WORKSHOP ON AUTOMATIC DEDUCTION

MIT

Cambridge, Mass.

Aug. 17 - 19, 1977

COLLECTED ABSTRACTS

CONTENTS

P. B. Andrews and E. L. Cohen

Theorem Proving in Type Theory.

F. Konrad

Weak Second Order Logic as Data Base Language.

P. L. Suzman

Some Issues in the Design of a Representation Language.

E. L. Lusk and R. A. Overbeek

Experiments with Resolution - Based Theorem Proving Algorithms.

R. Fikes and G. Hendrix

A Network - Based Knowledge Representation and its Natural Deduction System.

S. C. Shapiro

Compiling Deduction Rules from a Semantic Network into a Set of Procedures.

V. Marinov

Proper Role for Resolution Theorem Provers.

D. H. Fishman

A Proclarative Approach to Problem Solving.

D. McDermott

Deduction in the Pejorative Sense.

E. Sandewall

Predicate Calculus as a Blueprint for Programs.

K. L. Clark and S. Tarnlund

A First Order Theory of Data and Programs.

R. S. Boyer and J. S. Moore

Using Lemmas in an Automatic Theorem Prover for Recursive Function Theory.

Z. Manna and R. Waldinger

The Automatic Synthesis of Recursive Programs.

R. E. Shostak

An Algorithm for Reasoning about Equality.

D. S. Lankford and A. M. Ballantyne

Decision Procedures for Simple Equational Theories with Permutative Equations: Complete Sets of Permutative Reductions.

Using Rewriting Rules for Connection Graphs to Prove Theorems

by

C. L. Chang*

and

J. R. Slagle**

* C. L. Chang is with the Department of Computer Science, IBM Research Laboratory, San Jose, Calif. 95193

** J. R. Slagle is with Naval Research Laboratory, Wahington, D. C. 20375

Abstract

Essentially, a connection graph is merely a data structure for a set of clauses indicating possible refutations. The graph itself is not an inference system. To use the graph, one has to introduce operations on the graph. In this note, we shall describe a method to obtain rewriting rules from the graph, and then to show that these rewriting rules can be used to generate a refutation plan that may correspond to a large number of linear resolution refutations. The method is efficient because many redundant resolution steps can be aoided.

NOTES

HEREDITARY LOCK -- RESOLUTION:

A RESOLUTION REFINEMENT COMBINING LOCK RESOLUTION AND THE MODEL STRATEGY

David M. Sandford

ABSTRACT

Hereditary-Lock Resolution (HI-Resolution, or HLR) is a sound and complete refinement of unrestricted resolution. HL-Resolution combines the Lock Resolution strategy (LR) of R. Boyer with The Model Strategy (TMS) of D. Luckham to achieve a model based strategy which is almost singly connected. The main idea is that HLR generalizes the notion of a clause by appending to the usual literals (referred to as standard literals) of a clause an additional set of literals which are called the FSL (False Substitution List) literals of the clause. Through this representational augmentation HLR is able not only to combine LR and TMS as a complete combination strategy, but is also able to remove some inherent inefficiency of TMS. This inefficiency exists in most previously known model based resolution strategies, and its removal in HLR depends crucially on the FSL concept.

HLR requires that there be a Herbrand interpretation, H, according to which truth evaluations can be made. This model information is specified in HLR as a function, M, whose domain is

sets of literals in the language of the clause set to which HLR is being applied. The function M has the value "feasible" if its argument is a set of literals such that there exists a substitution that simultaneously converts each literal to a false ground literal in H. Otherwise M has the value "infeasible". A clause, C, in HLR must meet the requirement that the FSL of C is feasible with respect to H (i.e. M applied to the FSL of C has the value "feasible"). Such a requirement is equivalent to saying that a clause represents only those ground instances of its standard literals for which all of its FSL literals are false, and that clauses which represent no ground instances can be deleted from the search. When forming a resolvent in HLR, a FSL set of literals for the resolvent is constructed which consists of all of the FSL literals from both parents, and possibly some of the standard literals from both parents (according to the rules of HLR). The unifier used in the resolution step is also applied to the FSL of the resolvent. Then the resolvent is checked for feasibility, and if not feasible, it need not be kept in the search space.

In addition to presenting the proof of soundness and completeness of HLR, we will also discuss some notions that have been developed in a preliminary form concerning the specification modes of models for use in theorem proving. These notions assert that the appropriate level of description of a model involves three distinct components:

1. A set of primitive facts.
2. A translation procedure mapping statements from the language of the clauses into the language of the model.

3. A processing algorithm which makes truth decisions about the translated clauses in light of the primitive facts. It is assumed that this processing algorithm can itself be expressed as a set of logical axioms.

Each of these three components embodies information which defines the actual model which is constructed. Fundamental to this notion of a model is that the primitive facts and the processing algorithm together constitute a logical system in which statements (or sets of statements) in the language of the model may be tested for consistency and theoremhood. Such a model, as a theorem proving system itself, represents the set of Herbrand interpretations in which the primitive facts and the axiomatization of the processing algorithm are themselves satisfied. The translation procedure is the connecting link between the theorem proving environment in the language of the clauses and the theorem proving environment in the language of the model.

Resolution strategies such as TMS and HLR are predicated on the use of a single Herbrand interpretation as the model. However a Herbrand interpretation with a non-trivial structure is a difficult object both to generate and to utilize in clause evaluations. We will show that the above stated view concerning model specification is sufficiently close to the concept of a Herbrand interpretation so as to be compatable with the soundness and completeness of a strategy such as HLR, but yet seems to offer advantages with respect to ease of specification and adaptability.

NOTES

PATN:

Planning and Debugging in a Predicate Calculus Problem Solver

Ira P. Goldstein and Mark L. Miller

Massachusetts Institute of Technology
Artificial Intelligence Laboratory
March 1977

PATN is a procedural problem solving system which we have designed and are currently implementing. PATN accepts predicate calculus problem descriptions as input, and transforms these into procedures to accomplish the specified goals. The planning and debugging knowledge embodied by PATN bridges the gap between predicate calculus as a non-deterministic programming language [Kowalski 1973] and the sequential procedures of a deterministic programming language, thereby clarifying this distinction.

Figure 1 shows an hierarchical taxonomy of common planning techniques. According to this taxonomy, planning begins with a choice between three methods -- identification, decomposition and reformulation. By *identification*, we mean recognizing the problem as one which has previously been solved, or noticing that the current problem is a direct special case of one which has previously been solved. By *decomposition*, we mean dividing the current problem into sub-problems which are (hopefully) easier to solve. In PATN, decomposition techniques are organized around the standard logical operators; thus, PATN's decomposition knowledge amounts to a compiler for predicate calculus programs. Limitation to the initial problem representation, however, would severely detract from the system's effectiveness. Hence, PATN introduces a third category, *reformulation* techniques, for transforming a problem description into an alternative form whose solution is equivalent to, or a least a stepping stone towards, the solution of the original

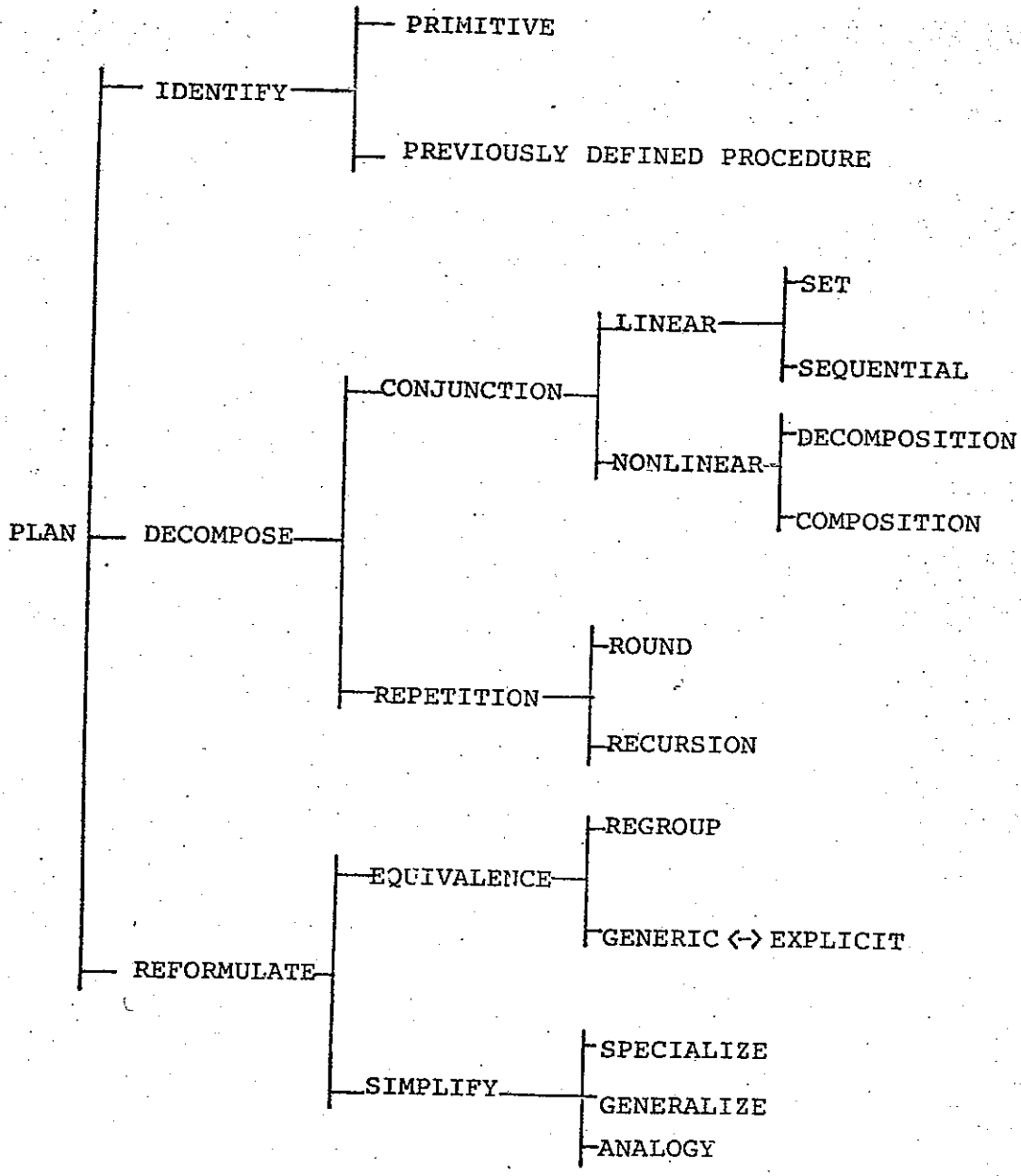


FIGURE 1
TAXONOMY OF PLANNING CONCEPTS

problem.

These definitions are formalized by representing them in an augmented transition network [Woods 1970]. Possible planning decisions are modelled as transitions between nodes of the network. The semantic context, including the problem description, is defined using the ATN's registers. Pragmatic knowledge, specifying which planning strategies to apply in which situations, is modelled by arc transition constraints. Figure 2 provides a global view of the PATN problem solver defined in this fashion, showing the connections between the various planning states.

The following augmentations are involved in converting the planning taxonomy to procedural form:

(1) *Registers*: Several registers are introduced to carry the semantics of the problem solving process. This includes the predicate logic specifications for the procedure currently being constructed (Model), and the currently proposed sequential solution (S).

(2) *Arc Ordering*: The arcs emanating from each node (representing alternative planning decisions) are ordered, thereby defining a backtracking algorithm. The default ordering from a given node is clockwise, beginning at the entrance point of the incoming arc. This ordering embodies prior judgments about the relative simplicity and probability of success of alternative planning methods.

(3) *Arc Predicates*: The basic arc ordering is supplemented by associating conditions (predicates) with arcs. In the ATN formalism, arc predicates are employed to determine the legality of a transition. By examining the contents of the registers, these arc predicates can make planning choices more sensitive to the problem context.

(4) *Arc Actions*: The contents of the registers may be modified by actions associated with various arcs. The actions are performed if and only if the arc is followed.

(5) *Linearization Cycle*: Interactions between instances of predicates can cause naive interpretations of predicate calculus formulae as programs to fail. As a result of carefully examining this difficulty, a linearization cycle has been introduced into PATN. If the linearization cycle is followed, the M register, containing the predicate logic problem description, is altered to reflect a non-linear decomposition.

(6) *Refinement Loop*: A sequential refinement loop is introduced, which selects a solution order for subgoals and recursively solves for them.

