was reported by [Bra 75]. An <u>immediate $\mathcal{R}$-narrowing</u> of a $\mathcal{P},\mathcal{R}$-clause $\mathcal{C}$ is obtained as follows. Let $\approx_{\mathcal{P}}(t)$ be an argument of some predicate of $\mathcal{C}$, let $t' \in \approx_{\mathcal{P}}(t)$, let $\approx_{\mathcal{P}}(L) \longrightarrow \approx_{\mathcal{P}}(R) \in \mathcal{R}$, let u be a paramodulant of t' by $L = R$ on L (one way substitution) on a subterm of t' that is not a variable, let μ be the unifier of that paramodulation, let $\mathcal{C}'$ be obtained from $\mathcal{C}\mu$ by replacing $\approx_{\mathcal{P}}(t)\mu$ by $\approx_{\mathcal{P}}(u)$, and let $\mathcal{C}'' = (\mathcal{C}')^*$. The immediate $\mathcal{R}$-narrowings of $\mathcal{C}$ are the clauses $\mathcal{C}''$ as described above. An immediate $\mathcal{R}$-narrowing is called <u>$\mathcal{P},\mathcal{R}$-blocked</u> in case the associated unifier of paramodulation μ is $\mathcal{P},\mathcal{R}$-irreducible. A <u>$\mathcal{P},\mathcal{R}$-resolvent</u> of two $\mathcal{P},\mathcal{R}$ clauses $\mathcal{C}$ and $\mathcal{D}$ is formed as follows. There are several ways that ordinary resolution can be defined, such as the key triple [Rob 65] or the factoring and binary resolution approach, as well as the many refinements. Here we use the factoring and binary resolution approach, but it appears that any other approach could be developed along these lines. Let $P(\approx_{\mathcal{P}}(t_{i1},\ldots,\approx_{\mathcal{P}}(t_{in}))$, $i = 1,\ldots,k$ be in $\mathcal{C}$, let t_{ij}' be in $\approx_{\mathcal{P}}(t_{ij})$, and let μ be the most general simultaneous unifier of $P(t_{i1}',\ldots,t_{in}')$, $i = 1,\ldots,k$. $\mathcal{C}\mu$ is called <u>a positive immediate $\mathcal{P},\mathcal{R}$-factor of $\mathcal{C}$</u>. A negative immediate factor is defined similarly. An <u>immediate factor</u> is a positive or negative immediate factor. A <u>factor</u> is an immediate factor of ... of an immediate factor. A factor is called <u>$\mathcal{P},\mathcal{R}$-blocked</u> in case it is $\mathcal{P},\mathcal{R}$-irreducible and all the immediate factors in the sequence producing it are $\mathcal{P},\mathcal{R}$-irreducible. It can be seen that there are finitely many factors of a clause. A <u>(binary) $\mathcal{P},\mathcal{R}$-resolvent</u> of $\mathcal{C}$ and $\mathcal{D}$ is $(\mathcal{A} - P) \cup (\mathcal{B} - Q)\mu$, where $\mathcal{A}$ and $\mathcal{B}$ are $\mathcal{P},\mathcal{R}$-blocked factors of $\mathcal{C}$ and $\mathcal{D}$, $P(\approx_{\mathcal{P}}(t_1),\ldots,\approx_{\mathcal{P}}(t_m))$ and $Q = \neg P(\approx_{\mathcal{P}}(u_1),\ldots,\approx_{\mathcal{P}}(u_m))$ are members of $\mathcal{A}$ and $\mathcal{B}$, t_i' are members of $\approx_{\mathcal{P}}(t_i)$, and μ is the most general unifier of $P(t_1',\ldots,t_m')$ and $P(u_1,\ldots,u_m)$.

<u>Theorem</u> If $\mathcal{P},\mathcal{R}$ is a complete set of permutative reductions, $\mathcal{C}$ is a set of clauses which do not contain the equality predicate, $\mathcal{E}(\mathcal{R}) = \{L = R \mid \approx_{\mathcal{P}}(L) \longrightarrow \approx_{\mathcal{P}}(R) \in \mathcal{R}\}$, $\mathcal{E}(\mathcal{R}) \cup \mathcal{P} \cup \mathcal{C}$ is equality unsatisfiable, $\mathcal{C}'$ is the result of replacing each term t of $\mathcal{C}$ by $\approx_{\mathcal{P}}(t)$, and $\mathcal{C}^*$ is the result of fully permutatively reducing $\mathcal{C}'$, then there is a refutation of the empty clause from $\mathcal{C}^*$ by immediate $\mathcal{P}$-inference, immediate $\mathcal{P}$-narrowing, and $\mathcal{P},\mathcal{R}$-resolution with all three kinds of inference $\mathcal{P},\mathcal{R}$-blocked.

<u>Proof</u> When all clauses of $\mathcal{C}$ are ground, the proof follows from [Plo 72] using $*$ as the normal form function. In the ground case there are no $\mathcal{P}$-inferences or $\mathcal{P}$-$\mathcal{R}$ narrowings. $\mathcal{C}^*$ is unsatisfiable, and $\mathcal{P},\mathcal{R}$-resolution is just ordinary resolution. For the general case the above ground result can be lifted similar to the ordinary case

as was done by [Lan 75]. The lifting involves showing that if $\mathcal{D}$ is an instance of a clause $\mathcal{C}$ then there is a clause $\mathcal{C}'$ which is obtained from $\mathcal{C}$ by finitely many $\mathcal{P}$-inferences and $\mathcal{R}$-narrowings such that $\mathcal{C}'$ has $\mathcal{D}$ as an instance. This says that all $\mathcal{P}$-inferencing and $\mathcal{R}$-narrowing can be done before any blocked $\mathcal{P},\mathcal{R}$ resolving is done. This restriction for ordinary resolution and paramodulation was noted by [And 70]. Of course, in general it cannot be decided how many rounds of blocked $\mathcal{P}$-inferencing and $\mathcal{R}$-narrowing are needed. We illustrate blocked $\mathcal{P}$-inference, $\mathcal{R}$-narrowing, and $\mathcal{P},\mathcal{P}$-resolution with the following example.

Example 1 If R is a ring, I is an ideal, and $1 \in I$, then $I = R$. A fragment of the proof is given below.

P1. $x + y = y + x$

P2. $(x + y) + z = x + (y + z)$

P3. $xy = yx$

P4. $(xy)z = x(yz)$

R1. $\approx(x + 0) \longrightarrow \approx(x)$

R2. $\approx(x + (-x)) \longrightarrow \approx(0)$

R3. $\approx(-0) \longrightarrow \approx(0)$

R4. $\approx(-(-x)) \longrightarrow \approx(x)$

R5. $\approx(-(x + y)) \longrightarrow \approx((-x) + (-y))$

R6. $\approx(x1) \longrightarrow \approx(x)$

R7. $\approx(x0) \longrightarrow \approx(0)$

R8. $\approx((-x)y) \longrightarrow \approx(-(xy))$

R9. $\approx(x(y + z)) \longrightarrow \approx((xy) + (xz))$

H1. $\approx(x) \notin I$, $\approx(y) \notin R$, $\approx(xy) \in I$

H2. $\approx(1) \in I$

D1. $\approx(a) \in R$

D2. $\approx(a) \notin I$

1. $\approx(1) \notin I$, $\approx(y) \notin R$, $\approx(y) \in I$ blocked immediate narrowing of H1 by R6
2. $\approx(y) \notin R$, $\approx(y) \in I$ blocked resolution H1, 1
3. $\approx(a) \in I$ blocked resolution D1, 2
4. empty clause blocked resolution D2, 3

Notice that H1 and H2 would not be allowed to resolve, since the resolvent would not be blocked.

3. CONCLUDING REMARKS

In this article we have shown how to construct a refutation complete procedure which combines complete sets of permutative reductions and resolution. The results can be extended in several directions. First, most interesting theorems fail to be of the form such that the main theorem applies, since equality predicates occur in non-unit clauses and sets of unit equations are often incomplete. We are confident that methods like those of [Lan 75] can be developed for these cases. Next, some $\mathcal{P}$ have (finite) complete unification algorithms. In that case $\mathcal{P}$-inference can be dispensed with provided that $\mathcal{R}$-narrowing and $\mathcal{P},\mathcal{R}$-resolution are defined appropriately in terms of $\mathcal{P}$-unification. Finally, there are some $\mathcal{P}$ with (finite) complete unification algorithms for which $\approx_{\mathcal{P}}(t)$ are not finite, such as associativity, commutativity, and idempotence. In this case, explicit mention of equivalence classes can be dispensed with provided that $\mathcal{R}$-narrowing and $\mathcal{P},\mathcal{R}$-resolution are defined in terms of the $\mathcal{P}$-unification algorithm.

