SRI International

THE ROLE OF LOGIC IN ARTIFICIAL INTELLIGENCE

Technical Note 335

July 1984

By: Robert C. Moore
Artificial Intelligence Center
Computer Science and Technology Division

SRI IR&D Project 500KZ

To be presented at the conference "Advanced Information Technology: Applications, Achievements, and Prospects," to be held at Churchill College, Cambridge, England, 20-22 September 1984.

Preparation of this paper was made possible by a gift from the System Development Foundation as part of a coordinated research effort with the Center for the Study of Language and Information, Stanford University.

SRI International®

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

ABSTRACT

Formal logic has played an important part in artificial intelligence (AI) research for almost thirty years, but its role has always been controversial. This paper surveys three possible applications of logic in AI: (1) as an analytical tool, (2) as a knowledge representation formalism and method of reasoning, and (3) as a programming language. The paper examines each of these in turn, exploring both the problems and the prospects for the successful application of logic.

I LOGIC AS AN ANALYTICAL TOOL

Analysis of the content of knowledge representations is the application of logic in artificial intelligence (AI) that is, in a sense, conceptually prior to all others. It has become a truism to say that, for a system to be intelligent, it must have knowledge, and currently the only way we know of for giving a system knowledge is to embody it in some sort of structure--a knowledge representation. Now, whatever else a formalism may be, at least some of its expressions must have truth-conditional semantics if it is really to be a representation of knowledge. That is, there must be some sort of correspondence between an expression and the world, such that it makes sense to ask whether the world is the way the expression claims it to be. To have knowledge at all is to have knowledge[1] that the world is one way and not otherwise. If one's "knowledge" does not rule out any possibilities for how the world might be, then one really does not know anything at all. Moreover, whatever AI researchers may say, examination of their practice reveals that they do rely (at least informally) on being able to provide truth-conditional semantics for their formalisms. Whether we are dealing with conceptual dependencies, frames, semantic networks, or what have you, as soon as we say that a particular piece of structure represents the assertion (or belief, or knowledge) that John hit Mary,

we have hold of something that is true if John did hit Mary and false if he didn't.

Mathematical logic (particularly model theory) is simply the branch of mathematics that deals with this sort of relationship between expressions and the world. If one is going to analyze the truth-conditional semantics of a representation formalism, then, a fortiori, one is going to be engaged in logic. As Newell puts it [1980, p. 17], "Just as talking of programmerless programming violates truth in packaging, so does talking of a non-logical analysis of knowledge."

While the use of logic as a tool for the analysis of meaning is perhaps the least controversial application of logic to AI, many proposed knowledge representations have failed to pass minimal standards of adequacy in this regard. (Woods [1975] and Hayes [1977] have both discussed this point at length.) For example, Kintch [1974, p. 50] suggests representing "All men die" by (DIE,MAN) & (ALL,MAN). How are we to evaluate such a proposal? Without a formal specification of how the meaning of this complex expression is derived from the meaning of its parts, all we can do is take the representation on faith. However, given some plausible assumptions, we can show that this expression cannot mean what Kintch says it does.

The assumptions we need to make are that "&" means logical conjunction (i.e., "and"), and that related sentences receive analogous representations. In particular, we will assume that any expression of

the form (P & Q) is true if and only if P is true and Q is true, and that "Some men dance" ought to be represented by (DANCE,MAN) & (SOME,MAN). If this were the case, however, "All men die" and "Some men dance" taken together would imply "All men dance." That, of course, does not follow, so we have shown that, if our assumptions are satisfied, the proposed representation cannot be correct. Perhaps Kintch does not intend for "&" to be interpreted as "and," but then he owes us an explanation of what it does mean that is compatible with his other proposals.

Just to show that these model theoretic considerations do not simply lead to a requirement that we use standard logical notation, we can demonstrate that ALL(MAN,DIE) could be an adequate representation of "All men die." We simply let MAN denote the set of all men, let DIE denote the set of all things that die, and let ALL(X,Y) be true whenever the set denoted by X is a subset of the set denoted by Y. Then it will immediately follow that ALL(MEN,DIE) is true just in case all men die. Hence there is a systematic way of interpreting ALL(MEN,DIE) that is compatible with what it is claimed to mean.

The point of this exercise is that we want to be able to write computer programs whose behavior is a function of the meaning of the structures they manipulate. However, the behavior of a program can be directly influenced only by the form of those structures. Unless there is some systematic relationship between form and meaning, our goal cannot be realized.