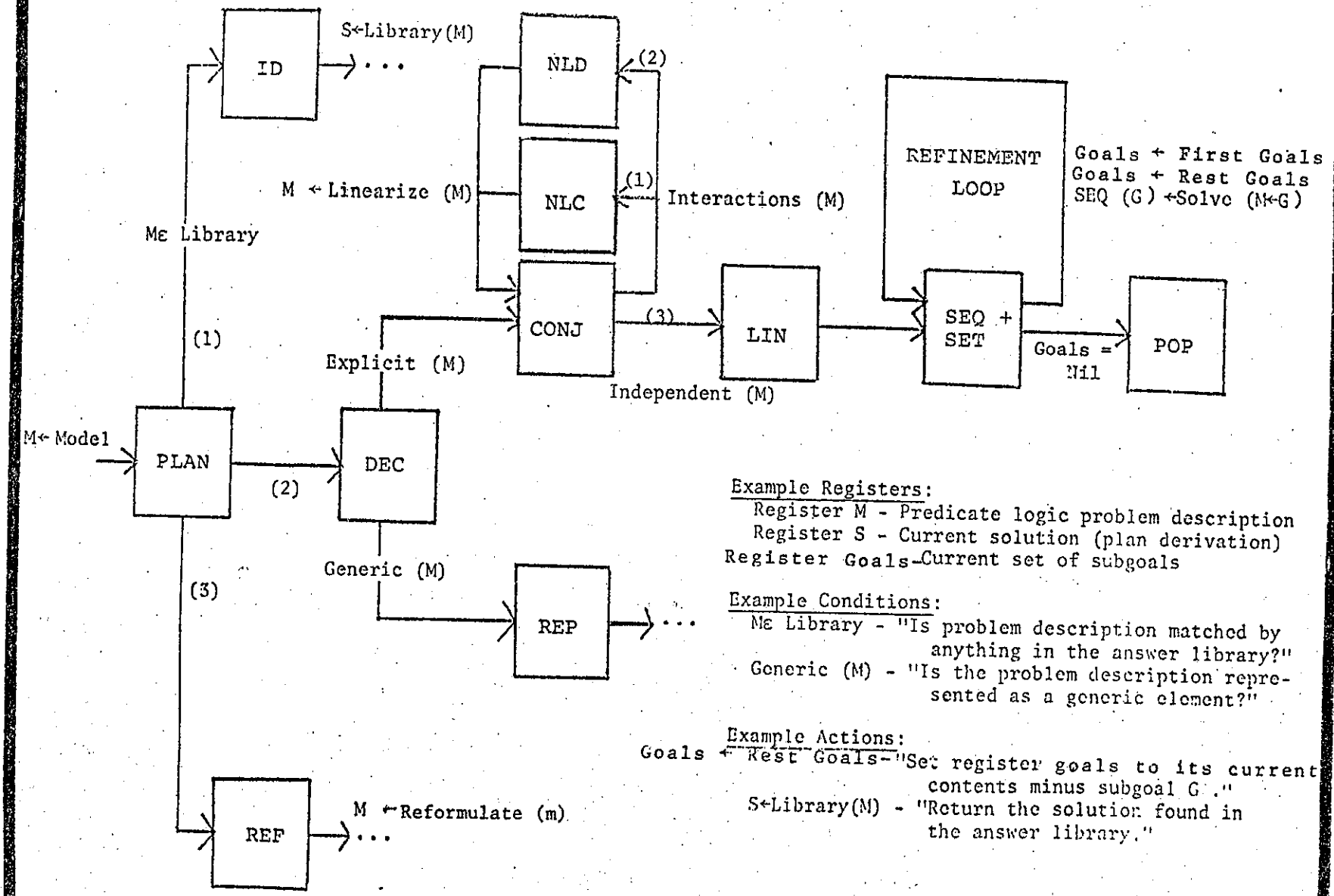


FIGURE 2 - A GLOBAL VIEW OF PATN

The possibility of *rational errors* makes debugging an important part of the theory. Rational errors are defined as mistakes in planning that arise from the use of reasonable heuristics. An example of a rational error is failure to recognize a particular interaction between predicates due to insufficient knowledge of the domain. This is developed by designing *DAPR* (an acronym for *Debugger of Annotated PRograms*), a debugging module for use with PATN. In DAPR terms, *diagnosis* is the isolation of incorrect or incomplete transitions made between ATN states during the planning process. *Repair* consists of re-planning, guided by advice from the diagnosis. A description of basic bug types in terms of specific errors in the planning process is undertaken. DAPR diagnoses and repairs *annotated* programs, in that a record of PATN's planning decisions (the *derivation tree*) is expected to be associated with the code.

Examples of the PATN approach will be drawn from two benchmark AI domains: the blocks world and the Logo turtle world. Blocks world problem solvers include SHRDLU [Winograd 1972], BUILD [Fahlman 1974], HACKER [Sussman 1973] and NOAH [Sacerdoti 1975]. Hence, applying PATN to the blocks world provides a common set of problems for comparison. The virtues of the Logo graphics world [Papert 1971] are: (a) graphics is an environment in which multiple problem descriptions are possible, ranging from Euclidean geometry to Cartesian geometry; (b) the possible programs range over a wide spectrum of complexity; and (c) extensive documentation exists on human performance in this area [G. Goldstein 1973; Okumura 1973].

References

Fahlman, Scott, "A Planning System for Robot Construction Tasks," in *Artificial Intelligence*, vol. 5, 1974, pp. 1-49.

Goldstein, Gerrianne, *LOGO Classes Commentary*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, LOGO Working Paper 5, February, 1973.

Kowalski, Robert, *Predicate Logic as a Programming Language*, University of Edinburgh, Department of Computational Logic, Schoo' of Artificial Intelligence, Memo 70, 1973.

Okumura, K., *LOGO Classes Commentary*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, LOGO Working Paper 6, February 1973.

Papert, Seymour A., *Teaching Children Thinking*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 247 (LOGO Memo 2), 1971.

Sacerdoti, Earl, "The Nonlinear Nature of Plans," in *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, September 1975, pp. 206-218.

Sussman, Gerald Jay, *A Computational Model of Skill Acquisition*, New York, American Elsevier, 1975; and Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 297, 1973.

Winograd, Terry, *Understanding Natural Language*, New York, Academic Press, 1972.

Woods, William A., "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM*, Vol. 13, No. 10, October, 1970, pp. 591-606.

NOTES

Some Thoughts on Resolution and Natural Deduction

Malcolm C. Harrison, Courant Institute, NYU.
Norman Rubin, Penn State University

It has been clear for some time that in any comparison between different methods of deduction, resolution would suffer because of its inefficient treatment of the equality property. In a recent paper (1) we reported a new method for incorporating equality into resolution, called equality-generalized resolution (e-g-resolution), which appears to be substantially more efficient than others which have been reported. It was shown that for Horn sets e-g-resolution was complete, and that the equality axions and any subset of the positive equality unit clauses could essentially be absorbed into the unification procedure. An implementation of e-g-resolution called coercion has been implemented in a system called DILEMMA (2), together with evaluatable predicates and functions, heuristic guidance functions, and a technique for facilitating inferences called completions. Experiments with DILEMMA suggest that these facilities combine to give a much more powerful deductive system than can be obtained by standard resolution. DILEMMA has solved the missionaries and cannibals problem, for example, deriving a solution equivalent to a resolution proof of 93 steps, and has also solved the geometry problems first proved by Gelernter,(3) and subsequently by Nevins (4) - a procedural approach by Goldstein (5) was less successful. The geometry problems were solved without the aid of a diagram, and without a specialized construction mechnism; constructions were done by the unification procedure using an 'intersection' function.

The results obtained with DILEMMA have reinforced our view that it is premature to dismiss resolution-based systems as being inherently inferior to other systems, and in particular to so-called 'natural deduction' systems. Natural deduction is a vague term which seems to be used to describe a number of deductive programs which appear to have in common the objective of producing proofs which are similar to those produced by humans. These programs are usually more complex and more specialized than resolution programs, and are considerably more difficult to analyze. It is our view that the main distinction between the two approaches is in fact this latter property; resolution enthusiasts wish to be able to make definite statements about the theoretical power of their system, which natural deduction enthusiasts are more concerned with effectiveness. We think that in a field so prone to overstatement it is important to be able to make precise statements about the range of problems which can be solved by a system, even if these statements are only correct in theory, and to carry out comparative experiments when problems are potentially solvable by more than one program.

Recently we have been looking at the interesting results obtained by the natural deduction program of Nevins (6). This program used 12 rules of inference to refute a formula; of these, 7 are concerned with Boolean transformations, 3 are related to modus ponens, and 2 are concerned with equality. The unification procedure uses specialized techniques for dealing with associativity and commutativity.

It is our impression that the rather impressive results obtained by this program are mainly due to two factors: (1) a relatively efficient but somewhat ad hoc treatment of equality; (2) the use of a considerable number of heuristics to cut down the size of the search space. We discuss these briefly below.

The treatment of equality in Nevin's program is not easy to characterize precisely. It permits substitutions of equal terms with a preference for simplifying substitutions and with built-in recognition of associativity and commutativity, and also permits a restricted form of non-equality inference from almost complementary literals. While this seems effective, it appears to have the same disadvantage as para-modulation--namely that substitutions seem to be made without regard for whether the resulting literal will be useful. For this reason we suspect that e-g-resolution could provide a superior treatment of general equality, though not as efficient for associativity and commutativity.

The heuristics used by Nevin's program include a strategy somewhat similar to unit preference in resolution, the preference of simplifying substitutions, and the distinction between $x \supset y$ and the formally identical $\neg x \vee y$ in making inferences, as well as other heuristics not described as easily. As we see it, the success of Nevin's program emphasizes once again the importance of carefully designed heuristics; However, we do not think that these heuristics are necessarily associated with natural deduction itself; it seems likely that similar effects could also be obtained in a resolution program.

The main theoretical advantage of natural deduction is that the resulting proof is not restricted to a sequence of clauses, but may contain formulae which are equivalent sets of clauses. Thus a natural deduction proof can be shorter. However, in practice it is not clear that the optimum use can be made of this ability. Nevin's program uses splitting into cases to reduce complex formulae to single literals; a refutation of $x \vee y$, $\neg x$, $\neg y$, for example, where x and y are not literals, would not be done in 3 steps. In fact the main advantage taken of the more general class of formulae permitted in Nevin's program appears to be that the occurrences of an existential variable are not treated as independent in descendent clauses; this effect could be obtained in resolution by adding an instantiation of each clause containing a Skolem function whenever a substitution is made inside that Skolem function, and arranging the heuristics so that these clauses were considered before the non-instantiated clauses.

A comparative study is being made by Aizik Leibovitch of the performance of Nevin's program and that of DILEMMA. We hope to be able to report on some of the results at the conference.

REFERENCES

(1) M.C. Harrison and N. Rubin, "A Generalization of Resolution", to be published in JACM.

(2) N. Rubin, "A Hierarchical Technique for Mechanical Theorem Proving and its Application to Programming Language Design", Courant Computer Science Report #10, November 76.

(3) H. Gelernter, "Realization of a Geometry Theorem Proving Machine", in "Computers and Though", ed. by E. Feigenbaum and J. Feldman, McGraw-Hill, 1963.

(4) A. Nevins, "Plane Geometry Theorem Proving Using Forward Chaining", Art. Int. Spring 1975.

(5) I. Goldstein, "Elementary Geometry Theorem Proving", MIT AI Memo 280, April 73.

(6) A. Nevins, "A Human Oriented Logic for Automatic Theorem Proving", JACM October 1974.

NOTES

ATP 33
February 1977

A Maximal Method for Set Variables in Automatic Theorem Proving

W.W. Bledsoe

ABSTRACT. A procedure is described which gives values to set variables in automatic theorem proving. The result is that a theorem is thereby reduced to first order logic, which is often much easier to prove. This procedure handles a part of higher order logic, a small but important part. It is not as general as the methods of Huet, Andrews, Pietrzykowski, and Haynes and Henschen, but it seems to be much faster when it applies. It is more in the spirit of J.L. Darlington's F-Matching. This procedure is not domain specific: results have been obtained in intermediate analysis (the intermediate value theorem), topology, logic, and program verification (finding internal assertions).

This method is a "maximal method" in that a largest (or maximal) set is usually produced if there is one.

A preliminary version has been programmed for the computer and run to prove several theorems. Some completeness results are given.

NOTES

A THEOREM PROVER FOR ELEMENTARY SET THEORY

by F.M. Brown

1. Introduction

This is a report of some of our research carried out during the summer and fall of 1974. It describes an implementation of a throrem prover based on truth value preserving transformations which has been applied to proving theorems in the domain of elementary set theory, specifically in the set theory system developed by Quine in his book: Set Theory and its Logic.

Our theorem prover consists of an interpreter for mathematical symbols, and many items of mathematical knowledge. We begin by describing this interpreter, and its basic method of evaluation by need. After this, we then describe the items of our sequent calculus, and in particular the various rationals behind each of our restrictions and strategies such as:

The forcing restriction (Bledsoe)

The forcing strategy

The variable restriction

After describing the logical systems, we then describe the set theoretic knowledge used by this theorem prover, in particular the axioms, reduction lemmas, existence lemmas, and definitions. We then give two example protocols of the theorem provers attempt to prove some theorems. The two examples are a generalization of Cantor's Theorem (the 17th formulae in Chapter 28 of Quine's book and the following theorem:

Example 1

THS101 The Cartesian product of two abstracts is contained in the powerset of the powerset of their union.

$$\vdash \alpha\times\beta \subseteq PP(\alpha \cup \beta) \qquad :Q2P2$$

$$\vdash \forall x\ x\varepsilon\alpha\times\beta \supset x\varepsilon PP(\alpha \cup \beta) \qquad :\vdash \forall$$

$$\vdash c\ \varepsilon\alpha\times\beta \supset c\varepsilon PP(\alpha \cup \beta) \qquad :\vdash\supset$$

$$c\ \varepsilon\alpha\times\beta \vdash c\varepsilon PP(\alpha \cup \beta) \qquad :Q9P11$$

$$c\varepsilon\{\langle xy\rangle : x\varepsilon\alpha \wedge y\varepsilon\beta\} \vdash c\varepsilon PP(\alpha \cup \beta) \qquad :Q9P4$$

$$c\varepsilon\{u:\exists x\exists y\ u=\langle xy\rangle \wedge x\varepsilon\alpha \wedge y\varepsilon\beta\} \vdash c\varepsilon PP(\alpha \cup \beta) \qquad :Q2P1$$

$$\exists x\exists y\ c=\langle xy\rangle \wedge x\varepsilon\alpha \wedge y\varepsilon\beta \vdash c\varepsilon PP(\alpha \cup \beta) \qquad :\exists\vdash$$

$$\exists y\ c=\langle ay\rangle \wedge a\varepsilon\alpha \wedge y\varepsilon\beta \vdash c\varepsilon PP(\alpha \cup \beta) \qquad :\exists\vdash$$

$$c=\langle ab\rangle \wedge a\varepsilon\alpha \wedge b\varepsilon\beta \vdash c\varepsilon PP(\alpha \cup \beta) \qquad :\wedge\vdash$$

$$c=\langle ab\rangle,\ a\varepsilon\alpha \wedge b\varepsilon\beta \vdash c\varepsilon PP(\alpha \cup \beta) \qquad :[=\vdash, E2]^*$$

$$a\varepsilon\alpha \wedge b\varepsilon\beta \vdash \langle ab\rangle \varepsilon PP(\alpha \cup \beta) \qquad :\wedge\vdash$$

$$a\varepsilon\alpha,\ b\varepsilon\beta \vdash \langle ab\rangle \varepsilon PP(\alpha \cup \beta) \qquad :D1$$

$$a\varepsilon\alpha,\ b\varepsilon\beta \vdash \langle ab\rangle \varepsilon \{u:u\subseteq P(\alpha \cup \beta) \qquad :[Q2P1, E2]^*$$

$$a\varepsilon\alpha,\ b\varepsilon\beta \vdash \langle ab\rangle \subseteq P(\alpha \cup \beta) \qquad :Q2P2$$

$$a\varepsilon\alpha,\ b\varepsilon\beta \vdash \forall x\ x\varepsilon\langle ab\rangle \supset x\varepsilon P(\alpha \cup \beta) \qquad :\vdash \forall$$