ACKNOWLEDGMENTS We would like to thank the referees for pointing out the fact that narrowing implicitly included a procedure which generates a complete set of unifiers for those equational theories with complete sets of reductions. We would also like to thank Prof. W. W. Bledsoe for his encouragement through the Automatic Theorem Proving Project at the University of Texas. Finally, we appreciate conversations and correspondence with G. Huet and G. Plotkin on various aspects of reduction and unification. This work was supported in part by NSF Grant DCR74-12866.

REFERENCES

[And 70] R. Anderson Some theoretical aspects of automatic theorem proving. Ph.D. Dissertation, Univ. of Texas, Austin, Texas, Aug. 1970.

[Ble&Bru 74] W. Bledsoe and P. Bruell A man-machine theorem proving system. Artif. Intell. 5, 1 (Spring 1974), 51-72.

[Bra 75] D. Brand Proving theorems with the modification method. Siam J. Comput. 4, 4 (Dec 1975), 412-430.

[Der&Man 78] N. Dershowitz and Z. Manna Proving termination with multiset orderings. Stanford Univ., Comp. Sci. Dept., Stanford CA 94305, Report No. STAN-CS-78-651, 1978.

[Fay 78] M. Fay First order unification in an equational theory (extended abstract). (to appear in the 1979 Deduction Workshop Proceedings, Univ. of Texas, Austin, Texas)

[Gua... 69] J. Guard, F. Oglesby, J. Bennett, and L. Settle Semi-automated mathematics. JACM 16, 1 (Jan. 1969), 49-62.

[Hue 77] G. Huet Confluent reductions: abstract properties and applications to term rewriting systems. In Proc. of 18th Annual IEEE Symp. on Foundations of Comp. Sci.

[Hue&Lan 78] G. Huet and D. Lankford On the uniform halting problem for term rewriting systems. IRIA Laboria, Rocquencourt, B. P. No. 105, 78150 - Le Chesnay, France, Rapport de Recerche No. 283, Mars 1978.

[Knu&Ben 70] D. Knuth and P. Bendix Simple word problems in universal algebras. In Computational Problems in Abstract Algebra, J. Leech, Ed., Pergamon Press, 1970.

[Lan 75] D. Lankford Canonical inference. Univ. of Texas, Math Dept., Automatic Theorem Proving Project, Austin, Texas 78712, Report ATP-32, Dec. 1975.

[Lan&Bal 77A] D. Lankford and A. Ballantyne Decision procedures for simple equational theories with commutative axioms: complete sets of commutative reductions. Automatic Theorem Proving Project, Report ATP-35, Mar. 1977.

[Lan&Bal 77B] D. Lankford and A. Ballantyne Decision procedures for simple equational theories with permutative axioms: complete sets of permutative reductions. Automatic Theorem Proving Project, Report ATP-37, Apr. 1977.

[Lan&Bal 77C] D. Lankford and A. Ballantyne Decision procedures for simple equational theories with commutative-associative axioms: complete sets of commutative-associative reductions. Automatic Theorem Proving Project, Report ATP-39, Aug. 1977.

[Lan&Mus 78] D. Lankford and D. Musser On semideciding first order validity and invalidity. (to appear as an ISI report) USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291.

[Lip&Sny 77] R. Lipton and L. Snyder On the halting of tree replacement systems. Conference on Theoretical Computer Science, Univ. of Waterloo, Jul. 1977.

[Mak 77] G. Makanin The problem of solvability of equations in a free semigroup. Soviet Math. Dokl. 18, 2 (1977), 330-334.

[Man&Nes 70] Z. Manna and S. Ness On the termination of Markov algorithms. Proc. Third Intl. Conf. on System Sciences, Honolulu, HI, Jan. 1970, 789-792.

[Nev 74] A. Nevins A human oriented logic for automatic theorem proving. JACM 21, 4 (Oct. 1974), 606-621.

[Plo 72] G. Plotkin Building-in equational theories. Machine Intelligence 7, B. Meltzer and D. Michie, Eds., John Wiley and Sons, New York - Toronto, 1972, 73-89.

[Rau&Sie 78] P. Raulefs and J. Siekmann Unification of idempotent functions. Institut fur Informatik I, Universitat Karlsruhe, Postfach 6380, D-7500, Karlsruhe 1, West Germany, 1978.

[Rau... 78] P. Raulefs, J. Siekmann, P. Szabo, and E. Unvericht A short survey on the state of the art in matching and unification problems. SEKI-Project, Institut fur Informatik I, Universitat Karlsruhe, Postfach 6380, D-7500, Karlsruhe 1, GFR, 1978.

[Rob&Wos 69] G. Robinson and L. Wos Paramodulation and theorem proving in first-order theories with equality. Machine Intelligence 4, B. Meltzer and D. Michie, Eds., Edinburgh University Press, 1969, 135-150.

[Rob 65] J. Robinson A machine-oriented logic based on the resolution principle. JACM 12, 1 (Jan. 1965), 23-41.

[Sla 74] J. Slagle Automated theorem proving for theories with simplifiers, commutativity, and associativity. JACM 21, 4 (Oct. 1974), 622-642.

[Sti&Pet 77] M. Stickel and G. Peterson Complete sets of reductions for equational theories with complete unification algorithms. The Univ. of Arizona, Dept. of Comp. Sci., Tucson, Arizona 85721, Sept. 1977.

APPENDIX

In the following discussion a unification procedure may have infinitely many ugu's while a unification algorithm has finitely many. We mentioned above that the referees for this article pointed out that narrowing implicitly includes a procedure which generates a complete set of unifiers for any theory with a complete set of permutative reductions. Since this development would not have the opportunity to be reviewed, we have included it as an appendix. It has also been brought to our attention that the existence of a complete unification procedure for equational theories with a complete set of reductions has been shown by [Fay 78], and a reading of that paper has assisted in the following development.

Here we develop the connection between ordinary narrowing and unification, and outline how the ordinary case can be extended to the permutative case. There are several reasons for our approach. First, the ordinary case is simpler than the permutative case. Second, the development of the ordinary case leads to a refinement of the notion of narrowing, which adds complexity to the permutative case. Third, in the ordinary case we will show that if each term has finitely many blocked narrowings, then there is a unification algorithm for the theory. A similar result holds for permutative reductions when the set of permuters consists of commutative axioms or commutative-associative axiom pairs. We conjecture that this result also holds for any complete set of permutative reductions which have a complete unification algorithm for $\mathcal{P}$. It is evident from these remarks that the connection between narrowing and unification is more than coincidental. Indeed, the narrowing approach is a general method for constructing complete unification procedures, and can be used to obtain many of the known results on theory-unification [Rau... 78], as well as some new results.

The connection between ordinary narrowing and unification can be shown as follows. Throughout, let $\mathcal{R}$ denote a complete set of reductions and $*$ the $\mathcal{R}$-normal form function for the equational theory $\mathcal{E}(\mathcal{R})$. An extended narrowing is defined as a term and a substitution. The initial extended narrowing of a term t it the pair t^*, ϵ where ϵ is the empty substitution. Recall that when (in the old sense) an immediate narrowing of t^* is formed, there is an associated mgu μ_{t^*} which unifies L (of some $L \longrightarrow R$ in $\mathcal{R}$) and a subterm of t^*, and that we require μ_{t^*} to be blocked ($\mathcal{R}$-irreducible). The extended immediate narrowings of t^* are defined to be u, μ_{t^*}. Suppose that u, θ is an extended narrowing of t and v is a blocked immediate narrowing of u with associated mgu μ_u (which was formed by standardizing apart from u, θ). A blocked extended narrowing is $v, \theta \cdot \mu_u$ provided that $\theta \cdot \mu_u$ is blocked.

A complete set of $\mathcal{E}(\mathcal{R})$-unifiers for two terms t and u can be formed as follows. Let t^N, ψ and u^N, λ be blocked extended narrowings of t and u which have been standardized apart. Let μ be the mgu of t^N and u^N and $\theta_{t,u} = (\psi \cup \lambda)\cdot\mu$ provided $(\psi \cup \lambda)\cdot\mu$ is blocked. The set of all $\theta_{t,u}$ can be shown to be a complete set of $\mathcal{E}(\mathcal{R})$-unifiers for t and u by an argument similar to the lifting of blocked resolution and anrrowing [Lan 75].

It is evident that when each term has finitely many blocked extended narrowings then the equational theory has a complete unification algorithm. For example, it has been shown [Rau&Sie 78] that $\{xx \longrightarrow x\}$ has a complete unification algorithm. This fact follows easily from the narrowing approach by observing that each narrowing by idempotence reduces the number of variable symbols. Unification algorithms for other theories, like $\{x1 \longrightarrow x\}$, $\{x(x^{-1}) \longrightarrow 1\}$, and so on, follow by similar arguments. The sets of unifiers produced by blocked extended narrowing are not generally minimal, since, for example, in the case of idempotence $\mathcal{R} = \{xx \longrightarrow x\}$, the terms xb and (ay)b have unifiers $\theta_1 = \{ay/x\}$ and $\theta_2 = \{a/x, b/y\}$, and θ_2 is an $\mathcal{E}(\mathcal{R})$-instance of θ_1. However, θ_2 is not a blocked instance of θ_1, and so minimizing the set of $\mathcal{E}(\mathcal{R})$-unifiers often has the undesirable side effect of requiring unblocked inferences if refutation completeness is to be maintained.

