MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Working Paper 201 July, 1980

A Synthesis of Language Ideas for AI Control Structures

William A. Kornfeld

ABSTRACT. Two well known programming methodologies for artificial intelligence research are compared, the so-called pattern-directed invocation languages and the object-oriented languages. The features and limitations of both approaches are discussed. We show that pattern-directed invocation is a more general formalism, but entails a serious loss of efficiency. We then go on to demonstrate that a language for artificial intelligence research can be created that contains the best features of both approaches.

1. Introduction

Quine once noted that "programming demands utter explicitness and formality in the analysis of concepts; furthermore it thrives on conceptual economy; and it rewards novel lines of analysis, with never a backward glance at traditional forms of thought." The issue of language design is very important in computer science as a whole, and particularly so in artificial intelligence.

There is a class of languages that have been studied extensively during the past decade in the artificial intelligence community known by the name *pattern-directed invocation.* In their purest form, pattern-directed invocation languages are conceptually very general; however this generality is handsomely paid for with serious efficiency problems. Few programmers are willing to pay the price of this great generality where it is only useful occasionally. In this paper we will try to explicate the sources of this generality and suggest a way of keeping the very clean conceptual model supplied by pattern-directed invocation where it is especially useful, while allowing more efficient means of information storage and retrieval in other situations.

We will also be concerned with another important class of languages studied in the artificial intelligence community, the so-called *object-oriented languages.* This interest will serve two purposes. These languages have been used with some success in the construction of relaxation-type constraint systems, process simulators, and KRL-like description languages. We will demonstrate that pattern-directed invocation type systems are conceptually more general, though less efficient than this class of languages in particular. We will then suggest how a system could be constructed that uses an object-oriented language for more efficient processing but can be made to look like a pattern-directed invocation language where this is desirable.

Our second reason for taking an interest in object-oriented languages is as a source of inspiration. These languages embody an important modularity principle. Our method for "having our cake and eating it too" derives from the application of this principle of modularity on a higher level.

2. Pattern-Directed Invocation Languages

Pattern-directed invocation grew out of early work in production systems. One of the first pattern-directed invocation languages was proposed by Hewitt [8] and implemented under the name Microplanner [23]. Various languages embodied extensions of these ideas including Conniver [17], QA4 [19], Qlisp [24], and Popler [5]. An excellent review of these languages is found in [2]. More recent developments include Amord [26] and Ether [11]. In what follows I will ignore the individual contributions of the individual languages and discuss them as a whole. When detail is required in places where the these languages differ I will largely follow my own biases as contained in Ether. The main difference between Ether and the other languages is the replacement of backtracking with parallel processing primitives as the vehicle for implementing heuristic search.

2.1 Assertions and Data Directed Processing

A key feature of these languages is the presence of a database of *assertions* representing facts learned about the problem at hand. These facts are often represented in a relational form such as:

```
(HAS MONKEY BANANAS)
```

The set of assertions forms a monotonically increasing collection of data. Procedures can be written that are invoked by matching their *patterns* against the assertions in the database. Following Ether, we will call these procedures *sprites*.† If a pattern match is successful, the body of the procedure is evaluated in an environment supplied by the pattern matcher. The amount of processing involved in matching a sprite's pattern against an assertion can be arbitrarily complex.‡ Sprites may take any action when triggered. In particular, they can add new assertions to the database or create new sprites capable of triggering on the database.

Pattern-directed invocation is a very powerful programming technique. We attribute this power to at least two sources:

(1) The system is very loosely coupled. It is easy for individual units of code embodying pieces of expertise to interact with one another. Separately written coding units do not have to be as meticulously intertwined as would be the case with more conventional coding methodologies. Other AI formalisms share this characteristic with pattern-directed invocation languages. These include the productions of Newell and Simon [18], the rule-based knowledge extensively developed by Shortliffe [20], Davis [6], and others, and the Hearsay II system of Lesser and Erman [14].

(2) The system is indifferent to the order of creation of sprites and potentially matching assertions. If a sprite and an assertion that can trigger it are created, the sprite will be triggered regardless of the order of their creation. This allows different parts of the system to operate as truly independent modules. Expert modules assert facts as they are learned. Other modules can ask whether these facts are known completely independently. This character of pattern-directed invocation languages eliminates module interaction

† Other names in the literature include *demons*, *methods*, and *rules*.
‡ Although none of the currently existing languages support this, the process of deciding whether an assertion matches a sprite could in itself involve a recursive call to the general reasoner (involving the use of assertions and sprites).

problems that can afflict programs written in conventional languages.