$$a\varepsilon\alpha,\ b\varepsilon\beta \vdash d\varepsilon\langle ab\rangle \supset d\varepsilon P(\alpha \cup \beta) \qquad :\vdash\supset$$

$$a\varepsilon\alpha,\ b\varepsilon\beta,\ d\varepsilon\langle ab\rangle \vdash d\varepsilon P(\alpha \cup \beta) \qquad :Q9P1$$

$$a\varepsilon\alpha,\ b\varepsilon\beta,\ d\varepsilon\{\{a\}\{ab\}\} \vdash d\varepsilon P(\alpha \cup \beta) \qquad :Q7P1B$$

$$a\varepsilon\alpha,\ b\varepsilon\beta,\ d\varepsilon\{u:u=\{a\}\vee u=\{ab\}\} \vdash d\varepsilon P(\alpha \cup \beta) \qquad :Q2P1$$

$$a\varepsilon\alpha,\ b\varepsilon\beta,\ d=\{a\}\vee d=\{ab\} \vdash d\varepsilon P(\alpha \cup \beta) \qquad :D1$$

aεα, bεβ, d={a}∨d={ab} → dε{u: u⊆α∪β} :Q2P1

aεα, bεβ, d={a}∨d={ab} → d⊆α∪β :Q2P2

aεα, bεβ, d={a}∨d={ab} → ∀xxεd⊃xεα∪β : → ∀

aεα, bεβ, d={a}∨d={ab} → eεd ⊃ eεα∪β : → ⊃

aεα, bεβ, d={a}∨d={ab}, eεd → eεα∪β :Q2P4

aεα, bεβ, d={a}∨d={ab}, eεd → eε{x: xεα∨xεβ} :Q2P1

aεα, bεβ, d={a}∨d={ab}, eεd → eεα∨eεβ : → ∨

aεα, bεβ, d={a}∨d={ab}, eεd → eεα, eεβ :∨ →

Case 1

aεα, bεβ, d={a}, eεd → eεα, eεβ :[=→,Q7P12]*

aεα, bεβ, eε{a} → eεα, eεβ :Q7P1A

aεα, bεβ, eε{x: x=a} → eεα, eεβ :Q2P1

aεα, bεβ, e=a → eεα, eεβ := →

aεα, bεβ → aεα, aεβ :atom

□

aεα, bεβ, d={ab}, eεd → eεα, eεβ :[= →,Q7P13]*

aεα, bεβ, eε{ab} → eεα, eεβ :Q7P1B

aεα, bεβ, eε{x: x=a∨ x=b} → eεα, eεβ :Q2P1

aεα, bεβ, e=a ∨e=b → eεα, eεβ : ∨ →

Case 2.a

aεα, bεβ, e=a → eεα, eεβ : = →

aεα, bεβ → aεα, aεβ :atom

□

Case 2.b

aεα, bεβ, e=b → eεα, eεβ := →

aεα, bεβ → bεα, bεβ :atom

□

time = 626 millisec

NOTES

Incorporating Mathematical Knowledge into an Automatic Theorem Proving System, investigated for the Case of Automata Theory, I.

S. Daniels, M. Livesey, Ch. Mathis,
P. Raulefs, J. Siekmann, W. Stephan,
E. Unvericht, G. Wrightson.

Institut für Informatik I
Universität Karlsruhe
D-7500 Karlsruhe 1
West Germany

Abstract:

An overview of some of the results obtained in an automatic theorem proving project at University of Karlsruhe is presented. We concentrate on theoretical results concerning the connection graph proof procedure and some special purpose unification algorithms.

Keywords: Artificial Intelligence, Automatic Theorem Proving, Matching Algorithms, Connection Graph Proof Procedure, T-unification.

0. Hypothesis.

In this paper we present some of the results obtained in an automatic theorem proving (ATP) project at the University of Karlsruhe [(i)].

The working hypothesis for this project is that ATPs have obtained a certain level of performance, which will not be significantly improved by:

(i) developing more intricate syntactically oriented derivation strategies (like e.g. lock-resolution, linear-resolution ...) nor by

(ii) using different logics, inference rules etc.

The relative weakness as compared to human performance of current ATP-systems is due to some extend to the lack of the rich mathematical and extramathematical knowledge human mathematicians have: in particular, knowledge about the subject and knowledge of how to find proofs in that subject.

Hence the object of this project is to make this knowledge explicit for the case of Automata Theory, to find appropriate representations and to find ways of how to use it.

(i) "Untersuchung zur Einbeziehung mathematischen Wissens beim Automatischen Beweisen am Beispiel der Automatentheorie", DFG-Forschungsprojekt, Aktenzeichen De 238/1

NOTES

ANALOGY AS A HEURISTIC FOR MECHANICAL THEOREM-PROVING

(Extended Abstract)

James Curie Munyer
Dept. of Info. Sci.
University of California
Santa Cruz, Ca. 95064

It is argued that human intelligence involves the use of non-logical or non-rigorous methods even in a rigorous task such as proving a theorem. The use of analogy in mechanical theorem proving is studied in this light. We describe an implementation which has a store of previously proved theorems and their proofs; when given a new theorem to be proved, it can find an analogy mapping from a previously proved theorem to the new theorem and use the analogy to achieve an often dramatic reduction in the computation required to construct a proof.

The generation of analogy mappings is actually fairly straightforward. By observing the close relation between analogy and induction, analogy mappings can be efficiently generated using existing induction (generalization) methods. If formula I is a common induct of formulas A and B, with substitutions α and β such that $I\alpha=A$ and $I\beta=B$, then an analogy mapping from A to B is $\alpha^{-1}\beta$. (It is possible to insure that α is invertible by distinguishing different occurrences of the same symbol in the formulas.) This can be done as simply as a unification operation, and hence looking for an analogy does not

place a significant overhead on the system.

Using an analogy in a deduction is a more difficult problem, because of the inherent non-rigorous nature of analogy. Specifically, problems arise when an analogy mapping changes during a deduction or when the analogy is not applicable at every step. These are characteristics of "weak analogies" which are nevertheless used routinely in human reasoning. Two previously studied methods of using analogies are discussed: using analogy to generate a plan or outline for a proof, and using analogy to choose a set of axioms from which a proof is attempted. The first method can potentially achieve an exponential reduction in computation, and can use analogies which are not applicable at every step of a proof. However, it cannot be used when the analogy changes because it is not possible in general to predict how the analogy will change, although some special cases are shown in which it is possible to do this. The second method can potentially achieve only a polynomial reduction in computation, and it cannot use an analogy which is not applicable at every step of a proof because it will not include an axiom used at a step where the analogy does not apply.

In our implementation analogies are used as a heuristic to guide a proof. Basically, when an "interesting" analogy mapping is detected between a clause generated while attempting to prove a theorem and a clause in the proof of a previously proved theorem, then a favorable heuristic evaluation is given to the newly generated clause. Some simple but effective methods are shown

for deciding when an analogy mapping is "interesting". This can achieve the same exponential reduction in computation as when analogy is used to generate a plan, but it has the advantage that it is not necessary to predict the intermediate steps in advance, as in a plan, so that it can use analogies which change.

Another advantage of this method is that an analogy can guide the proof through "loops" in the deduction; that is, cases where a group of steps in a previously proved theorem are repeated several times in the proof of a new theorem, usually accompanied by a change in the analogy mapping. A simple example is the evaluation of 4! by analogy with 2!; an analogy actually applies at every step even though 4! requires 16 steps and 2! only 10 steps.

Several examples are shown of the use of analogy in evaluating sums, demonstrating all the features discussed. The evaluation of $\sum_{i=0}^{n} i$ can provide an analogy for evaluating $\sum_{i=0}^{n} i^k$ for any k, which thus requires computational effort only linear in k instead of exponential in k. The evaluation of $\sum_{i=0}^{n} b^i$ can similarly be used to evaluate $\sum_{i=0}^{n} i^k b^i$ for any k; it can also be used to guide the evaluation of $\sum_{i=0}^{n} (\sum_{j=0}^{i} r^j) b^i$, reducing the computational effort to the cube root of what would be required without an analogy.

NOTES

Table 2: Definitions

Name	Definition	English translation
Q2P1:	$\forall y\ y\varepsilon\{x:\Gamma x\} \leftrightarrow \Gamma y$	the abstract of all x such that Γ
Q2P2:	$\alpha\subseteq\beta \leftrightarrow \forall x\ x\varepsilon\alpha \rightarrow x\varepsilon\beta$	is contained in
Q2P4:	$\alpha\cup\beta = \{x:x\varepsilon\alpha \vee x\varepsilon\beta\}$	union
Q2P5:	$\alpha\cap\beta = \{x: x\varepsilon\alpha \wedge x\varepsilon\beta\}$	intersection
Q2P7:	$\alpha=\beta \leftrightarrow \forall x\ x\varepsilon\alpha \leftrightarrow x\varepsilon\beta$	equals
Q2P8:	$\emptyset = \{z: \neq z\}$	null set
Q2P9:	$V = \{z: z=z\}$	universe
Q5P5:	$\{x:\Gamma x\}\varepsilon\beta \leftrightarrow \exists y\ y=\{x:\Gamma x\} \wedge y\varepsilon\beta$	the set $\{x:\Gamma x\}$ is in β
Q7P1a:	$\{\alpha\} = \{z: z=\alpha\}$	unit set
Q7P1B:	$\{\alpha\beta\} = \{z: z=\alpha \vee z=\beta\}$	pair set
Q9P1:	$\langle\alpha\beta\rangle = \{\{\alpha\}\{\alpha\beta\}\}$	ordered pair
Q9P4:	$\{\langle xy\rangle: \Gamma xy\} = \{u:\exists x\exists y\ u=\langle xy\rangle \wedge \Gamma xy\}$	abstract of ordered pairs
Q9P6:	$^{\circ}\alpha = \{\langle xy\rangle: \langle xy\rangle\varepsilon\alpha\}$	relational part
Q9P11:	$\alpha\times\beta = \{\langle xy\rangle: x\varepsilon\alpha \wedge y\varepsilon\beta\}$	cartesian product
Q9P14:	$\alpha"\beta = \{x:\exists y\ \langle xy\rangle\varepsilon\alpha \wedge y\varepsilon\beta\}$	image
Q10P1:	$\text{Func}\ \alpha \leftrightarrow (\forall x\forall y\forall z\ \langle xz\rangle\varepsilon\alpha \wedge \langle yz\rangle\varepsilon\alpha \rightarrow x=y) \wedge \alpha={}^{\circ}\alpha$	is a function
Q10P11:	$\alpha'\beta = \eta\, y\langle y\beta\rangle\varepsilon\alpha$	apply
D1:	$P\alpha = \{u: u\subseteq\alpha\}$	powerset
Q11P1:	$\alpha \leqslant \beta \leftrightarrow \exists f\ \text{Func}f \wedge \alpha\subseteq f"\beta$	The cardinality of α is less than or equal to the cardenality of β
Q20P3:	$\alpha < \beta \leftrightarrow \sim\beta\leqslant\alpha$	The cardinality of α is less than the cardinality of β

Table 3: Reduction Lemmas:

Q6P4: $\alpha=\alpha \leftrightarrow \Xi$

Q7P7: $\forall x \forall y\ \{x\} = \{y\} \leftrightarrow x=y$

Q7P8A: $\forall x\ \forall y\ \forall z\ \{xy\} = \{z\} \leftrightarrow x=z \wedge y=z$

Q7P8B: $\forall x\ \forall y\ \forall z\ \{z\} = \{xy\} \leftrightarrow z=x \wedge z=y$

Q7P9: $\forall x\ \forall y\ \forall u\ \forall v\ \{xy\} = \{uv\} \leftrightarrow (x=u \wedge y=v) \vee (x=v \wedge y=u)$

Q9P3: $\forall x\ \forall y\ \langle xy\rangle = \langle uv\rangle \leftrightarrow x=u \wedge y=v$

Q9P5: $\forall x\ \forall y\ \langle xy\rangle \in \{\langle uv\rangle : \phi uv\} \leftrightarrow \phi xy$

CRL1: $\text{Func } f \rightarrow (\langle wy\rangle \in f \leftrightarrow w = f'y)$

Table 4: Existence axioms and axiom of extentionality

Q7P10A: $\emptyset \in V$

Q7P10B: $\forall x\ \forall y\ \{xy\} \in V$

Q4P1: $\forall x\ \forall y\ \forall z\ (x=y \wedge x \in z. \supset y \in z)$

Table 5: Existence Lemmas:

Q7P12: $\{\alpha\} \in V$

Q7P13: $\{\alpha\beta\} \in V$

E2: $\langle\alpha\beta\rangle \in V$

Formulas for Generating Plans

Sharon Sickel
Information Sciences
University of California
Santa Cruz,
California

It has been shown that any problem expressible as a theorem in the predicate calculus can be represented by a context-free attribute grammar such that the language of the grammar represents the plans that solve the problem.* A closed form for the language is often derivable; the notation for the closed form is a regular algebra -- an extension of regular expressions. Formal language theory can be used to simplify the grammar and the corresponding language.

Theorem proving has been used in the past for question answering and to generate or verify solutions to specific (ground case) problems. Here we generate algorithms. For example, solve factorial(n) (rather than factorial(6)), or answer subset(S,T) (rather than subset($\{1, 2\}$, $\{0, 1, 2, 5\}$)). The closed form derivable from the grammar mentioned above gives the control structure of the algorithm. There may be more than one closed

*Formal Grammars as Models of Logic Derivations, Sharon Sickel, submitted to IJCAI 77.

form that will accomplish the task, and we choose among them. In this way we avoid the complexity of having to describe all solutions, and instead choose one that lends itself to execution. The data manipulation of the steps of the algorithm is given by the unification of components of the problem specification. Specifically, assignment statements in the algorithm assign values to the variables that correspond to values the variables are unified with in the specification.

The closed form also provides information about certain properties of the algorithm. The domain and range of the task are derivable from the closed form by replacing terminal symbols by substitutions and performing an operation on them similiar to composition. The closed form may also describe how to compute recursively defined functions iteratively by determining a priori when loops will terminate and by discovering an upper bound on the amount of information required at any one time. The original specification may inherently imply an algorithm containing redundancies. We may be able to automatically improve such algorithms. For example, if the zero function is described recursively, the implied computation is inefficient. However our analysis shows that the range consists of a single element. Therefore the algorithm for the function can be transformed to one that maps directly onto the single range element. Another example of this simplification occurs in generating plans for travel on a Manhattan grid with no barriers. To go from point (0,0) to point (m,n), we could do an arbitrarily large amount of meandering. However,

it is possible to derive a plan that will accomplish the trip in m+n steps by using a less general closed form that nevertheless achieves the task for all m and n.