Extended blocked narrowing can also be defined for the permutative case by redefining $\mathcal{P}$-inference, $\mathcal{R}$-narrowing, and $\mathcal{P},\mathcal{R}$-resolution to produce clause-substitution pairs. Each time a new object $\mathcal{O}', \theta'$ is formed from an old one $\mathcal{O}, \theta$, the associated mgu μ satisfies $\theta' = \theta \cdot \mu$. Now, in addition to the previous blocked requirements, we may require that θ' be blocked. When $\mathcal{P}$ has a complete unification algorithm we believe that $\mathcal{P}$-inference can be dispensed with by defining $\mathcal{R}$-narrowing and $\mathcal{P},\mathcal{R}$-resolution in terms of the $\mathcal{P}$-unification algorithm. For a set of commutative of commutative-associative functions it can be done by allowing narrowings by embeddings (in the C + A case) and replacing unification by commutative or commutative-associative unification. With this definition of narrowing, the result that A + C + I has a complete unification algorithm [Rau&Sie 78] can be shown by the narrowing approach.

UNIFICATION PROBLEMS FOR COMBINATIONS OF ASSOCIATIVITY, COMMUTATIVITY, DISTRIBUTIVITY AND IDEMPOTENCE AXIOMS

M. Livesey,

Dept. of Computer Science
University of St. Andrews
North Haugh
St. Andrews
Scotland

J. Siekmann, P. Szabó, E. Unvericht

Universität Karlsruhe
Institut für Informatik I
Postfach 6380
D-7500 Karlruhe 1
Germany

0. *Introduction*

For almost as long as attempts at proving theorems by machines have been made, a critical problem has been well known [1],[2]: Certain equational axioms, if left without precautions in the data base of an automatic theorem prover (ATP), will force the ATP to go astray. In 1967, Robinson [21] proposed that substantial progress ("a new plateau") would be achieved by removing these troublesome axioms from the data base and building them into the deductive machinery.

Four approaches to cope with equational axioms have been proposed:

(1) To write the axioms into the data base, and use an additional rule for inferences, such as paramodulation [22].

(2) To use special "rewrite rules" [11],[13],[10].

(3) To design special inference rules incorporating these axioms [25].

(4) To develop special unification algorithms incorporating these axioms [18].

At least for equational axioms, the last approach (4) appears to be most promising, however it has the drawback that for every new set of axioms a new unification algorithm has to be found. Also recently there has been interesting work on combinations of approach (2) and (4), [17].

The theoretical basis for utilizing unification algorithms incorporating equational axioms has been developed by G. Plotkin [18]. Plotkin has shown that whenever an ATP is to be refutation complete, its unification procedure must satisfy conditions given as follows: Assume T is the theory (set of axioms) considered, and t_1, t_2 are the terms to be unified; then, the following properties should hold for the set Σ of unifiers of t_1 and t_2:

1. Σ is correct, i.e. for every $\sigma \in \Sigma, \sigma(t_1) \overset{T}{=} \sigma(t_2)$ ($\overset{T}{=}$ denotes equality with respect to the theory T).
2. Σ is complete, i.e. for any substitution δ with $\delta(t_1)=\delta(t_2)$ there is some $\sigma \in \Sigma$ s.t. there exists a substitution λ so that $\delta = \lambda \bullet \sigma$.
3. Σ is minimal, i.e. no unifier σ_1 in Σ is an instance of some other unifier σ_2 in Σ.

Minimality is a property that has often been overlooked in the literature: If minimality is completely ignored we arrive at simply enumerating all substitutions and removing all that do not unify as an algorithm satisfying our requirements. Generating such a set of all unifiers, instead of a set of most general unifiers, essentially amounts to proving a theorem by the 'British Museum Algorithm' (i.e. by enumerating all Herbrand instances). Such procedures are called conservative in [22] and are distinctively different from proofs by the resolution principle at the 'lifted' most general level. However, minimality is more difficult to achieve than correctness and completeness.

Looking at unification of terms in first-order predicate calculus with an equational theory T, unification problems may be classified with respect to the cardinality of minimal and complete sets Σ of unifiers:

(i) Σ may always be a singleton: e.g. for $\Sigma = \emptyset$, that is for ordinary first order unification as in [20],[1],[11]. Another (trivial) problem in this class is the string matching problem for constant strings only, as encountered in string manipulation languages such as SNOBOL [4]. The nontrivial aspect of this problem is to find efficient algorithms [12],[5]. Another example is unification under homomorphism, isomorphism and automorphism [29].

(ii) Σ may have more than one element but at most finitely many: examples are the theory of idempotence plus commutativity [19]. Other examples are unification under commutativity [24]; unification under associativity and commutativity [26],[15]; unification under associativity, commutativity and idempotence [15] and the one way unification problem for strings.

(iii) Σ may sometimes be an infinite set: an example is unification under associativity [18],[23], [14]. This problem is equivalent to the problem of solving a set of equations over a free semigroup (the monoid problem) [8], the decidability of which had been an open problem for over twentyfive years [16]. Another problem in this class is unification under distributivity [28].

(iv) Σ may sometimes not exist at all, e.g. for unification in ω-order predicate calculus. In such cases there exist infinite chains of unifiers (ordered by increasing generality) with no upper bound [6].

For unification problems where complete sets of unifiers are always finite, it is not necessarily important that the unification procedure returns a minimal set of unifiers, since dependent unifiers can always be checked off. In this case, minimality of the unification procedure comes down to be a matter of computational efficiency.

Unification problems have significance beyond automatic theorem proving: certain axioms define structures which closely resemble familiar data structure (e.g. strings, sets, multisets, etc.), and most AI-languages have pattern matching algorithms for these cases built into their machinery (see e.g. [7]). Apart from the fact that these matching algorithms have without exception been designed ad hoc, i.e. without respect to completeness, minimality or sometimes even correctness, the basic question of whether a particular matching problem for a particular data structure is decidable has not been answered.

A little reflection will show that for very rich matching structures, as it has e.g. been proposed in MATCHLESS in PLANNER [7], the matching problem is undecidable. This presents a problem for the designer of such languages: on the one hand, very rich and expressive matching structures are desirable, since they form the basis for the invocation and deduction mechanism. On the other hand, drastic restrictions will be necessary if matching algorithms are to be found. The question is just how severe do these restrictions have to be.

Unification algorithms are also relevant for data base problems, information retrieval and computer vision.

The following chart provides a quick survey of unification problems as investigated so far and § 2-6 give a short exposition of [27],[28],[15],[24].
The axioms are:

A (associativity)	$f(f(x,y),z) = f(x,f(y,z))$
C (commutativity)	$f(x,y) = f(y,x)$
D (distributivity)	$f(x,g(y,z)) = g(f(x,y),f(x,z))$
	$f(g(x,y),z) = g(f(x,z),f(y,z))$
H (homomorphism)	$\varphi(x \circ y) = \varphi(x) \bullet \varphi(y)$
I (idempotence)	$f(x,x) = x$

Axioms	Problem decidable?	card(Σ)	rep. Alg. is minimal	Datastructure	Investigators
A	yes	∞	yes	strings	[8],[23], [14],[16]
C	yes	fin.	no	unordered trees (lists)	[24]
I	yes	fin.	no		[19]
A+C	yes	fin.	yes	finite multisets	[26] [15]
A+I	?	?	?		in progress
C+I	yes	fin.	no		[19]
A+C+I	yes	fin.	yes	finite sets	[15]
D	?	∞	?		[28]
D+A	no	∞	?		[27]
D+I	?	?	?		in progress
D+C	?	∞	?		in progress
D+C+I	?	?	?		in progress
D+A+C	no	∞	?		[27]
H	yes	1	yes		[29]
H+A	yes	∞	yes		[29]
H+A+C	?	∞	?		[29]
ω-order terms	$\omega \geq 3$ no	doesn't exist	no		[9]
first order terms	yes	1	yes	ordered trees (lists)	[1],[20], [11]

1. Notation and Definitions

Let $\mathcal{C}$ be a countable set of *constants* and $\mathcal{V}$ a countable set of *variables*. The set TERM is the least set such that (i) $t \in \mathcal{V}, \mathcal{C}$ then $t \in$ TERM and (ii) $t_1, t_2 \in$ TERM then $f(t_1,t_2) \in$ TERM. Terms in $\mathcal{V}, \mathcal{C}$ are called *primitive* and *nonprimitive* otherwise. VAR(t) is the set of variables occuring in term t.