2.2 Decentralized Control Structure and Belief

The earlier languages exploited many new ideas in control structure, in particular: backtracking and coroutines. In my own work [11, 12] I develop the thesis that parallel processing can replace backtracking and coroutining in these languages and has many advantages over them.

Advanced control structures are employed to enable exploration of multiple approaches to solving a problem. Pure backtracking allows the system designer to specify alternative ways of solving a problem without regard to the order the methods are tried. The running program will eventually stumble upon the winning method if one exists. Parallel evaluation of methods considerably improves on backtracking for two reasons. (1) The program can reallocate resources comparing partial results obtained by the methods as they are running. (2) In addition to running the various methods, other parallel activities can be initiated for the purpose of refuting the applicability of a method. If the program is successful at showing a method cannot possibly succeed, resources allocated to solving it can be reclaimed. We call methods attempting to eliminate other methods *opponents*. There are good reasons to believe reasoning systems employing *opponents* are more powerful and flexible than ones that do not [13].

Many methods for solving problems require the ability to *assume* facts and have the system reason about the implications of these facts. These languages supply various facilities for doing this. QA4 introduced the notion of a *context*. Ether generalizes this notion for use in a highly parallel environment to *viewpoint*. Amord uses a somewhat different mechanism known as a *truth maintenance system* [7]. These mechanisms have proved extremely important. The tendency seems to be going toward more and more intricate ways of expressing how one hypothetical world inherits facts from another. The difficulties in creating such mechanisms subsumes what McCarthy has called the *frame problem* [16].

3. Object-Oriented Languages

The ideas for the so-called *object-oriented languages* seem to originate with Simula [4], a language for writing simulation programs. Smalltalk [10] was developed as a language for a highly modular interactive computing facility based on *message passing* where all computation occurs by send messages to objects. ACT1 [9] builds on the ideas of Smalltalk by showing that message passing semantics is an ideal basis for developing notions of concurrent programming.

The central idea of objected-oriented languages is that all computation in the system is accomplished through the interaction of *objects.* Objects interact with other objects by sending them messages. There is no other mode of interaction between objects. It is here that the increased modularity of the approach is realized. To illustrate this, suppose we had a program that dealt with arrays as data. In an objected-oriented language we would obtain the value of an element of the array by sending the array a message with the appropriate index. Suppose I wanted this program to work on sparse arrays as well as normal arrays. A sparse array can be most compactly represented by a hash table containing the non-zero elements of the array. The algorithm for determining the value of an entry in this "array" is to hash the index. If the index hashes to an entry in the array, the corresponding value is returned; otherwise 0 is returned. We can make our program accept sparse arrays simply by providing sparse arrays with methods for responding to messages that are accepted by normal arrays. A program making use of this alternative representation for an array cannot tell it is anything other than a normal array. Data objects (such as arrays) take on a new status. They become autonomous objects whose internal structure is of no concern to the rest of the system. This aspect of object-oriented languages contains ideas similar to the *data abstractions* of CLU [15] and ALPHARD [25].

Another contribution of these languages is the notion of a *class.* Frequently different objects in a system will have certain characteristics in common. These objects can be defined as members of some class that express the commonality between them. Because all computation happens by message passing, the class specifies messages that are handled identically by all instances of the class. Classes can be structured into hierarchies so that members of one class will be automatically supplied with the methods of its superclass(es).

Object-oriented languages were originally developed to facilitate simulations where each object in the system represented a part of the system being simulated. The ability to simulate systems is clearly a useful feature in any reasoning system. The metaphor of *constraint propagation* has proved useful in some systems [3, 21]. When an object learns of a new constraint on itself, it sends constraint messages to all objects it is related to. These messages may result in the determination of new constraints resulting in more messages being sent. Some recent systems have adopted the constraint passing concept as a way to build KRL-like description systems [22, 1].