Some problems may be so hard that finding the closed forms directly from the grammar is not practical(or even possible). In these cases, we may be able to induce a general plan by solving the problem for a small set of elements of the domain and generalizing on those solutions. The generalization is guessed from the examples and must then be verified for the entire domain, usually by mathematical induction on the construction operator of the domain. If the domain is finite or recursively defined, this proof should be automatic.

The closed form used here provides an interface between formal specification of problems and the algorithms that solve them. The automatic generation of these forms is a step toward mechanized plan formation.

NOTES

THEOREM PROVING IN TYPE THEORY

Peter B. Andrews and Eve Longini Cohen

Mathematics Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

As one aspect of the endeavor to create new intellecual tools for mankind, we wish to enable computers to prove, and to assist in the proofs of, theorems of mathematics and (eventually) other disciplines which have achieved the requisite logical precision. For this purpose, a particularly suitable formal language is Church's formulation [4] of type theory with λ-conversion. In this language traditional mathematical notations can be expressed very directly, and the intuitive distinctions between different types of mathematical entities (such as numbers, functions, and sets of functions) are made syntactically explicit.

The program for proving theorems of type theory which we discuss is intended to provide experience relevant to such a project, and was developed with the aid of Charles E. Blair and John J. Grefenstette.

The user first types into the computer the sentence of type theory which is to be proved, using the usual notations and abbreviations of symbolic logic. (Of course, many mathematical statements can be expressed much more easily and naturally in type theory than in first order logic.) The types of variables need be mentioned only once. Many traditional mathematical notations can be used, since the system contains definitions of concepts such as equality, uniqueness, union, intersection, power set, image (of a set under a function), iteration, injection, associativity, group, and

---

This research was supported by NSF Grants DCR71-01953-A04 and MCS76-06087.

topological space. Additional definitions, which may contain type variables, can readily be added.

$[X_\alpha = Y_\alpha]$ is regarded as an abbreviation for $[Q_{o\alpha\alpha}X_\alpha Y_\alpha]$, where the equality relation $Q_{o\alpha\alpha}$ is defined to be $[\lambda x_\alpha \lambda y_\alpha \forall p_{o\alpha} . p_{o\alpha}x_\alpha \supset p_{o\alpha}y_\alpha]$. Since equality can be defined in type theory, no special axioms or rules of inference for equality are logically necessary, though they may prove useful.

The negation of the sentence to be proved is reduced to a set of clauses as in [1], essentially as in the resolution method. Definitions are instantiated by applying rules of λ-conversion. The program then seeks an acceptable mating [3], using Huet's algorithm [6] to find the required unifying substitutions.

In type theory unification of wffs A and B involves applying a substitution μ and then rules of λ-conversion, so that the λ-normal forms of μA and μB are identical. The process of constructing a unifying substitution may branch at each step, so the search for a unifying substitution is embodied in a matching tree [6], which may be infinite. The nodes of the tree are sets of pairs of wffs (disagreement pairs), and all disagreement pairs in some node must be simultaneously unified. We have found that matching trees often contain nodes which are (under appropriate renaming of variables) supersets (or duplicates) of other nodes, so our program deletes the redundant superset nodes. Heuristics are used to minimize the branching of the tree. In each branch of the matching tree, the possible substitutions for a variable are computed only once, and those not immediately applied are saved for future use.

Associated with each node N in the matching tree for the pair $\langle A,B \rangle$ of wffs is a substitution μ which must be composed with some other substitution to obtain a unifier of $\langle A,B \rangle$. We say that μ is a partial unifier of $\langle A,B \rangle$, and that it unifies $\langle A,B \rangle$ to

depth k, where k is the depth of the node N in the (downward) matching tree for <A,B>.

Before seeking an acceptable mating, the program finds the potential mates of each literal-occurrence by doing partial unifications, and constructs a connection graph similar to those in [8]. It prunes the graph by checking the horizontal consistency ([10], p.28) of the partial unifiers. It also deletes any clauses containing pure literals.

A potential mating (a relation between literal-occurrences which will be an acceptable mating if there is a substitution which makes mated literal-occurrences complementary) is built up by the method outlined in [3], with first priority given to finding mates for literal-occurrences in the set of support, if such a set has been specified. The unification procedure does not wait until a potential mating has been found, but works in parallel with the mating procedure. Each time a new pair of wffs is added to a partial mating, the unification procedure increases the depth of its search for a unifier associated with the partial mating. As the partial mating grows, the constraints on the unifying substitution are increased, and the branching of the matching tree is reduced. Also, as certain branches of the matching tree are eliminated, and others grow longer, the set of potential mates for each literal-occurrence is reduced. Thus the interaction between the mating and unification procedures limits the search space which each must explore.

Once a potential mating has been found, the unification procedure seeks to verify that it is acceptable. Since this may involve an endless search, the program simultaneously seeks new potential matings. After finding an acceptable mating, for the benefit of the user the computer constructs from it a more traditional refutation, using substitution, cut (ground resolution), and simplification of disjunctions as rules of inference.

The program can be run in automatic or interactive mode. The interactive system embodies a set of logical rules which is in a certain sense complete [1], but the user must provide some of the substitutions for predicate variables. (As shown in [2], even completeness in this weak sense is not trivial.) The program in automatic mode is not logically complete for type theory (though it is complete when applied to sentences of first order logic), since no practical method is known for automatically generating all required substitutions for predicate variables. This is the fundamental theoretical problem of automatic theorem-proving in type theory, and no practical general solution to it seems imminent, since substitutions for predicate variables often express the important concepts in a mathematical proof.

As noted in [1], for certain purposes axioms of extensionality, descriptions, or choice must be taken as hypotheses. Actually, the introduction of Skolem functions to eliminate essentially existential quantifiers involves an implicit use of the Axiom of Choice (AC) in type theory, so the system can prove certain consequences of AC, such as

$$\forall Y_\iota \exists Z_\iota [R_{o\iota\iota} Y_\iota Z_\iota] \supset \exists F_{\iota\iota} \forall Y_\iota [R_{o\iota\iota} Y_\iota [F_{\iota\iota} Y_\iota]].$$

Among the theorems which can be proved completely automatically are Cantor's Theorems that a set has more subsets than members, and that there are more functions on a non-unit set than members of the set. (Thus there are uncountably many functions of natural numbers.) Following [5], the Cantor Theorem for Sets can be expressed by the sentence $\sim \exists H_{o\iota\iota} \forall S_{o\iota} \exists J_\iota [H_{o\iota\iota} J_\iota = S_{o\iota}]$, which asserts that there is no function H from individuals to sets which has every set S in its range. The computer decides to substitute for $S_{o\iota}$ the wff $[\lambda X_\iota . \sim H_{o\iota\iota} X_\iota X_\iota]$, which denotes the set $\{x | x \notin Hx\}$, and expresses the key idea in the classical diagonal argument.

REFERENCES

[1] Peter B. Andrews, "Resolution in Type Theory", Journal of Symbolic Logic 36 (1971), 414-432.

[2] Peter B. Andrews, "Resolution and the Consistency of Analysis", Notre Dame Journal of Formal Logic XV (1974), 73-84.

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

[4] Alonzo Church, "A Formulation of the Simple Theory of Types", Journal of Symbolic Logic 5 (1940), 56-68.

[5] Gerard P. Huet, "A Mechanization of Type Theory", Third International Joint Conference on Artificial Intelligence, Stanford, California, August 1973, 139-146.

[6] Gerard P. Huet, "A Unification Algorithm for Typed λ-Calculus", Theoretical Computer Science 1 (1975), 27-57.

[7] D. C. Jensen and T. Pietrzykowski, "Mechanizing ω-Order Type Theory Through Unification", Theoretical Computer Science (to appear).

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

[9] J. A. Robinson, "New Directions in Mechanical Theorem Proving", Proceedings of the IFIP Congress, 1968, 206-210.

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

April, 1977

NOTES

Abstract Prepared for
A Workshop on Automatic Deduction
MIT, Cambridge, Mass.
Aug. 17-19, 1977

Berlin, April 13, 1977

WEAK SECOND ORDER LOGIC AS DATA BASE LANGUAGE

E. Konrad

Technical University of Berlin

1. Introduction

The purpose of this paper is to show how second order logic can be interpreted as data base language. We consider data base systems with deductive capabilities. A paradigma of a data base language is introduced, called WESOL (Weak Second Order Logic). The meaning of WESOL statements is precisely defined in terms of fixpoint semantics, model-theoretic semantics, and operational semantics.

Our work is partly based on Kowalkski's investigations in predicate logic programming (/3/) and motivated by problems in the area of data base systems (/2/). The semantical foundations are taken from Carnap's theory of meaning (/1/). A detailed discussion of the author's work is presented in (/4/).

2. Data Bases

Knowledge-based information systems have a data base as their main part. In our treatment a data base is a triple DB = $\langle K, A, F \rangle$, where the kernel K contains elementary sentences, the amplifier A empirical rules, and the filter F semantical integrity constraints. Syntactically, we have a class of atomic

formulas K, a class of Horn formulas A, and a class of second formulas F.

Retrieval and updating commands are executed by an abstract machine, which can be thought of as an interpreter. This machine must have the capability of doing second order deductions. A complete mechanization of second order logic is elaborated by Pietrzykowski (/5/).

The kernel K of the data base can be represented by a marked directed graph with one semantically fictitious node. An individual constant is assigned to each node, and a binary or an unary predicate to each arc. The restrictions of a predicate calculus to binary predicates is not essential as Löwenheim proved in 1915. It is shown that weak second order logic is adequate for describing every portion of the data base.

3. WESOL

WESOL is a precise model language which has the most important features of existing (nonprocedural) data base languages. WESOL can be considered as an applied second order predicate calculus with the higher order predicates TEST, FIND, ADD, DELETE and REPLACE. The meaning of predicate variables is restricted to predicates of finite extension.

a) Retrieval:

TEST (A), where A is a second order formula, can have the results $\mathfrak{l}$ (YES), $\mathcal{O}$ (NO), $\perp$ (I DO NOT KNOW), $\top$ (I AM DISTURBED). This is suggested by the fixpoint theory of Scott (/6/).
The result of the command FIND $((\lambda \pi_1 \ldots \pi_m)\ (\lambda x_1 \ldots x_n) A)$ is a matrix of predicates and individual constants.
TEST statements represent YES - NO questions, and FIND statements W-questions (who, where, what, ...).

b) Updating:

Changing the data base can be managed by three updating commands: DELETE (E), ADD (E), and REPLACE (E), where E is some elementary sentence in the kernel of the data base.

4. Semantics

We apply the above mentioned semantic approaches to the data base language WESOL. The fundamental concept "information content of a data base" is given three different explications. Carnap's method of extensions and intensions is included in our framework (/1/).

Explication 1 (fixpoint semantics):

The information content of a data base is the minimal fixpoint of its state space.

Explication 2 (model-theoretic semantics):

The information content of a data base is the class of consequences of its kernel.

Explication 3 (operational semantics):

The information content of a data base is the class of derivations of its kernel.

Explication 1 and explication 2 are equivalent, if the amplifier of the data base is a set of Horn clauses (Kowalski /3/). Explication 2 and explication 3 are equivalent, because the weak second order logic is complete.

The semantics of all WESOL statements can be defined with respect to the information content of the data base. The meaning of individual constants and predicates is given by the data base. Using semantical rules the meaning of complex WESOL statements can be determined recursively.

References:

/1/ Carnap, R.: Meaning and Necessity, Second Edition, Chicago Press 1956.

/2/ Date, C.J.: An Introduction to Data Base Systems, Addison-Wesley 1975.

/3/ Emden, M.H. van; Kowalksi, R.: The Semantics of Predicate Logic as Programming Language, Memo 73, Edinburgh 1974.

/4/ Konrad, E.: Formal Semantics of Data Base Languages (German), Thesis, Berlin 1976.

/5/ Pietrzykowski, T.: A Complete Mechanization of Second Order Type Theory, JACM 20 (1973), 333-364.

/6/ Scott, D.: Lattice Theory, Data Types and Semantics, in: Formal Semantics of Programming Languages, Prentice Hall 1972, 65-106.

Address: Dr. E. Konrad
TU Berlin
Otto-Suhr-Allee 18/20 VSH 9
D-1000 Berlin 10

Fed. Rep. Germany

NOTES

Some Issues in the Design of a Representational Language

(Extended Abstract)

P. L. Suzman

Department of Artificial Intelligence

University of Edinburgh

Introduction

Attempts to use First Order Predicate Calculus (FOPC) as a representational language have encountered two fundamental problems - the serious difficulties found in axiomatising problems and the inability of general purpose proving techniques to encompass even what might be called "common-sense" reasoning.

It is suggested that one of the fundamental reasons for this might be that FOPC is "neutral" in that it does not embody assumptions about how it will be used to model problems - it remains entirely uncommitted about the nature of the objects in the universe of discourse.

Just as the idea that sets play a fundamental role in mathematics gave a basis for a successful mathematical language (set theory), the idea is explored here that some such concept might prove similiarly fruitful in the design of a representational language for A.I.

On a Nominalistic View of the World

The particular concept suggested is that of an individual as espoused by Nominalist philosophers, particularly Goodman [1] and Eberle [2]. The world-view that this entails is of a world consisting of individuals whose parts (and "sums") are also individuals, and where individuals can enjoy certain (basic) properties - indeed are to be thought of as simply "bundles" of these properties.

Some Semantics for a Simple Nominalistic Language

A simple semantics for such a language proposed by Eberle is discussed. In this language, "b is red" is taken to be true iff the denotation of b overlaps (has something in common with) the denotation of red. Thus predicates have intensions as well as extensions - the predicate red is to be thought of as denoting the bundle which consists of all the properties (corresponding to different shades of red) which are present in all red things.

Computational Advantages

The computational advantages of such a language over FOPC as it is normally used are illustrated by means of an example involving mutually exclusive properties and hierarchies of properties. The advantages stem from both the intensional nature of the language, which allows one to reason about the properties of predicates, and the richer ontology, which allows one to shift between a logically

complex statement about several simple individuals and an equivalent logically simple statement about the complex individual which is their sum.