A *substitution* σ is a total mapping from $\mathcal{V}$ to TERM with $\sigma(v) = v$ almost everywhere. We denote σ as a set of pairs $\sigma = \{(v_1|t_1),(v_2|t_2),\dots,(v_n|t_n)\}$; $v_i \in \mathcal{V}$, $t_i \in$ TERM, where σ maps v_i to t_i; $1 \le i \le n$. Substitutions are extended to mappings from TERM to TERM by the usual homomorphism: $\sigma(f(t_1,t_2)) = f(\sigma(t_1),\sigma(t_2))$. *Composition* of substitutions $\sigma_1 \bullet \sigma_2$ is the functional composition: $\sigma_1 \bullet \sigma_2(t) = \sigma_1(\sigma_2(t))$. The set of substitutions is called SUB.

A *unifier* of two terms t_1, t_2 is a substitution δ such that $\delta(t_1) = \delta(t_2)$. Two terms t_1, t_2 are *equal under a theory* T, $t_1 \overset{\mathrm{T}}{=} t_2$, iff $\vdash_{\mathrm{T}} t_1 = t_2$. Two substitutions σ and τ are *V-equal* under T ($V \subset \mathcal{V}$), $\sigma \overset{\mathrm{T}}{\underset{V}{=}} \tau$ iff $\forall v \in V$: $\sigma(v) \overset{\mathrm{T}}{=} \sigma(v)$. A *T-unifier* for t_1, t_2 is a substitution δ s.th. $\delta(t_1) \overset{\mathrm{T}}{=} \delta(t_2)$. A unification problem under T is denoted as $\langle t_1;t_2\rangle_{\mathrm{T}}$. A T-unification problem under the empty theory $\emptyset$ is denoted as $\langle t_1;t_2\rangle_{\emptyset}$ and unified according to [20], the resulting unifier is called an R-unifier. The set of all T-unifiers for $\langle t_1;t_2\rangle_{\mathrm{T}}$ is denoted as $\Psi(\langle t_1;t_2\rangle_{\mathrm{T}})$. A *most general T-unifier* σ for $\langle t_1;t_2\rangle_{\mathrm{T}}$ is a T-unifier s.th. for any $\delta \in$ SUB with $\delta(t_1) \overset{\mathrm{T}}{=} \delta(t_2)$ there exist a λ s.th. $\delta \overset{\mathrm{T}}{=} \lambda \bullet \sigma$. And since in general a single most general unifier may not exist, we define Σ, *the set of most general unifiers*, for t_1 and t_2 as: (i) $\forall \sigma \in \Sigma: \sigma(t_1) \overset{\mathrm{T}}{=} \sigma(t_2)$ (correctness); (ii) for any unifier δ $\exists \sigma \in \Sigma$ and some $\lambda \in$ SUB such that $\delta \overset{\mathrm{T}}{=} \lambda \bullet \sigma$ (completeness); (iii) for no $\sigma_i, \sigma_k \in \Sigma$: $\sigma_i \overset{\mathrm{T}}{=} \sigma_k \bullet \lambda$ for $\lambda \in$ SUB (minimality).

2. The Undecidability of the D_A-Unification Problem

2.1 Introduction

It is shown that the decision problem for D_A-unification (that is: to determine whether or not two terms involving a function which distributes over an associative function are unifiable) is related to the decision problem of Hilbert's (modified) tenth problem (that is: to determine, of a diophantine equation with integer coefficients, whether or not it has a solution in nonnegative integers). Let $\{\#, \square\}$ be a set of two binary functionsymbols, then we modify the definition of TERM in 1. by using infix notation i.e.:

(i) $t \in \mathcal{V} \cup \mathcal{C}$ then $t \in$ TERM

(ii) $s,t \in$ TERM then $(s\#t), (s\square t) \in$ TERM.

Occasionally we shall deliberatly confuse the objects denoted by a term t with the name t, treated as a sequence of symbols (i.e. strings). The set of equational axioms D_A with $x,y,z \in \mathcal{V}$ is:

$$\left.\begin{array}{ll} \textit{Distributivity:} & (x\square(y\#z)) = ((x\square y)\#(x\square z)) \\ & ((x\#y)\square z) = ((x\square z)\#(y\square z)) \\ \textit{Associativity:} & (x\#(y\#z)) = ((x\#y)\#z) \end{array}\right\} D_A$$

Let $l_1 = (x\square(y\#z))$, $r_1 = ((x\square y) \# (x\square z))$

$l_2 = ((x\#y)\square z)$, $r_2 = ((x\square z) \# (y\square z))$

then we define the following relations $\vdash_{\overrightarrow{D}}, \vdash_{\overleftarrow{D}}$ as:

$s \vdash_{\overrightarrow{D}} t$ iff $\exists\, \delta \in$ SUB: $s = s_1\delta(l_i)s_2$, $t = s_1\delta(r_i)s_2$

$s \vdash_{\overleftarrow{D}} t$ iff $\exists\, \delta \in$ SUB: $s = s_1\delta(r_i)s_2$, $t = s_1\delta(l_i)s_2$

where $s,t \in$ TERM, $i = 1,2$. Take $\overset{*}{\vdash}_{\overrightarrow{D}}, \overset{*}{\vdash}_{\overleftarrow{D}}$ as the reflexive and transitive closure of $\vdash_{\overrightarrow{D}}, \vdash_{\overleftarrow{D}}$ and let $\vdash_{D} = \vdash_{\overrightarrow{D}} \cup \vdash_{\overleftarrow{D}}$ and let $\overset{*}{\vdash}_{D}$ be the reflexive and transitive closure of $\vdash_{D}$. Because of the associativity of # we may drop unnecessary brackets. By doing so we do not have to consider the separate associativity axiom for # and hence may extend $\overset{*}{\vdash}_{D}$ to $\overset{*}{\vdash}_{D_A}$ without any additional rules. The notions

$s \overset{\overrightarrow{D}_A}{=} t$, $\sigma \overset{\overrightarrow{D}_A}{=} \tau$, $\sigma(s) \overset{\overrightarrow{D}_A}{=} \sigma(t)$, $\langle s;t\rangle_{\overrightarrow{D}_A}$, $\psi(\langle s;t\rangle_{\overrightarrow{D}_A})$

are defined by replacing the relation $\overset{*}{\vdash}_{D_A}$ by $\overset{*}{\vdash}_{\overrightarrow{D}_A}$.

For abbreviation let us give now some additional *notation:*

$s \lessdot t$ iff s is a substring of the string t

$|t|$ = length of the string t

$|t|_x$ = number of occurences of the symbol x *in* t

If t contains no #-symbol then t is called *"product term"*

$\coprod_{i=1}^{n} t_i := t_1 \# t_2 \# \dots \# t_n$, where $t_i \in$ TERM.

2.D.1. For $t \in$ TERM: $\underset{e}{\sim}[t] := \{s \in \text{TERM} : s \overset{D_A}{=} t$ and $\neg\exists\delta \in \text{SUB}$ s.t. $\delta(l_i) \lessdot s,\ i = 1,2\}$

Elements of $\underset{e}{\sim}[t]$ are called *"expanded terms"*.

Note: $\underset{e}{\sim}[t]$ is a finite set.

2.D.2. For $s,t \in$ TERM: A *#-adapter for* $\langle s;t\rangle D_A$, $\alpha_{\#} \in$ SUB, is a substitution s.t.

$\exists t' \in \underset{e}{\sim}[\alpha_{\#}(t)], \exists s' \in \underset{e}{\sim}[\alpha_{\#}(s)] : |s'|_{\#} = |t'|_{\#}$.

Now some observations:

2.L.0. Let for $s \in$ TERM: $\underset{e}{\rightarrow}[s] := \{\hat{s} \in \underset{e}{\sim}[s] : s \xrightarrow[\vec{D}_A]{*} \hat{s}\}$, then all elements of $\underset{e}{\rightarrow}[s]$ are mutually #-adapted terms.

2.L.1. For $t \in$ TERM:
all elements of $\underset{e}{\sim}[t]$ are mutually #-adapted terms.

2.C.1. If $s \overset{D_A}{=} t$ then: $\exists s' \in \underset{e}{\sim}[s]$ and $t' \in \underset{e}{\sim}[t]$ s. th. $|s'|_{\#} = |t'|_{\#}$.

2.D.3. Let for $a \in C$

i) $TERM_a \subset$ TERM $\cap$ $(V \cup \{a\} \cup \{\#, \square\} \cup \{(,)\})^*$, the set of terms containing only one constant symbol.

ii) $RPT_a := \{p \in TERM_a :$ p is a right associative productterm$\}$.

iii) $EXTERM_a := \bigcup_{k \geq 1} \{\prod_{i=1}^{k} p_i : p_i \in RPT_a$ *and* $|p_i| = |p_j|,\ 1 \leq i,j \leq k\}$, the set of all "sums" of right associative productterms of "equal length".

iv) $\sigma = \{x_1|r_1, \ldots, x_n|r_n\} \in$ SUB then we say σ is $\square$-*compatible* iff $\prod_{i=1}^{n} r_i \in EXTERM_a$, i.e. all productterms occuring on the righthandside of σ are of the same length.

v) $s,t \in EXTERM_a$ then we say s,t are $\square$-*compatible terms* iff $s \# t \in EXTERM_a$.