4. Comparing The Two Approaches

In some sense comparing these two classes of languages is like comparing apples with oranges. Pattern-directed invocation languages supply mechanisms for problem solving-oriented control structures and relativized belief systems. Object-oriented languages typically do not supply such mechanisms. We will suggest shortly that there is no reason object-oriented languages cannot be augmented to include them. The point of comparison we wish to make in this section is between the means for storing and accessing knowledge. We believe that pattern-directed invocation supplies a conceptually more general framework with which to process knowledge; however this is at great computational cost.

4.1 Pattern Directed Invocation is Conceptually More Useful

To demonstratrate this point we will show (1) Any computation that is expressible using the object-oriented approach can be expressed just as succinctly using pattern-directed invocation; and (2) Pattern-directed invocation can define certain computations in a concepually cleaner manner than the object-oriented approach.

When a message is sent to an object, its function is to tell that object some fact about itself, request it to take some action, or ask it a question. This behavior of message passing can easily be emulated in pattern-directed assertional form by a trivial translation. Instead of sending object T meassage M, simply make an assertion by augmenting M to include a mention of T. If you wanted to tell T that it had a certain characteristic it would suffice to assert the T has that characteristic. Analogously the message handler could be transformed to be a sprite that watches for the assertion mentioning the object T.

As an example, let **Susan** be an object in the system. We might send **Susan** a message such as:

```
"You are a friend of Mary."
```

Susan might then wish to take some action upon receipt of **friend of** messages. It would have a message handler of the form:

```
When I receive a message of the form
        "You are a friend of =X",
        take some action.
```

A pattern-directed invocation language would handle it in the following way. An assertion would be made stating

```
"Susan is a friend of Mary"
```

and a sprite created:

```
When "Susan is a friend of =X" has been asserted,
      take some action.
```

As you can see, there is little difference in the form of the code.

Classes can be emulated with equal facility. Suppose we have a class **SociablePerson**, and a method that says

```
When a member of the class SociablePerson receives a message
        "You are a friend of =X",
        send yourself the message "You send X Christmas cards."
```

In a pattern-directed invocation system, we might assert

```
                                    "Susan is a SociablePerson"
```

and contain a sprite

```
When "=X is a SociablePerson"
     and "=X is a friend of =Y" have been asserted,
     assert "X sends Y Christmas cards."
```

Again, there is no difference in conceptual complexity.

The object-oriented approach supplies natural pigeonholes with which to associate procedures and knowledge in situations where all knowledge consists of *predicates on the individual objects of the domain.* It becomes awkward whenever a property applies to a constellation of objects that your system wishes to know about. As an example, imagine a system representing a network of lines and points as might be used in a geometry exercise. The "objects" of the system are points and lines. Each point knows about the lines radiating from it; each line knows about its endpoints. Now suppose we would like to find the *rhombuses* (quadrilaterals with four equal sides) in the figure. Polygons are not pre-defined objects of the system. An algorithm that worked by message passing could be constructed to solve this problem, though it would be very unintuitive. A pattern-directed invocation method (sprite) for doing this might look like:

```
(when {(EqualLength (LINE =w =x) (LINE =x =y))
       (EqualLength (LINE =x =y) (LINE =y =z))
       (EqualLength (LINE =y =z) (LINE =z =w))}
      body to deal with newly discovered rhombus WXYZ)
```

This sprite has a trigger requiring 3 assertions to be present. Each assertion is of the form:

```
                              (equal (LINE =p =q) (LINE =q =r))
```

It is looking for an assertion stating that two lines sharing a common point are equal in length. Any assertion of that form will trigger the sprite. The sprite as a whole attempts find four lines forming a closed path, all of which are equal. If it succeeds with all three assertions in the trigger then a rhombus has been discovered.†

† There is actually a bug in the sprite given. It will successfully trigger on, in addition to rhombuses, all pairs of equal lines sharing a common point. These are actually degenerate rhombuses, in which two of the four points are the same. To exclude these degenerate cases we must include two additional assertions in the trigger: `(NotEqual =x =z)` and `(NotEqual =y =w)`.