Other issues : Relations, Derived Properties, Sets, Views

The problem posed in such a system when dealing with relations is discussed, and some tentative solutions proposed.

This leads to the important related issue of derived properties which is discussed with particular emphasis on some of the different ways properties of complex objects can depend on properties of their parts. (Compare say colour and weight).

The need for some form of set is shown and a limited set theory in conjunction with the idea of the different "views" one can have of a complex object (e.g. one can regard a chess board as consisting of 64 squares or of 8 ranks) is proposed.

On Modelling a Changing World

The ease with which a representational language can cope with a world that can undergo change is suggested as a fundamental test for it. Indeed it is only in this problem that some of the philosophical issues discussed here are seen to be of practical importance.

The standard situational calculus [3] turns out to be an example of the use of FOPC in a manner not consistent with the world view suggested here, and an alternative formulation is proposed. In this the meaning of our predicates is left unchanged as compared with the static case (in contrast to the usual introduction of situational variable arguments), and instead we think in terms of individual concepts which may refer to different individuals in different situations. A particular situation is just then the totality of all individuals at some instant. The frame problem becomes simply that of determining which individual concepts still refer to the same individuals in the new situation as they did in the old.

On a Logic of Action

Hayes's proposals for a logic of actions [4] are re-interpreted in this context. Intuitively the proposed idea is that an action is regarded as affecting precisely some specified properties of some specified individuals. In the proposed system a distinction can be drawn between the "genuine" and "logical" side-effects of an action. The former occurs when, as a result of being somehow physically connected to the individuals directly affected by the action, another individual also becomes affected (e.g. a block is moved causing another block resting on it to also move).A logical side effect occurs in that some derived properties of a compound individual change because some properties of some part of that individual have been changed (e.g a block is moved causing the

distance between it and some other block to change).

Illustrations and Comparisons

The use of such a language is illustrated by means of an example in which Tic-Tac-Toe is modelled.

The possible use of it as a programming language by regarding steps of the program as actions is contrasted with the approach taken by Kowalski [5].

Finally the suggestions here are compared with the ideas used in KRL [6].

References

[1] Goodman,N. (1966): The Structure of Appearance (2nd ed.) Indianapolis.

[2] Eberle,R. (1970) : Nominalistic Systems, Dordrecht-Holland.

[3] McCarthy,J. & Hayes,P. (1969)Some philosophical problems from the standpoint of artificial intelligence, Machine Intelligence 4,pp. 463-502 (eds Meltzer,B & Michie,D.) Edinburgh University Press.

[4] Hayes,P.(1971): A logic of actions. Machine Intelligence 6, pp. 495-520 (eds Meltzer,B. & Michie,D.) Edinburgh University Press.

[5] Kowalski,R. (1974) Predicate logic as a programming language. Proc. IFIP Cong. 1974 pp. 569-574 . North-Holland.

[6] Bobrow,D. & Winograd,T. (1977): An overview of KRL, a knowledge representation language. Cog. Sc. 1 no. 1,1977.

NOTES

Experiments with Resolution-Based Theorem-Proving Algorithms

by E. L. Lusk and R. A. Overbeek

During the past ten years a variety of seemingly simple problems have proved intractable for resolution-based theorem provers. In this paper the authors present some enhancements made to an existing theorem prover which allowed the program to obtain proofs for several problems in a number of different areas which were very difficult for the original program. Most of these enhancements were based upon the previously proposed concepts of locking [2, 3], case analysis [5], bidirectional search strategies [4, 6], and qualification [7].

The first modification to the existing program involved the introduction of a bidirectional search strategy. It was found useful to maintain the distinction between two environments in several ways. The inference rules appropriate to one environment (roughly speaking, the environment analogous to forward-chaining from the hypothesis) were quite different from those found appropriate to the other environment (corresponding to back-chaining from the denial of the conclusion). The appropriate criteria for retaining and using derived clauses were also found to differ.

The second category of enhancements involved the partitioning of each clause into two sets of literals, those which are to be considered "active", and the others. Only the active literals are considered in determining whether a clause meets given criteria for retention and use. For example, if a clause containing a complex term is ordinarily "penalized" in some way, then

it will not be penalized if the complex term is in an inactive literal. Similarly, any algorithm which makes use of the fact that a clause is or is not a unit clause will regard a clause with only one active literal as a unit clause.

The partitioning of the literals in a clause is accomplished as follows. Lock numbers are assigned in decreasing order to literals appearing in ground clauses which are derived in the context of the "back-chaining" environment. A literal in a clause is said to be locked if there exists in the clause a literal with either a lower lock number or no lock number at all. A literal is a qualifier if it represents a condition of definition for a function appearing in another literal in the clause. (See [7] for a complete discussion of qualification.) Finally, a literal is active if it is neither locked nor a qualifier. This partitioning mechanism has a dramatic effect on the behavior of various inference rules and also leads to an implementation of case analysis.

To illustrate the usefulness of the above concepts, we will examine the performance of our theorem prover with the above enhancements on a set of five problems from widely differing areas, each of which was found to be either difficult or impossible before the enhancements were implemented.

The first of these is one of a set of problems on limits proposed by Bledsoe [1]. The theorem is that if $\lim(x \to x_0)\ f(x) = L_1$ and $\lim(x \to x_0)\ g(x) = L_2$ then for every $\varepsilon > 0$ there is a $\delta > 0$ such that if $|x-x_0| < \delta$, then $|f(x)-L_1+g(x)-L_2| < \varepsilon$. To the authors' knowledge this problem has not yet been done by any other domain-independent theorem-prover. Obtaining a proof of this theorem required the use of qualification

to cope with the functions $\delta_1(x)$ and $\delta_2(x)$ associated with f and g which are defined only for $x > 0$, and the use of the bidirectional search strategy to create the set of terms necessary for the proof.

In set theory, the problem $A \cup B = B \cup A$ appears trivial but can pose difficulties for resolution theorem provers because it is not a Horn set. The application of bidirectional searching and locking occurs in the derivation and use of the clause $\neg(A \cup B \subset B \cup A)$ or $\neg(B \cup A \subset A \cup B)$. One of the literals will be locked, so that the program will operate on the clause as if it were a unit until the clause containing the locked literal is derived (end of first case). This clause will subsume all the clauses used in its derivation, and the program will now proceed on the other literal (now unlocked).

The third problem, from group theory, was designed to stress the case analysis aspect of locking. Consider two groups G_1 and G_2, their multiplication tables given as axioms, and an explicitly defined isomorphism $f:G_1 \rightarrow G_2$. The problem is to prove that $f(x)f(y) = f(xy)$ for all x,y in G_1. If G_1 and G_2 are isomorphic to Z_2, this requires consideration of 8 cases, and in the Z_3 case, 27 cases. A further enhancement used in this problem was "conditional demodulation", in which clauses with all literals inactive except for one positive equality literal act as demodulators within the case being worked on. The time required to do the Z_2 problem was decreased by a factor of 7 when conditional demodulation was installed, and the Z_3 problem could not have been done without it.

The fourth problem, also in group theory, comes from a set proposed by Nevins [5]. If K is a subgroup of a group G and $g \in G$, then $g \in K$ if and only if $gK = K$. Qualification was heavily used to restrict the use of certain functions in contexts where they would not make sense.

The final problem is one of four verification conditions which arise from an attempt to verify an algorithm for finding the maximum element of a vector. Bidirectional search, locking, and conditional demodulation were all utilized in obtaining the proof.

In summary, we believe that our experiments have demonstrated the value of these ideas in significantly extending the power of resolution-based theorem provers while retaining their generality.

REFERENCES

1. W. W. Bledsoe, R. S. Boyer, and W. H. Henneman, Computer Proofs of Limit Theorems, Artificial Intelligence 3(1972), pp 27-60.

2. R. S. Boyer, Locking: a restriction of resolution, Ph.D dissertation, Univ. Texas, Austin, 1971.

3. C. L. Chang and RCT. Lee, Symbolic Logic and Mechanical Theorem Proving, New York: Academic Press, 1973.

4. D. Kuehner, Same special purpose resolution systems, Machine Intelligence 7 (1973), pp 117-128.

5. A. J. Nevins, A human oriented logic for automated theorem-proving, J.A.C.M. 21(1074) pp 606-621.

6. I. Pohl, Bidirectional search, Machine Intelligence 6 (1971), pp 127-140.

7. S. K. Winker, An evaluation of an implementation of qualified hyper-resolution, IEEE Transactions on Computers, vol. C-25, no. 8 (1976), pp 835-843.

NOTES

A Network-Based Knowledge Representation and its Natural Deduction System

by

Richard Fikes and Gary Hendrix

Abstract

We describe a knowledge representation scheme called K-NET and a deductive retrieval system called SNIFFER designed to answer queries using a K-NET knowledge base. K-NET uses a partitioned semantic net to combine the expressive capabilities of the first-order predicate calculus with full indexing of objects to the relationships in which they participate and with linkages to procedural knowledge. Facilities are also included for representing taxonomies of sets and for maintaining hierarchies of contexts. SNIFFER contains a logically complete set of natural deduction facilities that do not require statements to be converted into clause or prenex normal form. It uses a coroutine based control structure that constructs alternative proofs in pseudo-parallel and shares results among them. In addition, it uses deductive functions that embody the semantics of taxonomies, and allows augmentation by user-supplied derivation functions.

Introduction

This paper provides an overview of a network-based knowledge representation scheme called K-NET and a deductive retrieval system called SNIFFER designed to answer queries using a K-NET knowledge base. K-NET provides facilities that combine the expressive power of the first-order predicate calculus with a natural indexing scheme, a partitioning mechanism that allows the construction of hierarchical contexts, a taxonomy modeling

capability, and linkages to procedural knowledge. Typically, the representation scheme is used to create a model of some task domain about which questions are to be asked. SNIFFER (an acronym for Semantic Net Inference Facility Fortified with External Routines) provides a mechanism for obtaining information from such a knowledge base both by direct retrieval and by deduction. Such deductions may involve the invocation of procedures provided by users and are achieved through the coordination of multiple pseudo-parallel processes.

SNIFFER and K-NET are evolving systems, versions of which have been used as major components in larger systems developed in the SRI Artificial Intelligence Center, including the SRI Speech Understanding System (Walker 1976).

In this paper we first present what we consider to be the distinguishing or characterizing features of this system before focusing on a detailed description. Our goal is to highlight what we feel is interesting about what we have done, and to provide the reader who is familiar with similar efforts a set of observations that can be used to relate our work to other knowledge representation facilities and deductive retrieval systems. Following these lists of characterizing features, we provide an overview description of the system that elaborates on these features and provides intuition building examples.

Characterizing Features of SNIFFER

SNIFFER is a "natural" deduction system that is given two net structures as input, one representing a knowledge base and the other representing a query (usually a translation of a question stated in English). It treats the query as a pattern and attempts to find instances of the pattern in the knowledge base, or equivalently, it treats the query as a theorem to be proved and attempts to find instantiations for its existentially quantified variables. Answers are returned in the form of sets of "bindings" for the variables in the pattern. For example, the question "Who does John love?" is translated into a net structure representing the pattern "John loves x" (or the theorem $(\exists x)$[Loves(John,x)]), and SNIFFER returns

bindings for x such as (x, Mary). Answers may either be retrieved from the knowledge base or derived using knowledge base theorems and procedures.

SNIFFER can be characterized by considering the following list of features:

* Associative retrieval of relationships from the knowledge base is performed using the K-NET indexing facilities.

* Efficient, special purpose deductive procedures are used for extracting information from the K-NET taxonomies. For example, if the knowledge base indicates that x is an element of the set of Mustangs, that Mustangs are a subset of the set of sports cars, and that sports cars are a subset of the set of automobiles, then SNIFFER can conclude that x is an automobile by using procedures that follow the chain of **elementOf** and **subsetOf** arcs, thereby bypassing the more cumbersome, general-purpose deductive machinery.

* Facilities are included for answering questions and using knowledge base statements composed of conjunctions, disjunctions, and implications, containing arbitrarily embedded universally and existentially quantified variables.

* Queries and knowledge base statements are processed in the "natural" form in which they are input, without converting into a canonical form such as clause form or prenex normal form. This capability eliminates "explosive" conversions (such as converting the disjunction $(a \wedge b \wedge c) \vee (d \wedge e \wedge f) \vee (g \wedge h \wedge i)$ into clause form which consists of 27 clauses each containing 3 disjuncts) and unnecessary conversions (such as conversion of a disjunctive question's complex disjuncts when one of its simple disjuncts can easily be shown to be true). In addition, the intuitiveness and heuristic value of the form in which statements are input (as implications, for example) is maintained.

* A logically complete set of natural deduction rules are used that reason backwards from the question. These rules use such techniques as establishing subgoals, case

analysis, and hypothetical reasoning. For example, to answer a question that is in the form of an implication, SNIFFER might use hypothetical reasoning by assuming the implication's antecedent and then pursuing a proof of the consequent as a subgoal.

* A powerful coroutine based control structure allows the construction of alternative proofs in a pseudo-parallel manner, with results being shared among the alternatives. Each partial proof has its own local scheduler to determine how its proof attempt should be continued, and there is an executive scheduler that uses information supplied by the local schedulers to determine which partial proof is to be given control at each step. The various schedules provide the facilities necessary to allow reasonable heuristic guidance of the total deduction and retrieval process.

* User-supplied procedures may participate in the attempt to find answers in two ways. First, procedures included in the knowledge base may be invoked to access information in knowledge sources that are external to K-NET. Second, SNIFFER allows the inclusion of user-supplied procedures that extend the system's deductive strategies. Facilities are available to these procedures for creating alternative proofs, manipulating schedules, altering priorities, and establishing "demons" so that the user can create strategies that augment and interact with those that already exist in the system.

* A "generator" control structure (see Teitelman, 1975) is used that can be restarted after returning a answer to seek another answer to the same query. SNIFFER saves its internal state before returning an answer, and each time it is "pulsed", it continues from its previous state and seeks another answer. For example, the first pulse for the question "Who owns a Mustang?" may produce the answer "John", a second pulse may produce "Mary", etc. This style of answer production allows the user to examine each answer as it is produced and determine whether additional answers are needed.

* "No" answers are determined by finding an affirmative answer to the question's negation. For example, if given the question "Does John love Mary?", SNIFFER will attempt to prove "John does not love Mary" in addition to attempting to prove "John loves Mary".