The intention is to show (the stronger result) that the D_A-unification problem (D_A-UP) with $\square$-compatible terms is undecidable, from which follows the undecidability of the whole D_A-UP.

2.L.2. For $t \in RPT_a$, $\square$-compatible $\sigma \in$ SUB: all $t_i, t_k \in \underset{e}{\sim}[\sigma(t)]$ are $\square$-compatible terms. Lemma 2.2 shows that "expansion" after a $\square$-compatible substitution does not change the $\square$-compatibility. The following lemma expresses the fact, that for $\square$-compatible terms s,t, if there is a D_A-unifier σ[i.e. $\sigma(s) \overset{D_A}{=} \sigma(t)$] then there is also a $\vec{D}_A$-unifier $\hat{\sigma}$[i.e. $\hat{\sigma}(s) \overset{\vec{D}_A}{=} \hat{\sigma}(t)$]. Therefore the D_A-UP with $\square$-compatible terms is reduced to $\vec{D}_A$-UP.

2.L.3. For $\square$-compatible terms s,t:

$$\Psi(<s;t>_{D_A}) \neq \emptyset \text{ iff } \Psi(<s;t>_{\vec{D}_A}) \neq \emptyset.$$

Before the main theorem can be formulated and proved we need some technical lemmas and a suitable way to express the relation of the decision problem of the D_A-UP to the decision problem of diophantine equations.

2.2. Decomposed Diophantine Equations

As in [3] a diophantine equation (DE) is a polynomial equation P = 0, where a polynomial is a function:

$$(*) \quad \sum_{0 \leq i_j \leq n_j} a_{i_1 i_2 \ldots i_k} x_1^{i_1} x_2^{i_2} \ldots x_k^{i_k}, \quad 1 \leq j \leq k$$

For our purpose we transform such a DE into a decomposed form, (called δ-*form*) which is more suitable for the D_A-UP and can be obtained by the following trivial transformation: *decompose* every subexpression in (*) of the form $\pm c x_1^{j_1} x_2^{j_2} \ldots x_r^{j_r}$ (called *"product expression"*) into a sum of c product expressions $\pm x_1^{j_1} \cdot \ldots \cdot x_r^{j_r}$ ('$\cdot$' denotes integer multiplication); *expand* each product expression $\pm x_1^{j_1} \cdot x_2^{j_2} \cdot \ldots \cdot x_r^{j_r}$ to $\pm 1 \cdot 1 \cdot \ldots \cdot 1 \cdot x_1^{j_1} \cdot \ldots \cdot x_r^{j_r}$ s.t. all product expressions obtained thus are of equal length; *transfer* all negative product expressions to the righthandside of the =-symbol.

Example: For $P = x^2y - 3y^2 + 2z = 0$ the δ-form is $x \cdot x \cdot y + 1 \cdot 1 \cdot z + 1 \cdot 1 \cdot z = 1 \cdot y \cdot y + 1 \cdot y \cdot y + 1 \cdot y \cdot y$. It is obvious that a DE in δ-form has the same set of solutions.

2.3. The Connection between DE-Problem and D_A-UP

Let us now define to every product term t the corresponding polynomial $\pi(t)$ by interpreting the elements of VAR(t) as integer variables and by replacing the constantsymbol 'a' in t by the integer constant '1'. In addition we interpret '$\square$' as integer multiplication.

Finally: given two terms $s,t \in EXTERM_a$, with

$$s = \prod_{i=1}^{n} s_i \quad \text{and} \quad t = \prod_{j=1}^{m} t_j \;,$$

define the corresponding polynomial equation.

$$PE(s,t) := \left(\sum_{i=1}^{n} \pi(s_i)\right) = \left(\sum_{j=1}^{m} \pi(t_j)\right)$$

(where Σ is the usual integer sum).

2.L.4. For $s,t \in EXTERM_a$ $\exists$ #-adapter for $<s;t>_{\vec{D}_A}$ iff $\exists$ diophantine solution for PE(s,t).

2.L.5. For $\square$-compatible terms $s,t \in EXTERM_a$: $\Psi(<s;t>_{\vec{D}_A}) \neq \emptyset$ iff $\exists$ #-adapter for $<s;t>_{\vec{D}_A}$. So far the intention is: lemma 2.4 establishes the relationship between DE and #-adapter. Lemma 2.5 then establishes the relationship between #-adapter and unifier. Lemma 2.5 holds only for the restricted class $EXTERM_a$, but note that DE in δ-form has a similar structure (as it contains at most the one integer constant '1').

Now we are ready to formulate the main theorem

Theorem 2.T.1. D_A-UP with □-compatible terms is decidable iff the DE-problem is.

Proof: (1) suppose D_A-UP with □-compatible terms is decidable.

Given a DE transform it in its δ-form P = Q. Now consider the D_A-UP $<\bar{P};\bar{Q}>_{D_A}$, where $\bar{P},\bar{Q}$ is obtained by inserting brackets into the product expressions of P,Q in a *uniform* way, say right associative and replace integer multiplication '·' by '□', replace '1' by the constant 'a': e.g. 1·1·x·x·y becomes (a□(a□(x□(x□y)))). Also replace integer plus by ⋕. Now $PE(\bar{P},\bar{Q})$ is of the form P = Q and since $\bar{P},\bar{Q}$ are □-compatible lemma 2.5 is applicable. Using lemma 2.3 we only have to consider $\vec{D}_A$.

With Lemma 2.4 and 2.5 it is possible to show: $\Psi(<\bar{P};\bar{Q}>_{\vec{D}_A}) \neq \emptyset$ iff there exist a diophantine solution for $PE(\bar{P},\bar{Q})$ resp. for P = Q.

(2) suppose the DE-problem is decidable.

To determine whether for □-compatible terms $s,t \in EXTERM_a$ $\Psi(<s;t>_{D_A}) \neq \emptyset$ by lemma 2.3 we have to consider if $\Psi(<s;t>_{\vec{D}_A}) \neq \emptyset$.

Denote by L(DE) the solution space of a DE. Then by Lemma 2.4 and Lemma 2.5:

$L(PE(s,t)) \neq \emptyset$ iff $\Psi(<s;t>_{\vec{D}_A}) \neq \emptyset$. ■

Suppose now the D_A-unification problem was decidable, then the restricted problem, D_A-UP, for □-compatible terms would be decidable. By the above theorem would follow Hilbert's tenth problem [3] was decidable, hence D_A-UP is undecidable (of the same degree as the DE-problem).

2.4. Conclusion

As a consequence of theorem 2.1 further unification problems are unsolvable, viz.: D+A-UP, that is the unification problem for which '□' is associative too. In this case we even have the simplification, that we may drop the rightassociative requirement. Also the D+A+C-UP is undecidable, since integer '+' and '·' are commutative. In fact any combination of D_A with the Peanoaxioms of arithmetic remains undecidable considered as a unification problem: Hilbert's tenth problem is the "unification problem" for integer arithmetic. As a byproduct we have shown that we may drop all these additional axioms except for D_A and still remain undecidable, i.e. that the undecidability of Hilbert's tenth problem did <u>not</u> depend on the notion of integers, but only on the "power" of D and A.

3. D-Unification

In the previous chapter the undecidability of the unification problem under distributivity and associativity has been shown. As we know that the unification problem for associative terms is decidable (a consequence of Makanin [16]) the question whether or not the unification problem for distributive terms is decidable is outstanding: If the D-unification problem, D-UP, (for the axioms see chapter 0) was decidable the D_A-UP would be the first undecidable UP of our chart resulting from two axioms, whose (single) UPs are decidable. If the D-UP was undecidable, the distributivity axiom would be the first *single* axiom with an undecidable UP. As yet we do not have a proof either way.

So far we do have a complete enumeration algorithm for D-unifiers, the minimality of which is under investigation.

The minimality problem is not trivial as the set of D-mgus is infinite in general as the following example demonstrates:

Let $<f(x,b);f(a,y)>_D$ be a D-UP with $x,y \in \mathcal{V}$, $a,b \in \mathcal{C}$. Then

Φ: $\{x|a, y|b\}$
$\{x|g(a,a), y|g(b,b)\}$
$\{x|g(g(a,a),a), y|g(g(b,b),b)\}$
$\{x|g(a,g(a,a)), y|g(b,g(b,b))\}$
.
.
.

is a subset of $\Sigma(<f(x,b);f(a,y)>_D)$. It is easy to see that all elements of Φ are unifiers for $<f(x,b);f(a,y)>_D$. It is also possible to show that the elements of Φ are independent and that there is no upper bound σ with $\varphi \leq \sigma$ $\forall \varphi \in \Phi$.

D-unification appears to be among the most difficult T-unification problems as investigated so far.

The above example also demonstrates that addition of associativity and/or commutativity does not change the cardinality of Σ.