4.2 Pattern Directed Invocation Has Serious Efficiency Problems

In a pattern-directed invocation system, all knowledge is represented and stored as actual assertions. Often there are more efficient or natural ways to represent the knowledge. If we were constructing a reasoning system for programming in Lisp we might want to reason about the contents of a particular list, L = (A B C D). The number of facts we might wish to use about this list is enormous. Some examples are:

```
(IS L (QUOTE (A B C D)))
 (IS (CAR L) (QUOTE A))
   (IS (LENGTH L) 4)
```

In programs using more conventional programming methodologies these facts could be easily derived "on the fly" from the list L. There is no need to save explicit assertions in associative memory which may require significant computation for the storage and retrieval process. The proliferation of storage and processing-inefficient assertions in pattern-directed invocation languages seems unavoidable.

A related source of inefficiency in these languages is the *global nature of all interactions.* Any assertion can *potentially* match any sprite pattern. In a simple-minded implementation of a pattern-directed invocation language, the efficiency of *all interaction* will degrade linearly with the size of the database. This is clearly unsatisfactory.

There are various schemes involving computer science techniques such as *discrimination nets* to ameliorate this situation. The idea is to effectively compile away the global aspect of the search to gain a pattern-retrieval time that is essentially independent of the size of the database. The problem with these schemes is that they depend critically on the characteristics of the pattern matching techniques. A slight increase in the power of the pattern matcher can require an enormous increase in the complexity of the encoding scheme to maintain the independence of access time as a function of system size. The flexibility of these languages seems to depend critically on the flexibility of pattern matching. This fact makes successful compilation at best difficult.

5. Combining The Two Approaches

We would like to demonstrate two points. We would like to show that the decentralized control and belief aspects of pattern-directed invocation languages can be integrated into an object-oriented system. Following this are some ideas on how the more general though less efficient pattern-invoked procedures can be made to work in cooperation with other programming formalisms, in particular object-oriented systems.

5.1 Heuristic Search in A Constraint System

Object-oriented constraint passing systems that work by relaxation have certain theoretical limitations. An example that has been studied extensively by the author are the so-called *cryptarithmetic* problems found in Newell and Simon's book [18]. A well-known example from their book is the following:

```
  D O N A L D
+ G E R A L D
  -----------
  R O B E R T
```

An object-oriented constraint-passing system for solving this kind of problem might consist of three kinds of objects:

> One for each letter. Each letter contains a list of the digits it can possibly be. Whenever it receives messages on further restrictions on its values, it sends messages to the respective digit nodes indicating they are eliminated. If its list of possible digits is lowered to 1, it sends a message to that digit node indicating which letter it is.
>
> One for each column. Each column knows the current constraints on the letters it contains, and on its carry-in and carry-out. Whenever it receives messages containing constraints on its letters or carry-in or carry-out, it computes its new constraints. If there are any new constraints, messages containing them are sent to the respective letters or columns.
>
> One for each digit. Each digit contains a list of the letters it can possibly be. It receives messages from letters on its list indicating that it can or cannot by that particular letter. If its list of possible letters is reduced to 1, it sends a message to the remaining letter indicating that it must be this particular digit.

It is easy to verify that the system is "maximally constraining." All information derivable at any given node is made available to all users of this information. This system, however, will not solve the problem example cited above.† The solution of this problem requires the ability to *hypothesize* assignments in *hypothetical worlds* and then reason antecedently from these assumptions (i.e. plug them into the constraint network). A constraint system has been constructed for solving this class of problems that makes use of the heuristic search mechanisms developed for the Ether language [11].

† It *will* work with the additional constraint that D=5, however. It is worth pointing out that even without the constraint D=5 there is only one consistent assignment of letters to digits. Thus the inability of the system to discover this assignment indicates a fundamental limitation of constraint systems without a heuristic search mechanism.

To explain how this system works a few words must be said about the parallel processing primitives of Ether.

Ether contains the notion of an *activity*. All processing happens as part of some activity. Within the context of our cryptarithmetic problem solver, one activity may contain all work (i.e. message transmissions) pursuing constraint propagation under the assumption that, say, D=6. Another activity may be pursuing constraint propagation with the assumptions D=5 and E=4. The activity that a message transmission is in is totally invisible to the constraint system itself. Activities can be created at will and become subactivities of the activity that created them. So, for example, in pursuing the possible constraint D=6 it is decided to pursue the *additional* constraint that E=3, this would probably take place in a subactivity of the activity pursuing D=6. There are language primitives for controlling activities. The only one we will mention here is stifle. When an activity is stifled all work happening in the activity (such as messages transmissions in progress) is halted.