NOTES

Compiling Deduction Rules From a Semantic Network Into a Set of Processes

Stuart C. Shapiro
Department of Computer Science
State University of New York at Buffalo
Amherst, New York 14226

Detailed Summary

For some time, we have been investigating the representation of deduction rules in semantic networks [1;2;9-13]. Recently, we have been implementing an inferencing system which, given the pattern for a piece of network to be deduced, locates relevant deduction rules [12], and "compiles" them into a set of processes which are then given to a multi-processing system for execution. The multi-processing approach was motivated partly by Kaplan's producer-consumer model of parsing [7] and partly by Wand's frame model of computation [14], which itself was based on the "little man" metaphor of Papert [8] and Hewitt's ACTOR model [3;4;5].

Deduction rules are represented in semantic network form for several reasons: they can be entered in the same way as other information, either in the same formal input language or in (some subset of) a natural language using the same parser and grammar; they can be treated as data - entered, retrieved, discussed, etc.; relevant deduction rules can be retrieved using the same network matching routines and in the same operation as retrieving explicit information; in semantic networks it is natural to represent a rule as a connective and an unordered set of arguments, delaying the decision of which argument(s) is(are) the antecedent(s) and which are(is) the consequent until the rule is to be used in a deduction. To illustrate the last point, consider the rule stating that the following propositions are equivalent:

1. Block x supports block y.
2. Block x is under block y.
3. Block y is above block x.

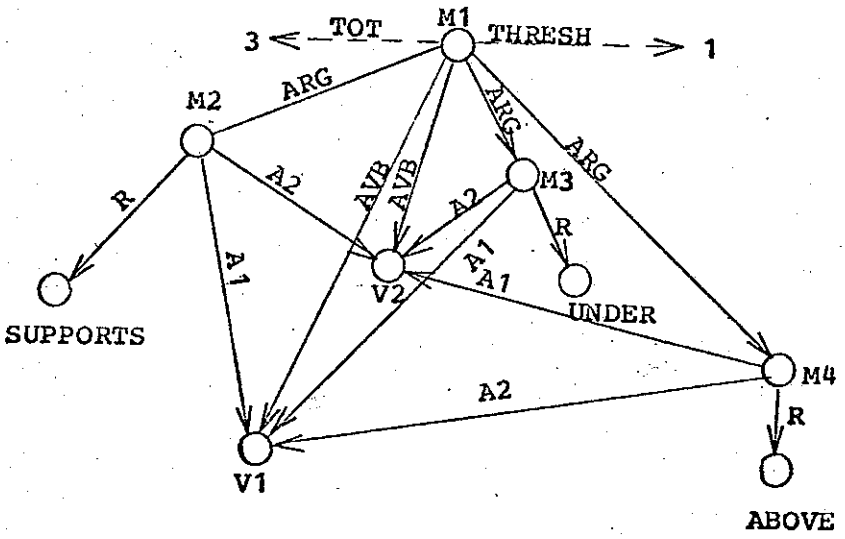
We may write this rule symbolically as:

$$\forall x,y \; {}_{3}\theta_{1} \quad (\text{Supports}(x,y), \text{Under}(x,y), \text{Above}(y,x))$$

in the SNEPS input language [11], this is written as:

```
(BUILD AVB($X $Y) TOT 3 THRESH 1
    ARG((BUILD A1 *X R SUPPORTS A2 *Y)
        (BUILD A1 *X R UNDER A2 *Y)
        (BUILD A1 *Y R ABOVE A2 *X)))
```

and as a semantic network, we draw it as:



If we want to deduce a node that matches M2,M3, or M4, we can use M1 as a consequent theorem [6], and the other two nodes become antecedents. If a node matching M2,M3, or M4 is asserted, we can use M1 as an antecedent theorem and the other two nodes become consequents.

When an argument of a deduction rule is matched, processes are created to carry out the indicated inference. Some processes analyze the deduction rule and create other processes that are specialized for using the deduction rule in the proper direction

and with the proper sets of antecedents and consequents. For example, if node M2 were matched during a backward chaining operation, processes would be created for using rule M1 as a consequent theorem with M3 and M4 as antecedents, either of which is sufficient for deducing M2. Once created, these processes can be saved so that if the same rule is needed again to deduce the same consequent, the processes need not be recreated. Processes may be assigned priorities and resource bounds. They may be executed in parallel (or simulated parallel) subject to differences in priorities. When a process expends its resources, it is suspended until it is assigned additional resources.

Every process has a name which defines the action the process will perform and a continuation link to the process that is to be scheduled for activation after it has completed its job. Each process also has other "slots" or "registers" peculiar to the action it will perform. Processes pass information back along their continuation links by scheduling instances of the "messenger" process, ANS, which inserts its message in the MSG register of the receiving process and then schedules that process.

Two kinds of processes control the use of deduction rules used in a backward-chaining (consequent) manner. The process USE controls the top level of a deduction rule, while the process USE-1 controls embedded rules. Both processes have registers for the rule (RULE), the consequent (CQ) and a binding of the variables used in the rule (BNDG). For example,

suppose the rule

$\forall x,y(x \text{ ON } y \rightarrow \forall z(y \text{ ON } z \rightarrow x \text{ ON } z))$

were to be used to deduce answers to the question, (A ON ?). A USE process would be created whose registers would be*

RULE: $\forall x,y(x \text{ ON } y \rightarrow \forall z(y \text{ ON } z \rightarrow x \text{ ON } z))$

CQ: $\forall z(y \text{ ON } z \rightarrow x \text{ ON } z)$

BNDG: ((x.A)(z.?))

Above it on a path of continuation links would be the USE-1 process with registers

RULE: $\forall z(y \text{ ON } z \rightarrow x \text{ ON } z)$

CQ: x ON z

BNDG: ((x.A)(z.?))

wnen the USE process receives a message informing it that (A ON B) is valid, it would create and schedule a specialized USE-1 process with registers

RULE: $\forall z(y \text{ ON } z \rightarrow x \text{ ON } z)$

CQ: x ON z

BNDG: ((x.A)(y.B)(z.?))

This process would attempt to answer the question (B ON ?).

Both USE and USE-1 processes nave continuation links to processes with the name ANS-CATCH. This process has three registers: MSG, DATA, and BOSSES. The contents of BOSSES is a list of processes. Whenever ANS-CATCH is activated, it takes its messages (from MSG), and whichever ones are not already in DATA are added to DATA and sent to all the BOSSES. When a process wants to use a deduction rule to deduce some consequent,

---

*Actually the RULE and CQ registers would contain nodes, not symbolic expressions.

it first checks if a USE or USE-1 process already exists to use that rule for that consequent with a binding compatible with its own. If one is found, the process adds itself to the list of BOSSES in the ANS-CATCH above the USE or USE-1 and immediately takes all the answers in the DATA register of the ANS-CATCH. In this way, if a deduction rule is useful in several places in a deduction, duplicate work is avoided. In the case of recursive rules, like those for transitive relations, the result is a cycle of continuation links - an ANS-CATCH among whose BOSSES is a process with a path of continuation links to the ANS-CATCH itself. Answers will circulate in this cycle of processes, moving one link in the chain of transitive relations with each cycle, until no more answers can be produced. The ANS-CATCH process can be viewed as a specialized data base connected to rules which can be used antecedently whenever an assertion is added to its DATA register. Although these rules are pattern-directed, they are guaranteed to match any assertion that gets added to the ANS-CATCH.

References

1. Bechtel, R.J. Logic for semantic networks, M.S. Thesis. Technical Report No. 53, Computer Science Department, Indiana University, Bloomington, IN., July, 1976.

2. Bechtel, R.J.,and Shapiro, S.C. A logic for semantic networks. Technical Report No. 47, Computer Science Department, Indiana University, Bloomington, IN., March, 1976.

3. Greif, I., and Hewitt, C. Actor semantics of PLANNER-73, Proc. 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, 1975.

4. Hewitt, C.; Bishop, P.; and Steiger, R. A universal modular ACTOR formalism for artificial intelligence. Proc. IJCAI 3, Stanford, CA., Aug., 1973, 235-245.

5. Hewitt, C., et al. Actor induction and meta-evaluation. Proc. 1st ACM Symp. on Principles of Programming Languages, Boston, 1973, 153-168.

6. Hewitt, C. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. AI-TR-258. M.I.T., A.I. Lab., 1972.

7. Kaplan, R.M. A multi-processing approach to natural language, Proc. NCC, 1973, 435-440.

8. Papert, S.A. Teaching children to be mathematicians versus teaching about mathematics. Int. J. Math. Educ. Sci. Technol. 3 (1972), 249-262.

9. Shapiro, S.C. The MIND System: a data structure for semantic information processing. R-837-PR, The Rand Corporation, Santa Monica, California, August, 1971.

10. Shapiro, S.C. A net structure for semantic information storage, deduction and retrieval. Proc. Second Int. Joint Conference on Artificial Ingelligence, The British Computer Society, London, England, September, 1971, 512-523.

11. Shapiro, S.C. An introduction to SNePS. Technical Report No. 31, Computer Science Department, Indiana University, Bloomington, IN., Revised December, 1976.

12. Shapiro, S.C. Representing and locating deduction rules in a semantic network. To be presented at the Workshop on Pattern-Directed Inference Systems, U. of Hawaii, May 23-27, 1977.

13. Shapiro, S.C. and Bechtel, R.J. Non-standard connectives and quantifiers for question-answering systems, in progress.

14. Wand, M. The frame model of computation. Technical Report No. 20, Computer Science Department, Indiana University, Bloomington, IN., December, 1974.

NOTES

PROPER ROLE FOR RESOLUTION THEOREM PROVERS

Vesko G. Marinov
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

There is a widespread belief at the present time that resolution cannot be very helpful in automatic theorem proving. This is a consequence of the disappointing experience with previous use, particularly in view of the high hopes which it had originally raised. It is the author's opinion that this is an overreaction which possibly prevents researchers from applying resolution to tasks where its application is in fact appropriate. Such applications are naturally rather modest compared to what was at one time hoped for resolution. It is an established fact now that, despite the wide variety of restrictions and strategies developed for resolution, it remains too weak for proving theorems of any interest in mathematics. However, in places where simple but tedious logical computations involving predicate calculus formulas are needed resolution seems to be the best answer.

This claim will be supported with examples drawn from the experience with a system for teaching axiomatic set theory developed at the Institute for Mathematical Studies in the Social Sciences at Stanford and used for regular course instruction since the fall of 1974. The most essential element of the system is a proof checker based on the logical system of P. Suppes, used for determining the correctness of the students' proofs. The most powerful and most frequently used rules of inference utilize a resolution theorem prover. To the best of our knowledge this is the only resolution theorem prover, in fact probably the only general purpose theorem

prover, used in actual production.

The experience with the CAI system shows that a well-organized resolution theorem prover gets most of the inferences seen intuitively by the user while working on a proof. The user has no interaction with the theorem prover except for asking that a formula be verified and supplying the references, which he thinks the formula follows from. The prover is used primarily for the rules VERIFY and CONTRADICTION. While using VERIFY the user has to type the line, whose negation together with the references is passed to the prover. If the prover is able to confirm the inference it signals the proof checker to accept the line. For the CONTRADICTION rule the user merely points to the references which he believes form an inconsistency. If such is detected by the prover, the proof checker returns the negation of the last assumption on which the references depend. (Presumably there must be an incorrect assumption in order to reach a contradiction.)

The main reason for selecting a resolution theorem prover was our belief that for the same generality and the same power it can be designed in a much more compact way than a heuristic theorem prover. For the purposes we are using it, simply a mechanical tool is needed and resolution seems to be exactly that. The prover was written in UCI-LISP. Together with the converter of the formulas into clausal form it is about 10 pages of pretty printed code.

One thing that has plagued work on resolution in the past has been preoccupation with completeness. Recognizing that a prover is working in an undecidable domain it is obvious that completeness is going to be restricted by the real factors of time and space. The main objective in choosing a strategy and tuning a prover's parameters

is optimizing the number of inferences it gets. It is the author's conviction that in this context incompleteness is a feature, rather than a drawback. Thus, completeness in the prover in question is restricted severely in many different ways.

The prover employs the MU strategy. It consists mainly of keeping only resolvents containing merge literals or having a unit parent. It has been shown that if in a refutation there are resolvents not satisfying the above restriction, there always exists another refutation (from the same input set) where such resolvents are obtained first. With this in view the strategy occasionally allows for a round of general resolution after which the restriction is imposed. Experiments with different strategies for resolution, carried out earlier by the author at the University of Texas have shown the MU strategy to be quite efficient in the set-theoretical domain. One property of the MU strategy, coupled with a limit on the depth of functional nesting in the resolvents, is that it usually runs quickly out of possibilities to resolve when given a satisfiable set of clauses (i.e. insufficient references). This is very important in CAI applications because one very frequent error of the student users has been to supply insufficient or incorrect references. In such a case it is very desirable that the prover detects this fact as soon as possible, rather than grind until the time limit is reached. This property has strongly influenced the selection of the MU strategy.

Equality plays a very important role in just about any mathematical theory, including set theory. Consequently it gets special treatment in the prover. Like the resolution strategy, the equality replacement

is very restricted. First of all, only demodulation is performed, i.e., only less complex terms are substituted for more complex. (The measure for complexity is the depth of functional nesting). Second, replacements are done only on the basis of unit equality clauses. An important kludge is that if the input set contains a ground unit equality clause, one of the terms is substituted uniformly for the other throughout the set and the clause is dropped. Nevertheless, treatment of equality remains the weakest point in the prover and the majority of inferences seen by the users and missed by the prover involve equality. A large quantity of theoretical results on equality has been accumulated lately, but little has been drawn from practical experience. There is a need of implementation oriented strategies for equality based on the above-mentioned principle of optimizing the number of inferences within the prover's reach.

Probably the most frustrating property of the prover from a CAI point of view has been the fact that thus far it has been impossible to characterize the class of theorems accepted by the prover despite substantial effort on the part of members of the IMSSS staff. This is most likely a consequence of the unnatural way in which resolution works. Sometimes the prover is able to verify steps much larger than the user can see, while other times it fails at steps which the user expects to be accepted. It would have been very convenient if one could give the users some more accurate idea what to expect from the prover.