4. *A+C-Unification*

4.1. *Introduction*

In this section we investigate the problem of unifying two terms in TERM for which the associativity and commutativity axioms hold. The observation that this A+C-unification problem reduces to the solution of certain linear diophantine equations is the basis for a complete and minimal unification algorithm.
It is also shown that completeness and minimality is closely related to the notion of a basis for the linear solutionspace of these equations.
The restriction to terms in TERM is justified as the general case of FOPC terms follows by a simple extension of the presented algorithm in the following sense: if there is a unification algorithm which solves the unification problem for this limited class of terms then the extended algorithm solves the unification problem for the full class of predicative terms. This extension is unproblematic as long as all additional functions occuring are to be unified with Robinson's unification algorithm [20]. The problem is open otherwise.
A more natural representation may drop the f's in TERM, e.g.: f(f(x,y),f(a,z)) is represented as the 'Abelian string': xyaz. That is, an A+C-unification problem is the pair $<t_1;t_2>_{AC}$ where t_1 and t_2 are Abelian strings over $(\mathbb{V} \cup \mathbb{C})^*$. Abelian strings (or multisets) are a particular data structure explicitely or implicitely provided for in many programming languages. In some *AI*-programming languages, such as PLANNER [7] these data structures, sometimes called bags, play an essential rôle in the matching process and this problem was first investigated in [26]. The approach presented in this paper (the original work of which was done independently to [26]) has some advantages over [26]:

(i) A proper reduction to the basis for the solution space of a linear equation system and the close relationship between a most general unifier and such a basis.

(ii) The avoidance of the "variable abstraction" of [26] and hence a more efficient and direct algorithm

(iii) Less many mgu's (since our algorithm is minimal).

The difference may best be seen by comparing our example in § 4.4 with the same example as reported in [26] page 74.
An additional motivation for the presentation of our work is that the solution of the ACI-unification problem in § 5. directly follows from our solution to the AC-unification problem.

4.2. *The Method*

For ease of presentation we assume the existence of an identity element "e" with: ex = xe = x.
4.D.1. $<t_1;t_2>_{AC}$ is *normalized* iff the symbols in t_1 and t_2 are disjoint.
4.L.1. Let $<t_1;t_2>_{AC}$ be the normalized problem of $<\hat{t}_1;\hat{t}_2>_{AC}$. Then $\Sigma(<t_1;t_2>_{AC}) = \Sigma(<\hat{t}_1;\hat{t}_2>_{AC})$ up to renaming.
In order to demonstrate the method consider the following example
4.E.1: $<x^3y^2a^2b;zcd>$ where x,y,z are variables and a,b,c,d are constants.
Now we ask what general form a unifying substitution δ might have: firstly it will substitute symbols only for the variables occuring in the two strings, in this case for x,y and z respectively:

$$\delta = \{(x|t_1)(y|t_2)(z|t_3)\}$$

where the t_i may be empty or some strings otherwise. Secondly δ will only substitute constants already occuring in the unification problem, since it is easy to see that otherwise δ would not be most general. And finally δ may substitute variables already occuring in the two strings and/or it may substitute 'new' variables. Using a simple trick and making use of the $\underset{\mathbb{V}}{=}$-construct it is however always possible to replace a unifier δ which substitutes variables already occuring in the two strings by a unifier $\hat{\delta}$ which only substitutes new variables such that $\delta \underset{\mathbb{V}}{=} \hat{\delta} \bullet \lambda$. Hence a unifier for 4.E.1. will have the general form:

4.E.2.
$$\delta = \{(x|v_1^{n_1}v_2^{n_2}a^{n_3}b^{n_4}c^{n_5}d^{n_6})$$
$$(y|v_1^{n_7}v_2^{n_8}a^{n_{10}}b^{n_{11}}c^{n_{12}}d^{n_{13}})$$
$$(z|v_1^{n_{14}}v_2^{n_{15}}a^{n_{16}}b^{n_{17}}c^{n_{18}}d^{n_{19}})\}$$

with v_i 'new' variables and $n_i \in \{0,1,2,..\}$. For consistency we adopt the convention that to the power of zero means that the symbol does not occur at all, i.e. $a^0=$,$a^1=a$,.... The unification problem is now reduced to the problem to find appropriate values

for the $n_1, \ldots, n_{19}$.
In order to determine the values for the n_i's we observe the following: let δ be a unifier for $\langle t_1;t_2\rangle_{AC}$. Then for each symbol, the number of occurences of that symbol in $\delta(t_1)$ must be the same as the number of occurences in $\delta(t_2)$. For example $\delta = \{(x|v_1c)(y|v_2)(z|v_1^3v_2^2c^2da^2b)\}$ is a unifier for 4.E.1. as is easily checked by a symbolcount.
This fact completely determines the n_i, since for each symbol we can set up a corresponding equation: Consider the constant "a" in 4.E.2. above: δ will substitute n_3"a's" for x, n_{10}"a's" for y and n_{16}"a's" for z. Now since x occurs three times in the first string, y occurs twice in the first string and z occurs once in the second string we obtain the following diophantine equation for the "a's":

$$3\cdot n_3 + 2\cdot n_{10} + 2 = n_{16}$$

and obviously every triple of nonnegative integers satisfying this equation determines a unifying substitution in a. The set of linear diophantine equations for all symbols in δ will then determine every possible unifying substitution.
Hence the set of equations for 4.E.1. (obtained with 4.E.2) is:

4.E.3.	$3\cdot n_1 + 2\cdot n_7 - n_{14} = 0$	for v_1
	$3\cdot n_2 + 2\cdot n_8 - n_{15} = 0$	for v_2
	$3\cdot n_3 + 2\cdot n_{10} - n_{16} = -2$	for a
	$3\cdot n_4 + 2\cdot n_{11} - n_{17} = -1$	for b
	$3\cdot n_5 + 2\cdot n_{12} - n_{18} = 1$	for c
	$3\cdot n_6 + 2\cdot n_{13} - n_{19} = 1$	for d .

In [15] we quote some results on solving homogeneous equations over $\mathbb{N}$ and also show how to solve inhomogeneous diophantine equations over $\mathbb{N}$.

4.3. The Algorithm

step 0. Given an AC-unification problem $\langle s_1';s_2'\rangle_{AC}$ where s_1' and s_2' are Abelian strings compute it's normalized form $\langle s_1;s_2\rangle_{AC}$.
step 1. Assuming vectors to be ordered by $\sqsubseteq$ in some way. Let E be an inhomogeneous equation system with integer coefficients. The (unique) *special solution list* SS = $\langle \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\rangle$ for E is defined as follows:
(i) $\bar{x}_1$ is the smallest solution vector for E
(ii) $\bar{x}_i \sqsubseteq \bar{x}_{i+k}$
(iii) Every solution $\bar{x}_m \sqsubseteq \bar{x}_n$ for E (with $\bar{x}_m$ p-linear independent of the $\bar{x}_k \in SS$) is in SS
(iv) $\bar{x}_n$ is the smallest solution vector such that for every solution $\bar{x}_k$, with $\bar{x}_n \sqsubseteq \bar{x}_k$, there exists $\bar{x}_i \in SS$ s. th. $\bar{x}_k - \bar{x}_i$ is not negative.
Let $c_1, c_2, \ldots, c_p$ be all the constants occuring in s_1 and s_2. For each c_i set up the linear equation as in 4.E.3. and compute SS_i for each equation. Form the cross product: $\mathbb{C}$: $SS_1 \times SS_2 \times \ldots \times SS_p$.
step 2. Solve the (single) homogeneous equation in 4.E.3.

$$\text{Let } \bar{x}_1 = (x_{11}, x_{12}, \ldots, x_{1m})$$
$$\vdots$$
$$\bar{x}_n = (x_{n1}, x_{n2}, \ldots, x_{nm})$$

be the positive independent base vectors spanning the solution space. Here m is the number of distinct variables in $\langle s_1;s_2\rangle_{AC}$.
step 3. The set $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ where k is the number of elements in $\mathbb{C}$, is computed as follows:
(3.1) The 'new' variables
Each σ_i substitutes n 'new' variables, where n is the number of independent vectors as computed in step 2. Let $u_1, u_2, \ldots, u_m$ be the 'old' variables in $\langle s_1;s_2\rangle$ and $v_1, v_2, \ldots, v_n$ be the new variables. Then each σ_i is of the form:

$$\begin{aligned}\sigma_i = \{&(u_1|v_1^{x_{11}}v_2^{x_{22}} \ldots v_n^{x_{n1}}\ \text{const1})\\ &\quad\vdots\\ &(u_m|v_1^{x_{1m}}v_2^{x_{2m}} \ldots v_n^{x_{nm}}\ \text{constm})\}\end{aligned}$$

where constk is some combination of constants of $\langle s_1;s_2\rangle$, which are determined as follows:
(3.2) The constants
Let $(\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_p)$ be some member of $\mathbb{C}$, with $\bar{y}_i = (y_{i1}, y_{i2}, \ldots, y_{im})$ for $1 \le i \le p$. Then:

$$\begin{aligned}\text{const1} &= c_1^{y_{11}}c_2^{y_{21}}c_3^{y_{31}} \ldots c_p^{y_{p1}}\\ &\vdots\\ \text{constm} &= c_1^{y_{1m}}c_2^{y_{2m}}c_3^{y_{3m}} \ldots c_p^{y_{pm}}\end{aligned}$$