A cryptarithmetic problem solver has been constructed that works as follows. An activity is created in which constraints are passed through the network. When this activity becomes *quiescent* (when its constraints have totally relaxed), the most constrained letter that has more than one possible assignment is found. For each possible digit assignment of this letter, a new activity is created to pursue it. Similarly, if any of these new activities quiesce, the now most constrained letter is selected and new activities are created to pursue each possible assignment to this letter. This process continues until some activity has produced a unique assignment. Additional detectors are put at the digit and letter nodes looking for inconsistent hypothetical assumptions. An assumption is inconsistent if it leads to a situation where a letter cannot be any of the digits or, conversely, a digit cannot be any of the letters. When such a situation is noticed, the activity pursuing the particular hypothetical assignments is stifled.

Another mechanism is required to insulate the constraints of one assumption from the constraints of incompatible assumptions. In Ether we call these *viewpoints*. Each node of the constraint network contains separate databases indexed by viewpoint that contain the incompatible sets of constraints.

The language in which nodes of the constraint network are designed knows about viewpoints and thus knows when constraints that apply to one viewpoint should be passed to subviewpoints (containing more hypothetical assignments). This passing happens automatically. It turns out that the form of the code for our constraint network allowing heuristic search differs little from the code for a conventional constraint network.

5.2 Virtual Collections of Assertions

We would like to make it possible to interface pattern-invoked methods (sprites) with systems in which most, if not all knowledge, is represented in more specialized ways such as by being attached to an object that the knowledge can be said to predicate. Pattern-directed invocation systems normally represent knowledge by assertions in a database. We propose a kind of procedure that can determine if an assertion is virtually present in the environment (i.e. the knowledge is there, but not in assertional form). We call such a procedure a *virtual collection of assertions* (or VCA for short).

When a sprite is created with a given pattern it is compared against *actual* assertions in the database. In addition to checking for matching assertions, its pattern is compared against patterns of the VCAs, i.e. we check for the existence of a procedure that can decide if the assertion is *virtually present* in the database. If a VCA pattern matches the sprite pattern, the sprite trigger pattern is replaced with a call to this procedure. The VCA procedure is now responsible for evaluating the sprites body whenever a new fact has been learned that is in the domain of the particular VCA. The "facts" are stored in whatever way is convenient for this VCA. What we have done is short-circuited the default method for storing and retrieving information (assertions and pattern matching) whenever a more efficient method is known. The form of the high-level sprite code is not changed.

As an example, suppose we have a class of objects in our system **Person**. Associated with this class is a VCA for deciding virtual presence of assertions of the form: **(SociablePerson =p)**. This VCA will be able to respond to two classes of sprites. The first is where an individual person appears in the second position, such as:

```
(when (SociablePerson Susan)
      -- body to be evaluated --)
```

If this sprite is created, the VCA will know to check the **Susan** object for presence of this information. If its is found, the body is evaluated.

The VCA can also be made to handle requests of the form:

```
(when (SociablePerson =x)
      -- body to be evaluated --)
```

When the VCA encounters this sprite, it checks *every* **Person** node. For each node that the fact is believed true of, the body is evaluated in an environment where **x** is bound to the node. The VCA has the further responsibility of remembering the sprite requests so that if the fact is learned at any future time the sprite body will be evaluated.

The concept of a VCA is very general. A VCA can be built around any kind of procedure or data structure. Information can, for example, be represented as a highly compact table of bits and yet still be processable via pattern-directed invocation. The modularity principle embodied in the concept of a VCA is quite similar in spirit to that of the object-oriented languages we have been studying. A sprite using a VCA need not know how information is actually stored and retrieved for it to make use of this information.

6. Summary

We have examined two classes of languages of great interest in our community, *pattern-directed invocation* languages and *object-oriented* languages. We have shown that instead of being competing metaphors for AI programming they are in fact complimentary. The powerful ideas of pattern-directed invocation include control structures and belief systems for performing heuristic search as well as a very general mechanism for performing deductions. The object-oriented approach supplies efficient ways for storing and manipulating many kinds of knowledge by the use of message passing techniques. The ideas expressed in this paper have been implemented including the cryptarithmetic problem solver mentioned above. We have not as of this writing implemented any problem solving systems that make significant use of the virtual collections idea.