Inferences which are missed by the prover, while being obvious to the user, are largely due to the fact that resolution breaks down

the formulas to the atomic level before it can find a proof. For example, the inference

$$(\forall\, m,n,p,A,B)(A \cap B = 0 \wedge K(A) = m \wedge K(B) = n$$
$$\wedge K(A \cup B) = p \rightarrow m + n = p)$$

from

$$(\forall\, m,n,p)(\exists\, A,B)(A \cap B = 0 \wedge K(A) = m \wedge K(B) = n$$
$$\wedge K(A \cup B) = p) \leftrightarrow m + n = p)$$

cannot be verified by the prover. The human user, though, with his well-developed abstraction capabilities quickly sees that the left side of the implication can be looked at as $\phi(m,n,p,A,B)$. Hence the inference becomes:

$$(\forall\, m,n,p,A,B)(\phi(m,n,p,A,B) \rightarrow m + n = p)$$

from

$$(\forall\, m,n,p)((\exists A,B)\phi(m,n,p,A,B) \leftrightarrow m + n = p)\ .$$

Of course, seen this way, the inference is immediate for the prover, too. There is a provision in the proof checker for the user to pass to the prover the abstracted form of the formulas. What would be desirable here is to have a heuristic coupler between the prover and the proof checker which looks at the possibilities of abstraction in the set of formulas.

NOTES

ABSTRACT

A Proclarative Approach to Problem Solving

by

Daniel H. Fishman
Bell Telephone Laboratories

The thesis presented here is that there is no conflict between the "procedural" and "declarative" approaches to inference and problem solving. Indeed, rather than being in conflict, these approaches are complementary. They should be brought together where they can function in a common framework. In this paper, we make a case for embedding a declarative-type deductive system into a procedural language for problem solving. The resulting proclarative system will be significantly more powerful than either the pure procedural or declarative systems by themselves. Such a merger would have the effect of making it easier to construct problem-solving systems, while simultaneously making the resulting systems more effective.

It is widely agreed that deductive methods are important tools for problem solving. Rudimentary methods for automatic deduction have been incorporated into the languages developed for problem solving. In contrast to the approach taken in a language like PLANNER, we believe that the use of deduction should be explicit and amenable to control rather than implicit and submerged in the machinery of the language. Easy access to the mechanisms controlling the deductive processes will allow its user to guide and limit the search as necessary. Furthermore, we believe that the syntactic representation of declarative knowledge admits more effective control and communication between active "problem-solving methods" than does procedural representation. That is, in a declarative system, the frontier of the search tree is available for examination at each search step permitting the automatic selection of the "best" next subproblem to attack. Although the analog exists in the procedural setting, it is quite unwieldy to control. In a language like CONNIVER which permits such control, the programmer must provide it himself at every point where a complex search is invoked. What is most likely is that the programmer will use the default search control mechanisms provided by the language, resulting in the most inefficient search. In PLANNER and CONNIVER this results in a depth-first search.

Currently, the capabilities of deductive systems may be brought to bear only on problems in a fixed and static data base. On the other hand, in problem solving, one is interested in exploring arbitrary numbers of data bases (contexts, world models, etc.), in search of a world model in which some goal condition is satisfied. We believe that it is useful to view

problem solving as involving two separate search problems. One of these is a low-level deductive search of a data base representing a world model. For such searches, a primarily declarative approach seems best suited. The other search problem involves the high-level exploration of a space of world models. For such searches, a primarily procedural approach seems best suited. The use of a deductive system incorporating a powerful search procedure to oversee the interaction of "problem-solving methods" in controlling the low-level search, can free its user from the intimate control of this search and allow him to concentrate on the high-level search of alternative world models.

Deductive systems (theorem provers) have advanced well beyond the use of purely syntactic inference rules and blind search strategies. Methods exist to employ semantic information (semantic nets, types, counts, advice, etc.) to restrict inferences and control searches. Flexible strategies exist for subproblem selection and avoidance of redundant inferences. Systems have been developed (though are largely untested) for dealing with large data bases. Inference rules exist for dealing with sets of objects. All of these tools may brought to bear on a given problem automatically. Thus, if one could rely on such facilities being invoked as a default, rather than the simple depth-first search provided as a default by current problem solving languages, considerable progress will have been achieved.

The proclarative approach to problem solving would possess the following features:

- A powerful deductive system with many features such as those noted above. This system would be used to perform deductive searches of a given data base of assertions, and of general rules which are analogous to THCONSE's and IF-NEEDED's.

- A host language containing many of the features of a problem-solving language such as CONNIVER. In particular, it would include a hierarchical data base facility in which to represent and dynamically explore a model search space. It would also include the ability to add and delete nodes in the hierarchy and to add, delete, and modify elements of any node.

- Any model, represented by the contents of nodes in a path through the hierarchy, would be accessible to the deductive system as a data base for a given request. Thus, a request would specify a path representing the data base, and the problem (or question) to be solved (or answered). It would also specify any restrictions which should be employed, e.g., it could require that the search be restricted to assertions only, or that it be incremental, perhaps using assertions first and general rules later, if the search is restarted.

- The control structure of the deductive system would be accessible to the programmer, giving him the option of allowing independent control, or of providing an appropriate measure of control himself.

- The ability would also be provided to suspend a search after some increment of search effort, to efficiently save its state, and to resume or delete a suspended search as desired.

Implicit in this paper is a rebuttal to the question of whether "declarative" methods or "procedural" methods are better. Rather, we believe that the question that should be asked is how can we combine these approaches to achieve the best possible results. We believe that a proclarative approach to inference and problem solving such as the one described above offers the best of both possible worlds.

NOTES

Deduction in the Pejorative Sense

Drew McDermott Yale

The word "deduction" has many ranges of meaning in ordinary language. Sherlock Holmes used it to mean "reasoning generally." Traditionally, it has meant logically necessary reasoning, contrasted with induction. In AI research, for most of the last decade, it has often referred to a representational scheme and class of algorithms ("theorem provers") so abhorrent as to be avoided at all costs.

In this paper I will argue that this form of deduction is with us to stay, not as the main control structure required for intelligence, but as an inevitable component of the information-retrieval systems intelligent programs need in order to be flexible. I will report on the results of using a theorem prover as such a component of a problem solver, and on some mini-experiments with other novel uses.

There are good reasons for the dislike of theorem proving by AI researchers, but many people who dislike it do so for bad reasons, or just because of fashion.

All theorem provers operate by applying the "resolution" rule: From P′ and P->Q, infer Q′, where P matches P′ and Q′ is the result of substituting into Q the variables bindings obtained from the match. Often this rule is used in a "backward" way: from ¬Q′ and P->Q, infer ¬P′. Given a set of axioms and a goal G, most theorem provers form the skolemized negation ¬G of what they are trying to prove and apply "backward" resolution until a

goal ¬P finds a P′ which matches it in the axiom set.

In order to handle conjunctive goals, we must generalize the rules to be

$$\frac{\neg(Q' \wedge C) \qquad P \rightarrow Q}{\neg(P' \wedge C')}$$

and

$$\frac{\neg(P' \wedge C) \qquad P}{\neg C'}$$

which say, roughly, solve one conjunct, then go on to the others.

All "theorem provers" (and AI languages with "pattern-directed procedure invocation") work this way, chaining backwards from goals through implications to facts. Unfortunately, the resolution literature is perversely obscure about such matters. Once they are made clear, it can be seen that some criticisms that are often made, that resolution isn't goal-directed, or that looking for a refutation is unnatural, are misguided.

The real problem with theorem provers is also the problem with pure AI languages, semantic networks, and large data bases. This is the problem of handling conjunctive goals. The rules I gave above amount to an implementation of a "generate and test" strategy. Solutions (variable bindings) which satisfy P′ may not satisfy C at all. For example, consider the goal ¬((SOBER ?X) ∧ (IRISH ?X)). The rule as given entitles us to generates SOBER persons, and detach ¬(IRISH ...) for each of them. This could create an arbitrarily large number of goals with (in this case) a very small return.

There is nothing wrong with "generate and test." In cases like this it is the only possible strategy. But for doing problem solving or symbolic computation, it is a disaster. (Green, 1969)

In my opinion, criticizing all other problems with deduction is a waste of time, since the compromises necessary to correct them have been accepted by almost everyone. The result of these compromises is an AI language like PLANNER (Hewitt, 1972). For example, the deductive systems cannot conclude a fact, even provisionally, from an inability to prove it false. PLANNER, and any other AI language, can do it with something like the THNOT operator.

In spite of the weakness of deduction in the pejorative sense, it is indispensable, for these reasons:

> It treats quantifiers and other complexities in a pleasing manner.

> The "backward chaining" paradigm is useful in other contexts besides straightforward deduction. "Abduction" and "matching" (in the GPS sense) may be thought of as "deduction with gaps."

> Unlike its competitors, deduction is able to accept new information (facts and implications) easily.

> It is easy to keep track of deductive operations; conclusions can be tagged with their proofs, and these "data dependencies" can be used for data base debugging and other purposes. (Stallman and Sussman, 1976)

> Possible explosions from moronic generate-and-test situations can be caught and returned to the problem-solving monitor or human user for help.

I have tested these ideas in a couple of ways: first, by implementing a problem-solving system which accesses all its information through a theorem prover (McDermott, 1976); and second, with some smaller-scale experiments in using backward chaining for abduction (Pople, 1973) and induction. (Winston, 1975)

References

Green, C. Cordell (1969) Theorem-proving by resolution as a basis for question-answering systems. In Meltzer, Bernard, and Michie, Donald (1972) (eds.) Machine Intelligence 7.

Hewitt, Carl (1972) Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. Cambridge: MIT AI Lab Techical Report 258.

McDermott, Drew V. (1976) Flexibility and efficiency in a computer program for designing circuits. Unpublished Ph.D. thesis, MIT.

Pople, Harry (1973) On the mechanization of abductive logic. Proc. IJCAI 3.

Stallman, Richard M. and Sussman, Gerald J. (1976) Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Cambridge: MIT AI Lab Memo 380.

Winston, Patrick (1975) Learning structural descriptions from examples. In Winston, Patrick (1975) The Psychology of Computer Vision. New York: McGraw-Hill Book Company.

NOTES

Informatics Laboratory
Linkoeping University
Erik Sandewall, lew

April 18, 1977

Predicate calculus as a blueprint for programs

My view of the appropriate usage of predicate calculus for knowledge-based program systems (for example Q.A. Systems) can be summarized as follows:

- some parts of such systems can be well expressed in P.C., others can not.

- a specification in P.C. of the appropriate parts of such a system is (or at least, should be) a very high level representation. Its reductic to actual execution in the computer, through an interpreter such as a theorem prover or through a compiler, is a complex and delicate operati

- at least for the time being, it is therefore best to view the P.C. repr sentation as a specification for a program, which is then transferred manually into an actual program. This mode of operation is analogous to what is done in programming applications involving numerical computatic where a mathematical formulation of the problem serves as the blueprint for the program. It provides the necessary structure to the programming process and the program itself, and is also an aid to documenting the program and explaining its performance.

I propose to talk partly about this approach and partly about my experienc when it was used for a collection of axioms expressing a number of common natural-language constructs. The actual program reflects the given axioms, and can easily be understood if the axioms are known, but in writing the program it was also natural to introduce a number of implementation decisions to enhance efficiency. An analysis of those decisions provides guidelines both for continued use of the approach, and for possible future axiom compilers which efficiently convert axioms into corresponding, efficient code. One observation is that such compilers should analyze the given set of axioms before they proceed to compile the individual axioms. This means also that the presently fashionable approach to view predicate calculus as a programming language, to be interpreted by a theorem-prover is computationally wasteful.

NOTES

A FIRST ORDER THEORY OF DATA AND PROGRAMS

Keith L. Clark and Sten-Åke Tärnlund

Department of Computing and Control
Imperial College of Science and Technology
London University

Depatment of Computer Science
University of Stockholm
Sweden

Abstract.

In this paper we propose first order predicate logic as a formalism for data specification, program writing and program verification.

It has been proposed (Hoare [6], Liskov et.al [9]) that programming languages should have data definition facilities which allow the programmer to characterize his data by its essential properties. In logic the facility is axiomatic definition. We demonstrate this in section 2 by giving axiomatic definitions of several fundamental data structures.

The relation that always holds between the input and output structures of some algorithm partly characterises the algorithm. In section 3 we give axiomatic definitions of the input-output relation of two fundamental algorithms for the data structures we have defined. From these we derive a set of Horn clauses. Under the procedural interpretation of logic (Hayes [4], Kowalski [8]) this set of clauses becomes a logic program for the algorithm when we add suitable control information. In fact for the logic programs we derive an ordering of the literals (procedural calls) in the body of each clause is the only control required. With a suitable ordering we get efficient computational behaviour from a top-down theorem prover for Horn clauses (Kowalski [8]). The programs can be run interpretively on the Marseille PROLOG (Roussel [11]) or compiled and executed as DEC 10 machine code by a version of PROLOG implemented at Edinburgh (Warren [12]).

If possible programs should be verified. In section 4 we show how the axioms that define the data and the input-output relation of the program can be used to verify the program. We use induction schemas to prove general theorems about input-output relation which quarantee termination and correctness of the program. In effect these are structural induction proofs (Burstall [2]) within a first order theory (cf. Boyer and Moore [1]). Each induction schema is derived from the axiomatic characterization of the data.

The whole sequence, the data axiomatization, the input-output axiomatization and the verification can be thought of as steps in the development of a first order theory of data and programs. Indeed our axiomatizations and proofs are analogous to those of a first order theory of arithmetic (Mendelson [10]). However, such a theory not only provides us with a means of clarifying concepts, it also provides us with a methodology for writing and verifying programs.

NOTES

USING LEMMAS IN AN AUTOMATIC THEOREM PROVER FOR RECURSIVE FUNCTION THEORY

by

Robert S. Boyer

and

J Strother Moore

Computer Science Laboratory
Stanford Research Institute
Menlo Park, Ca. 94025

The research reported here has been supported by the Office of Naval Research under Contract N00014-75-C-0816, the National Science Foundation under Grant DCR72-03737A01, and the Air Force Office of Scientific Research under Contract F44620-73-C-0068.

We have implemented a computer program that proves theorems about recursive functions defined on finitely representable objects. We describe how that program uses lemmas. The new work significantly extends our earlier work [1], [2], [3], [4] because it is not limited to primitive recursion or the world of binary trees. This work is closely related to (but independent of) the recent work of Aubin [5] and Cartwright [6].

A SKETCH OF THE THEORY

The mathematical theory in which our theorem prover operates is a free-variable equational calculus for recursive functions in the spirit of Skolem [7] and Goodstein [8]. The theory admits the introduction of arbitrary total recursive functions. The domain of objects is partitioned into an infinity of disjoint "type classes."