4.4. An Example

Let $\langle x^2yac;b^2zc\rangle$ be an AC-unification problem.
step 0. $\langle x^2ya;b^2z\rangle$ is its normalized form.
step 1. The σ_i are of the form:

$$\begin{aligned}\sigma = \{&(x|v_1^{n_1}v_2^{n_2}a^{n_3}b^{n_4})\\ &(y|v_1^{n_5}v_2^{n_6}a^{n_7}b^{n_8})\\ &(z|v_1^{n_9}v_2^{n_{10}}a^{n_{11}}b^{n_{12}})\}.\end{aligned}$$

The constant equations are

$2n_3 + n_7 - n_{11} = -1$ for a

$2n_4 + n_8 - n_{12} = 2$ for b

$SS_a = \langle(0,0,1)\rangle$ $SS_b = \langle(1,0,0),(0,2,0)\rangle$.

step 2. The solution space for the homogeneous equation $2n_3 + n_7 - n_{11} = 0$ is spanned by

$\bar{x}_1 = (0,1,1)$, $\bar{x}_2 = (1,0,2)$.

step 3. The two mgus are:

$\sigma_1 = \{(x|v_2b)$ $\sigma_2 = \{(x|v_2)$

$(y|v_1)$ $(y|v_1b^2)$

$(z|v_1v_2^2a)\}$ $(z|v_1v_2^2a)\}$.

The same example is used in [26] to demonstrate that method and it shows the improvement achieved by our method: the ease and simplicity by which the unifiers are computed, the direct relationship between mgu and base vectors and finally we compute less many mgus as our set Σ of generated mgus is minimal.

In [15] correctness, completeness and minimality of Σ, the set of unifiers returned by the previous algorithm, is shown. We also show how to compute SS and prove that it is our intended set. It is also shown how to obtain the set of most general unifiers if the underlying theory does not contain an identity element.

5. *A+C+I-Unification*

We are interested in the unification of terms in TERM under the assumption that the associativity, commutativity and idempotence axioms hold for the distinguished function symbol 'f'. As before we assume the existence of an identity element "e" with: $f(x,e) = f(e,x) = x$.

A possible representation of such terms is the stringrepresentation of the previous section with the additional proviso that we do not have multiple occurences of symbols. Intuitively such strings resemble sets, which is part of the motivation for this work: i.e. the matching problem for the datastructure *sets*, as encountered in AI-languages such as [7]. The other obvious motivation springs from automatic theorem proving, where such axioms occur. The solution of an ACI-unification problem proceeds now in an exactly parallel manner to the solution of an AC-unification problem with the proviso that the arithmetic equations are replaced by Boolean equations.

5.1. *The Algorithm*

Consider the following two strings to be unified: <xy;aab>. By the idempotent law, the two occurences of "a" reduce, i.e. the unification problem becomes:

5.E.1. <xy;ab>. As for the AC-unification problem, the two strings in 5.1.E.1. are unified iff for each symbol the number of occurences in one string equals the number of occurences in the other string. But, because of the additional idempotence law, each symbol either does not occur at all or occures at most once. Hence the corresponding equations for each symbol do not range over nonnegative integers as in the previous section but only over: {0,1}.

Furthermore we have to change the operation of addition, which was ordinary integer addition in the AC-unification problem: if in the above example "a" is substituted for "x" *and* for "y", the total occurence of a's in the lefthand string is still one, because of the idempotence. For that reason we define the following operation $\oplus$:

$1 \oplus 0 = 0 \oplus 1 = 1$; $0 \oplus 0 = 0$; $1 \oplus 1 = 1$; i.e. in fact we are solving Boolean equations.

For 5.E.1. a potential unifier has the form:

$\sigma = \{(x|a^{n_1}b^{n_2}),(y|a^{n_3}b^{n_4})\}$ with the following equations for the n_i:

$n_1 \oplus n_3 = 1$ for "a"

$n_2 \oplus n_4 = 1$ for "b".

The solutions to these equations determine nine ACI-mgus for 5.E.1.

$\sigma_1=\{(x|e),(y|ab)\}$ $\sigma_5=\{(x|b),(y|a)\}$

$\sigma_2=\{(x|ab),(y|e)\}$ $\sigma_6=\{(x|ab),(y|ab)\}$

$\sigma_3=\{(x|ab),(y|b)\}$ $\sigma_7=\{(x|b),(y|ab)\}$

$\sigma_4=\{(x|a),(y|b)\}$ $\sigma_8=\{(x|a),(y|ab)\}$

$\sigma_9=\{(x|ab),(y|a)\}$

6. *C-Unification*

6.1 *The Obvious Solution OS*

We are interested in the unification of terms in TERM where the commutativity axiom holds for "f".

6.D.1. For $t \in$ TERM, and $\delta \in$ SUB:

$T(t) := \{\hat{t} \mid \text{for all } \hat{t} \overset{C}{=} t\}$.

Now the *obvious solution* (OS) to a C-unification problem $\langle t_1;t_2\rangle_C$ is the following procedure: Apply Robinson's unification algorithm [20] to each element $(\hat{t}_1,\hat{t}_2) \in T(t_1) \times T(t_2)$. The resulting set of

unifiers is $\Psi\langle t_1;t_2\rangle_C$, which is always finite.
There are two main problems with OS:
<u>(1.)</u> The *completeness* of this algorithm is <u>not</u> obvious, as in the case of an associative function it is known that the obvious method is <u>not</u> complete [18],[23]. *Similarily there is no reason to assume that the obvious solution for the commutativity axiom is complete,* and in view of [18],[23] intuition seems to suggest the contrary.
<u>(2.)</u> The second problem concerns the proliferation of identical and mutually dependent unifiers *(minimality):*
<u>(2.1.)</u> For example $\langle f(x,a),f(a,y)\rangle_C$ would produce the unifiers: $\delta_1 = \{(x|y)\}$, $\delta_2 = \{(x|a),(y|a)\}$ the second of which is an instance of the former, thus violating the minimality condition for Σ.
<u>(2.2.)</u> For the final remark consider the example: $\langle f(f(a,y),f(b,z));f(f(x,a),f(u,c))\rangle_C$. There are $2^3 = 8$ swaps each, leading to $8 \times 8 = 64$ unification problems, of which only 8 are different, the other 56 are in fact some permutation of those eight problems.

6.2. Labeltransformations, Swaps and Maximal Swaps

To formalize the commuting of arguments, we first note, that the elements of TERM may denote binary trees and the commutativity axiom states the equivalence of such trees under the usual tree ordering.
We shall identify a node by a binary label and assume this node to be labelled with some $s \in \{f\} \cup C \cup V$.
Then, taking nodes (binary labels) as binary numbers we define:
6.D.2. A *labeltransformation* is a mapping
NODES × NODES → NODES. Let
$m = \sum_{i=0}^{l} a_i \cdot 2^{l-i}$ $k = \sum_{i=0}^{n} b_i \cdot 2^{n-i}$ $a_i, b_i \in \{0,1\}$.
Then *m transformed by k*, $k \oplus m$, is:

```
$k \oplus m$ := if $k \geq m$ then $m$
elseif $b_0 = a_0$ and $b_1 = a_1$ and ... and $b_n = a_n$
then $\sum_{i=0}^{n} a_i \cdot 2^{l-i} + \sum_{i=n+2}^{l} a_i \cdot 2^{l-i} + (1-a_{n+1}) \cdot 2^{l-n-1}$
else $m$ fi.
```

Note: $\oplus$ is neither associative nor commutative.
The labeltransformation checks whether the first n digits in m and k are the same: if false the value is m and if true then the $(n+1)$th digit in m is changed to 1 if it was 0 and changed to 0 if if was 1.
The intention is to transform the right son of k (and its descendants) into the left son and vice versa. A labeltransformation is extended to sets of nodes by $k \oplus \{m_1,m_2,\dots,m_q\} := \{k \oplus m_1, k \oplus m_2,\dots,k \oplus m_q\}$.
6.D.3. A *single swap* ρ is a mapping ρ: TERM → TERM.
Let $t \in$ TERM, $\rho = \langle k\rangle$; then $\rho(t)$ is the term obtained from t by replacing every node $n \in$ NODE(t) by $k \oplus n$.
The composition of two single swaps is defined as the functional composition and a *swap* is the composition of finitely many single swaps.
6.D.4. A *swap* $\rho = \langle k_1,k_2,\dots,k_p\rangle$ is in *normal form* iff $k_1 < k_2 < k_3 < \dots < k_p$.
The following lemma is technical and shows how to "commute" two single swaps. This result is then used to prove the normal form lemma 6.L.3.
6.L.1. $\langle k_1\rangle \bullet \langle k_2\rangle = \langle k_1 \oplus k_2\rangle \bullet \langle k_2 \oplus k_1\rangle$.
6.D.5. Let $\rho = \langle k_1,k_2,\dots,k_p\rangle$, $\hat{\rho} = \langle m_1,m_2,\dots,m_q\rangle$ be two swaps,
(i) $\hat{\rho} \sqsubseteq \rho$ iff $\hat{\rho} = \rho$ and $q \leq p$
(ii) ρ is *maximal* iff $\forall\ \hat{\rho}$ with $\hat{\rho} = \rho : \hat{\rho} \sqsubseteq \rho$.
6.L.2. To every swap $\hat{\rho}$ exists a unique maximal swap ρ in normal form s. th. $\hat{\rho} = \rho$.
The following lemma finally shows the close correspondence between the notion of a maximal swap and a most general unifier:
6.L.3. Let $t_1 \overset{C}{=} t_2$. Then exists a unique maximal swap ρ in normal form such that $\rho(t_1) \overset{\emptyset}{=} t_2$.