Our long term goal is to be able to write programs in which individual components store knowledge and reason in very different ways; yet where these individual components can interact with one another through clean, uniform interfaces.

7. References

[1] Attardi, Giuseppe, Carl Hewitt, *The Omega Description System*, Work in preparation, 1980.

[2] Bobrow, Daniel G., Bertram Raphael, *New Programming Languages for Artificial Intelligence Research*, Computing Surveys, Vol. 6, No. 3, September 1974.

[3] Borning, Alan, *Thinglab -- A Constraint-Oriented Simulation Laboratory*, XEROX PARC report SSL-79-3, July 1979.

[4] Dahl, Ole-Johan, Kristen Nygaard, *Simula -- An ALGOL-Based Simulation Language*, Communications of the ACM, September 1966.

[5] Davies, Julian, *POPLER 1.5 Reference Manual*, School of Artificial Intelligence, University of Edinburgh, TPU Report no. 1, May 1973.

[6] Davis, Randall, *Applications of Meta Level Knowledge to the Construction Maintenance and Use of Large Knowledge Bases*, Stanford Artificial Intelligence memo AIM-283, July 1976.

[7] Doyle, Jon, *Truth Maintenance Systems for Problem Solving*, MIT Artificial Intelligence Laboratory TR-419, January 1978.

[8] Hewitt, Carl, *PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot*, First International Joint Conference on Artificial Intelligence, 1969.

[9] Hewitt, Carl, *Viewing Control Structures as Patterns of Passing Messages*, Artificial Intelligence Journal, vol. 8, no. 1, June 1980.

[10] Ingalls, Daniel, *The Smalltalk-76 Programming System: Design and Implementation*, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson Arizona, January 1978.

[11] Kornfeld, William, *Using Parallel Processing for Problem Solving*, MIT Artificial Intelligence Laboratory memo 561, December 1979.

[12] Kornfeld, William, *Replacing Backtracking with Parallel Control for Heuristic Search*, work in preparation, 1980.

[13] Kornfeld, William, Carl Hewitt, *The Scientific Community Metaphor*, IEEE Systems Man & Cybernetics, *forthcoming*, 1980.

[14] Lesser, Victor R., Lee D. Erman, *A Retrospective View of the Hearsay II Architecture*, Fifth International Joint Conference on Artificial Intelligence, 1977.

[15] Liskov, Barbara, *CLU Reference Manual*, MIT Laboratory for Computer Science TR-225, October 1979.

[16] McCarthy, John, Pat Hays, *Some Philosophical Problems from the Standpoint of Artificial Intelligence*, Stanford Artificial Intelligence Project memo No. AI-73, November 1968.

[17] McDermott, Drew, Gerald Sussman, *The CONNIVER Reference Manual*, Artificial Intelligence Laboratory memo 259a, January 1974.

[18] Newell, Alan, Herbert A. Simon, *Human Problem Solving*, Prentice Hall, 1972.

[19] Rulifson, J., Jan A. Derksen, Richard J. Waldinger, *QA4: A Procedural Calculus for Intuitive Reasoning*, Stanford Research Institute Artificial Intelligence Center Technical Note 73.

[20] Shortliffe, E. H., *Computer-based Medical Consultations: MYCIN*, American Elsevier, 1976.

[21] Steele, Guy L., Gerald Sussman, *Constraints*, MIT Artificial Intelligence Laboratory memo 502, November 1978.

[22] Steels, Luc, *Reasoning Modeled as a Society of Communicating Experts*, MIT Artifical Intelligence Laboratory TR-542, June 1979.

[23] Sussman, Gerald, T. Winograd, E. Charniak, *Micro-Planner Reference Manual*, MIT Artificial Intelligence Laboratory memo 203, 1970.

[24] Wilber, Michael B., *A QLISP Reference Manual*, Stanford Research Institute Artificial Intelligence Center Technical Note 118.

[25] Wulf, W., R. London, M. Shaw, *An Introduction to the Construction and Verification of Alphard Programs*, IEEE Transactions on Software Engineering, SE-2, 1976.

[26] de Kleer, J., J. Doyle, C. Rich, G. Steele, G. Sussman, *AMORD A Deductive Procedure System*, MIT Artificial Intelligence Laboratory Memo 435, January 1978.