The user is free to define axiomatically the properties of any class. The only initial objects are TRUE and FALSE. The only initial functions are IF and EQUAL. (IF x y z) is z if x is FALSE and is y otherwise. (EQUAL u v) is TRUE if u is v and FALSE otherwise.

Integers, literal atoms, pairs, push-down stacks, characters, and strings of characters are examples of classes that one may axiomatize. There is a facility for succinctly axiomatizing new classes of objects with properties virtually identical to Burstall's structures [9] and close to the style of Clark and Tarnlund [10].

The theory also contains a version of the principle of induction called the Generalized Principle of Induction (Noetherian induction) in Burstall [9]. The principle allows one to induct over any well-founded partial ordering. In our implementation of it, one can induct on any well-founded partial ordering that can be induced by mapping objects (or n-tuples of objects) into the ordinals.

HOW LEMMAS ARE USED

The new system uses lemmas in many ways; we will explain four of the most common uses. The lowest level use of lemmas and axioms in our system is in the routine called TYPE.SET, which takes an expression and returns a set of primitive type classes, one of which contains the value of the expression. TYPE.SET works by recursively exploring the expression, unioning the type sets of both outputs of IF's, using axioms to determine the types of axiomatically defined functions and inductively computing the type sets of defined functions (unless lemmas are available that specify the possible types).

The second use of lemmas is as rewrite rules during simplification. Of course, the simplifier uses function definitions as rewrite rules to "open up" a function call by replacing it with its definition (when the result is simpler in some sense). The simplifier also uses TYPE.SET, for example, to rewrite (EQUAL u v) to FALSE if the types of u and v do not intersect. But more generally, the theorem prover interprets all axioms and lemmas as rewrite rules in the following way.

Given any formula (axiom or lemma) consider all of the sequents

H1 & H2 & ... & Hn -> C

one can deduce from it by propositional calculus. We classify each

sequent according to the form of C. If C is of the form (NOT u) then the rule can be used to rewrite any term which unifies with u to FALSE, provided the instantiated Hi can be established. If C is of the form (EQUAL u v) it can be used to rewrite u to v, again provided the instantiated Hi can be established. Otherwise, if C is just u, where u is Boolean (i.e. TYPE.SET(u) contains nothing but TRUE and FALSE) then it can be used to rewrite u to TRUE, under the same provision.

When the simplifier has decided to use a given rewrite it tries to establish the Hi by (recursively) simplifying them (to non-FALSE).

Care is taken to avoid infinite regression (e.g., repeated applications of a commutivity rewrite or "pumping" up a term with a rewrite like (P (F x)) -> (P x)). For example, we avoid the first problem by refusing to apply a rewrite of the form (EQUAL u v) when u and v are variants unless the result is lexicographically less than the original formula. Thus, the lemmas

```
(PLUS i j) = (PLUS j i)
```

and

```
(PLUS i (PLUS j k)) = (PLUS j (PLUS i k)),
```

with the lexicographic restriction, cause nested plus expressions to be right associated with their arguments in (lexicographically) ascending order.

A third use of lemmas arises when the system generalizes a conjecture to be proved. As in our earlier LISP theorem prover [1], the system generalizes subterms common to both sides of an equality or implication. Unlike the LISP theorem prover, the new system is sensitive to previously proved facts about the subterm. Thus, if it had proved that (SORT x) is a list of numbers when x is a list of numbers and it generalizes (SORT x) in:

```
(LIST.OF.NUMBERS x) & (ORDERED (SORT x)) & (NUMBERP i)
->
(ORDERED (MERGE I (SORT x)))
```

it produces:

```
(LIST.OF.NUMBERS x) & (ORDERED z) & (NUMBERP i)
                    & (LIST.OF.NUMBERS z)
->
(ORDERED (MERGE i z)).
```

This use of lemmas allows the new system to avoid one of the most common failure modes of our earlier LISP theorem prover: generalizing a conjecture too much.

A fourth way lemmas can be used is in the establishment of an induction principle. For example, the lemma:

i < max -> max-(ADD1 i) < max-i

informs the system that it is sound to induct up by ADD1 to a maximum (e.g., that to prove P(x, y) inductively one may prove it when ~(x < y) and prove it when (x < y) assuming P(x+1, y)).

As observed in Boyer and Moore [1], what makes an induction principle appropriate for a conjecture is whether it supplies inductive hypotheses about the recursive calls introduced when some chosen set of functions in the induction conclusion are opened up. Thus, whenever a new function is introduced, the system analyzes it and tries to construct one or more inductive principles from its known lemmas that are appropriate for the new function. At prove time, it goes over all the possible principles suggested by functions in the conjecture, combines them when they are similar (an induction down (CDR x) is merged into one down (CDDR x)) or when they compete for the same variable (an induction on x and y would merge with one on y and z to become an induction on x, y, and z), and finally selects the most appropriate induction on the basis of which one satisfies the largest number of recursions.

The analysis at definition time (to create induction principles that supply hypotheses about the recursive calls) is the crucial step involving lemmas. Using its lemmas, the system tries to find some combination of measures on subsets of the arguments that would decrease on each recursive call. It can use the principle of lexicographic ordering to combine measures to explain the recursion of functions like

Ackermann's function (where a measure of one argument goes down or stays constant while another measure on another argument goes down whenever the first measure of the first argument stays constant).

By finding such a measure and inventing the necessary conditions from the lemmas, the system can obtain a sound induction hypothesis about each n-tuple of values of variables changed in recursive calls (regardless of what the function actually tests before recursing or whether all of the variables are involved in the measure).

REFERENCES

1. R. Boyer and J Strother Moore, "Proving Theorems about LISP Functions," JACM, Vol. 22, No. 1, pp. 129-144 (1975).

2. J Strother Moore, "Computational Logic: Structure Sharing and Proof of Program Properties," Ph.D. thesis, University of Edinburgh (1973).

3. J Strother Moore, "Automatic Proof of the Correctness of a Binary Addition Algorithm," SIGART Newsletter, No. 52, pp.13-14 (1975).

4. J Strother Moore, "Introducing Iteration into the Pure LISP Theorem Prover," IEEE Trans. Soft. Eng., Vol. 1, No. 3, pp. 328-338 (1975).

5. R. Aubin, "Mechanizing Structural Induction," Ph.D. Thesis, University of Edinburgh, 1976.

6. R. Cartwright, forthcoming Ph.D thesis, Computer Science Department, Stanford University, Stanford, California(1977).

7. T. Skolem, "The Foundations of Elementary Arithmetic Established by Means of the Recursive Mode of Thought, without the Use of Apparent Variables Ranging over Infinite Domains," in From Frege to Goedel, (ed. Jean van Heijenoort) Harvard University Press, Cambridge, Mass., pp. 305-333 (1967).

8. R. L. Goodstein, Recursive Number Theory, North-Holland Publishing Co., Amsterdam, (1957).

9. R. M. Burstall, "Proving Properties of Programs by Structural Induction," The Computer Journal, Vol. 12, No. 1, pp. 41-48 (1969).

10. K. Clark and S-A. Tarnlund, "A First Order Theory of Data and Programs," Department of Computing and Control, Imperial College, London (1976).

NOTES

Jan. 77

THE AUTOMATIC SYNTHESIS OF RECURSIVE PROGRAMS

ZOHAR MANNA
Artificial Intelligence Lab
Stanford University
Stanford, Ca.

RICHARD WALDINGER
Artificial Intelligence Center
Stanford Research Institute
Menlo Park, Ca.

Abstract

We describe a deductive technique for the automatic construction of recursive programs to meet given input-output specifications. These specifications express what conditions the output of the desired program is expected to satisfy. The deductive technique involves transforming the specifications by a collection of rules, summoned by pattern-directed function invocation. Some of these transformation rules express the semantics of the subject domain; others represent more general programming techniques. The rules that introduce conditional expressions and recursive calls into the program are discussed in some detail.

The deductive techniques described are embedded in a running system called SYNSYS. This system accepts specifications expressed in high-level descriptive language and attempts to transform them into a corresponding LISP program. The transformation rules are expressed in the QLISP programming language. The synthesis of two programs performed by the system are presented.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, by the National Science Foundation under Grant DCR72-03737 A01, by the Office of Naval Research under Contracts N00014-76-C-0687 and N00014-75-C-0816; and by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, Stanford Research Institute, or the U.S. Government.

NOTES

AN ALGORITHM FOR REASONING ABOUT EQUALITY

by

Robert E. Shostak
Stanford Research Institute
Menlo Park, California 94025

1. Introduction

To be useful for program verification a deductive system must be able to reason proficiently about equality. Important as its semantics are, equality is often handled in an ad hoc and incomplete way--most usually with a rewrite rule that substitutes equals for equals with some heuristic guidance. This article presents a simple algorithm for reasoning about equality that is fast, complete (for ground formulas with function symbols and equality), and useful in a variety of theorem-proving situations. A proof of the theorem on which the algorithm is based is given as well.

NOTES

DECISION PROCEDURES FOR SIMPLE EQUATIONAL THEORIES WITH PERMUTATIVE EQUATIONS: COMPLETE SETS OF PERMUTATIVE REDUCTIONS

by D. S. Lankford and A. M. Ballantyne

SUMMARY

Familiarity with concepts related to complete sets of reductions is assumed, see, Knuth and Bendix (1), Lankford (2), and Slagle (3). The primary disadvantage with complete sets of reductions is that certain axioms, like commutative axioms, cannot be included in complete sets of reductions. In this article we seek to overcome this difficulty by using finite equivalence class methods.

The basic idea is the following: to determine if $t = u$ is a consequence of a set $\mathcal{E}$ of equations, it suffices to determine if t and u are in the same $\approx_{\mathcal{E}}$ equivalence class, where $t \approx_{\mathcal{E}} u$ iff $t = u$ is a consequence of $\mathcal{E}$. The difficulty with this approach is that there is no algorithm to decide if t and u are in the same $\approx_{\mathcal{E}}$ equivalence class. However, if the equations of $\mathcal{E}$ are such that all $\approx_{\mathcal{E}}$ equivalence classes are finite, then in principle $t = u$ can be decided. Since we do not know of an algorithm which, given a set $\mathcal{E}$ of equations, decides whether all $\approx_{\mathcal{E}}$ equivalence classes are finite, we consider a subclass of the general problem. Let $n(x,Y)$ be the number of occurrences of the symbol x in the

basis of theorem provers for equality. In addition, we believe that many common decidable equational theories can be decided with complete sets of permutative reductions. For example,

R1. $\{x \cdot 1 , 1 \cdot x\} \longrightarrow \{x\}$,

R2. $\{x \cdot (x^{-1}) , (x^{-1}) \cdot x\} \longrightarrow \{1\}$,

R3. $\{1^{-1}\} \longrightarrow \{1\}$,

R4. $\{(x^{-1})^{-1}\} \longrightarrow \{x\}$, and

R5. $\{(x \cdot y)^{-1} , (y \cdot x)^{-1}\} \longrightarrow \{(x^{-1}) \cdot (y^{-1}) , (y^{-1}) \cdot (x^{-1})\}$

form a complete set of permutative reductions relative to

$$\mathcal{P} = \{x \cdot y = y \cdot x , (x \cdot y) \cdot z = x \cdot (y \cdot z)\} ,$$

which decides equational Abelian group theory.

Let us illustrate permutative reduction by showing how $(1 \cdot x) \cdot ((y \cdot (x^{-1})) \cdot z) = (w \cdot z) \cdot (y \cdot (w^{-1}))$ is proved. Form $\approx_{\mathcal{P}}((1 \cdot x) \cdot ((y \cdot (x^{-1})) \cdot z))$ and $\approx_{\mathcal{P}}((w \cdot z) \cdot (y \cdot (w^{-1})))$ and permutatively reduce them as far as possible:

$$\approx_{\mathcal{P}}((1 \cdot x) \cdot ((y \cdot (x^{-1})) \cdot z)) \longrightarrow \approx_{\mathcal{P}}(x \cdot (y \cdot (x^{-1})) \cdot z))$$
$$\longrightarrow \approx_{\mathcal{P}}(y \cdot z) = \{y \cdot z , z \cdot y\} , \text{ and}$$

$$\approx_{\mathcal{P}}((w \cdot z) \cdot (y \cdot (w^{-1}))) \longrightarrow \approx_{\mathcal{P}}(z \cdot y) = \{z \cdot y , y \cdot z\} .$$

Since equivalence classes are either equal or disjoint, we only need check if a member of one occurs in the other. Thus, the given theorem is proved.

term Y. We say $t = u$ is a permutative equation iff $n(x,t) = n(x,u)$ for each symbol x. For example, $x \cdot y = y \cdot x$ and $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ are permutative equations. If ρ is a finite set of permutative equations, then all $\approx_\rho$ equivalence classes are finite.

Equations $t = u$ which are not permutative equations are treated as permutative rewrite rules $\approx_\rho(t) \longrightarrow \approx_\rho(u)$ or $\approx_\rho(u) \longrightarrow \approx_\rho(t)$, where $\approx_\rho(t)$ is the $\approx_\rho$ equivalence class of t. Notions of immediate permutative reduction, finite termination property, unique termination property, and complete sets of permutative reductions are defined in the obvious way, based on the corresponding concepts in Knuth and Bendix (1) and Lankford (2).

This article develops two mathematical characterizations of the unique termination property for finite sets of permutative rewrite rules known to have the finite termination property. When ρ consists entirely of commutative equations, it is shown that there is an algorithm which decides unique termination for finite sets of permutative rewrite rules known to have the finite termination property. For more general sets ρ it is not presently known if there is an algorithm which decides unique termination for finite sets of rewrite rules known to have the finite termination property. Nevertheless, it is shown that the mathematical characterization of unique termination can be used with the completion attempting methods of Knuth and Bendix (1) to form the

REFERENCES

1. Knuth, D. E. and Bendix, P. B. Simple word problems in universal algebras. Computational Problems in Abstract Algebras, J. Leech, Ed., Pergamon Press, 1970, 263-297.

2. Lankford, D. S. Canonical inference. Automatic Theorem Proving Project, Depts. Math. and Comput. Sci., Univ. of Texas, report ATP-25, Jan. 1976.

3. Slagle, J. R. Automated theorem proving for theories with simplifiers, commutativity, and associativity. JACM 21, 4 (Oct. 1974), 622-642.

NOTES

NOTES

NOTES