6.3. Completeness and Correctness of OS

The correctness of every unifier as generated by OS follows trivially from the correctness of R-unification. To show the completeness of OS we use the following lemma.
6.L.4. Let $\langle t_1;t_2\rangle_C$ be a C-unification problem with C-unifier δ: $\delta(t_1) \overset{C}{=} \delta(t_2)$.
Let ρ_1 and ρ_2 be swaps compatible with δ such that $\rho_1(\delta(t_1)) = \rho_2(\delta(t_2))$. Let $\hat{t}_1 := \rho_1(t_1)$ and $\hat{t}_2 := \rho_2(t_2)$. Then exists $\hat{\delta} \overset{C}{=} \delta$ such that $\hat{\delta}$ is R-unifier for $\langle \hat{t}_1;\hat{t}_2\rangle_\emptyset$: $\hat{\delta}(t_1) \overset{\emptyset}{=} \hat{\delta}(t_2)$.
6.Th.1. (Completeness) Let $t_1;t_2 \in$ TERM and $\Psi\langle t_1;t_2\rangle_C$ be the set of unifiers generated by OS.
Then $\forall\ \delta$ with $\delta(t_1) \overset{C}{=} \delta(t_2)$, $\exists\ \lambda$ and $\sigma \in \Psi$ such that $\delta \overset{C}{=} \lambda \bullet \sigma$.

6.4. Two Necessary Minimality Conditions

The following two lemmas state, that if the arguments in a unification problem are equal under commutativity, then a swap will result in an equal R-unifier for both or in mutually dependend R-unifiers:

6.L.5. Let

(i) $<f(t_1,t_2);\ f(s_1,s_2)>_\emptyset$ and

(ii)$<f(t_1,t_2);\ f(s_2,s_1)>_\emptyset$

be two unification problems; $s_1,s_2,t_1,t_2 \in$ TERM. If $t_1 \overset{C}{=} t_2$ then $\delta_i \overset{C}{=} \delta_{ii}$, where δ_i and δ_{ii} are R-mgus for (i) and (ii) respectively.

6.L.6. Let

(i) $<f(t_1,t_2);\ f(s_1,s_2)>_\emptyset$ and

(ii)$<f(t_1,t_2);\ f(s_2,s_1)>_\emptyset$; $t_1,t_2,s_1,s_2 \in$ TERM. If $s_1 \overset{C}{=} t_1$ then 1) if R-mgus exist for (i) and (ii) then $\exists\lambda$ s.th. $\delta_{ii} = \delta_i \bullet \lambda$, 2) if mgu δ_{ii} exists then mgu δ_i exists. Unfortunately the above conditions are only necessary but not sufficient; i.e. if $\delta_{ii} = \lambda \bullet \delta_i$ one can not necessarily assume, that some of the arguments are equal under commutativity as demonstrated in [24].

6.5. A Modification to the Obvious Solution, MOS

6.D.6. A C-unification problem $<t_1;t_2>_C$ is *ordered* iff $\|t_1\|_f \geq \|t_2\|_f$. Using this definition OS is modified to MOS: Let $<t_1;t_2>_C$ be an ordered C-unification problem. Apply a modification of Robinson's unification algorithm to each element of $\{t_1\} \times T(t_2)$.

The modification of Robinson's unification algorithm, CR-UNIFY, is obtained from Robinson's unification algorithm by a redefinition of "disagreement set" to "C-disagreement set".

The resulting unifier is called a CR-mgu and is unique under renaming and $\overset{C}{=}$.

6.Th.2. MOS is complete iff OS is. In [24] we present an algorithm close to MOS which takes the above (and some other) minimality conditions into account.

7. References

[1] Bennett, Easton, Guard, Settle. CRT-aided semi-automated mathematics. Techn. Rep. AFCRL 67-0167,1967, Applied Logic Corp., Princeton.

[2] S. Cook. Algebraic techniques and the mechanization of number theory. Techn. Rep. RM-4319-PR, 1965, Rand Corp., Santa Monica, Cal.

[3] M. Davis. Hilbert's tenth problem is unsolvable. Amer. Math. Monthly, vol 80, 1973.

[4] P.J. Faber, R.E. Griswald, I.P. Polonsky.SNOBOL as String Manipulation Language.JACM,vol 11, no 2, 1966.

[5] J. Fischer, S. Patterson. String Matching and other Products, MIT, Project MAC,Rep. 41,1974.

[6] W.E. Gould. A matching procedure for ω-order logic. Scientific rep. no 4, AFCRL-666-781,1966.

[7] C. Hewitt. Description and theoretical analysis of PLANNER. Ph.D.-Thesis,MIT,1972, Art. Int. Lab., Cambridge.

[8] J. Hmelevskij. The solution of certain systems of word equations. Dokt.Akad. Nauk SSSR (1964), (1966),(1967),(Soviet Math. Dokl.).

[9] G. Huet. A unification algorithm for typed λ-calculus, Theoretical Comp. Sci. 1.1., 1975.

[10] G. Huet. Confluent Reductions."Abstract Properties and Applications to Term Rewriting Systems", Pap.Rech. no 250,IRIA Lab., Rocquencourt, France, 1977.

[11] D.E. Knuth, P.B. Bendix. Simple Word Problems in Universal Algebras, in "Computational Problems in Abstract Algebra", J. Leech (ed), Pergamon Press, Oxford 1970.

[12] Knuth, Morris, Pratt. Fast Pattern Matching in Strings. Stan-CS-74-440, Stanford Univ., Comp. Science Dept., 1974.

[13] D. Lankford. Complete sets of reductions. Univ. of Texas, Autom.Theor. Proving Project,Austin, Tex., Techn. Rep. ATP-35,ATP-37,ATP-39, 1977.

[14] M. Livesey, J. Siekmann. Termination and Decidability Results for String Unification. Essex Univ., Computing Centre, Memo CSM-12, 1975.

[15] M. Livesey, J. Siekmann. Unification of Bags and Sets. Int. Ber. 3/76, Inst. f. Inf. I, Univ. Karlsruhe, 1976.

[16] G.S. Makanin. The Problem of Solvability of Equations in a Free Semigroup, Soviet Akad. Nauk SSSR, Tom 233, no 2, 1977.

[17] G.E. Peterson, M.F. Stickel. Complete Sets of Reductions for Equational Theories with Complete Unification Algorithms. Dept. Comp. Sci., Univ. of Arizona, Tucson, Techn. Rep., 1977.

[18] G. Plotkin. Building in equational theories. Mach. Intelligence, vol 7, 1972.

[19] P. Raulefs, J. Siekmann. Unification of Idempotent Functions. Univ. Karlsr.,Inst.f.Inf.I,1978.

[20] J.A. Robinson. A machine oriented logic based on the resolution principle. JACM:12, 1965.

[21] J.A. Robinson. A review on automatic theorem proving. Symp.Appl.Math.,vol 19,1-18, 1967.

[22] G. Robinson, L. Wos. Maximal models and refutation completeness: Semidecision procedures in automatic theorem proving. In Boone et al. (eds.), "Word problems", North Holland, 1973.

[23] J. Siekmann. String unification. Essex Univ., Memo CSM-7.

[24] J. Siekmann. Unification of commutative terms. Int.Ber. 2/76, Univ. Karlsr.,Inst. f. Inf. I.

[25] J.R. Slagle. ATP with built in theories including equality, partial ordering, and sets. JACM:19, 1972.

[26] M. Stickel. A complete unification algorithm for associative-commutative functions. Proc. 4th IJCAI, Tblisi, USSR, 1975.

[27] P. Szabó. The undecidability of the D_A-unification problem. Univ. Karlsr., Inst.f.Inf.I,1978.

[28] P. Szabó, E. Unvericht. D-unification has infinitely many mgus. Univ. Karlsr.,Inst.f.Inf.I, (forthcoming).

[29] E. Vogel. Unifikationsalgorithmen für Morphismen.Dipl.Arb.,Univ. Karlsr.,Inst.f. Inf. I.'78.

Index of Authors