II LOGIC AS A KNOWLEDGE REPRESENTATION AND REASONING SYSTEM

A. The Logic Controversy in AI

The second major application of logic to artificial intelligence is to use logic as a knowledge representation formalism in an intelligent computer system and to use logical deduction to draw inferences from the knowledge thus represented. Strictly speaking, there are two issues here. One could imagine using formal logic in a knowledge representation system, without using logical deduction to manipulate the representations, and one could even use logical deduction on representations that have little resemblance to standard formal logics; but the use of a logic as a representation and the use of logical deduction to draw inferences from the knowledge represented fit together in such a way that it makes most sense to consider them simultaneously.

This is a much more controversial application than merely using the tools of logic to analyze knowledge representation systems. Indeed, Newell [1980, p. 16] explicitly states that "the role of logic [is] as a tool for the analysis of knowledge, not for reasoning by intelligent agents." It is a commonly held opinion in the field that logic-based representations and logical deduction were tried many years ago and were found wanting. As Newell [1980, p. 17] expresses it, "The lessons of

the sixties taught us something about the limitations of using logics for this role."

The lessons referred to by Newell were the conclusions widely drawn from early experiments in "resolution theorem-proving." In the mid 1960s, J. A. Robinson developed a relatively simple, logically complete method for proving theorems in first-order logic, based on the so-called resolution principle:[2]

$$(P \vee Q),\ (\neg P \vee R) \mid\!- (Q \vee R)$$

That is, if we know that either P is true or Q is true and that either P is false or R is true, then we can infer that either Q is true or R is true.

Robinson's work brought about a rather dramatic shift in attitudes regarding the automation of logical inference. Previous efforts at automatic theorem-proving were generally thought of as exercises in expert problem solving, with the domain of application being logic, geometry, number theory, etc. The resolution method, however, seemed powerful enough to be used as a universal problem solver. Problems would be formalized as theorems to be proved in first-order logic in such a way that the solution could be extracted from the proof of the theorem.

The results of experiments directed towards this goal were disappointing. The difficulty was that, in general, the search space

generated by the resolution method grows exponentially (or worse) with the number of formulas used to describe the problem and with the length of the proof, so that problems of even moderate complexity could not be solved in reasonable time. Several domain-independent heuristics were proposed to try to deal with this issue, but they proved too weak to produce satisfactory results. In the reaction that followed, not only was there was a turning away from attempts to use deduction to create general problem solvers, but there was also widespread condemnation of any use of logic in commonsense reasoning or problem-solving systems.

B. The Problem of Incomplete Knowledge

Despite the disappointments of the early experiments with resolution, there has been a recent revival of interest in the use of logic-based knowledge representation systems and deduction-based approaches to commonsense reasoning and problem solving. To a large degree this renewed interest seems to stem from the recognition of an important class of problems that resist solution by any other method.

The key issue is the extent to which a system has complete knowledge of the relevant aspects of the problem domain and the specific situation in which it is operating. To illustrate, suppose we have a knowledge base of personnel information for a company and we want to know whether any programmer earns more than the manager of data processing. If we have recorded in our knowledge base the job title and salary of every employee, we can simply find the salary of each

programmer and compare it with the salary of the manager of data processing. This sort of "query evaluation" is essentially just an extended form of table lookup. No deductive reasoning is involved.

On the other hand, we might not have specific salary information in the knowledge base. Instead, we might have only general information such as "all programmers work in the data processing department, the manager of a department is the manager of all other employees of that department, and no employee earns more than his manager." From this information, we can deduce that no programmer earns more than the manager of data processing, although we have no information about the exact salary of any employee.

A representation formalism based on logic gives us the ability to represent information about a situation, even when we do not have a complete description of the situation. Deduction-based inference methods allow us to answer logically complex queries using a knowledge base containing such information, even when we cannot "evaluate" a query directly. On the other hand, AI inference systems that are not based on automatic-deduction techniques either do not permit logically complex queries to be asked, or they answer such queries by methods that depend on the possesion of complete information.

First-order logic can represent incomplete information about a situation by

Saying that something has a certain property without saying which thing has that property:
$\exists x P(x)$

Saying that everything in a certain class has a certain property without saying what everything in that class is:
$\forall x(P(x) \supset Q(x))$

Saying that at least one of two statements is true without saying which statement is true:
$(P \vee Q)$

Explicitly saying that a statement is false, as distinguished from not saying that it is true:
$\neg P$

These capabilities would seem to be necessary for handling the kinds of incomplete information that people can understand, and thus they would be required for a system to exhibit what we would regard as general intelligence. Any representation formalism that has these capabilities will be, at the very least, an extension of classical first-order logic, and any inference system that can deal adequately with these kinds of generalizations will have to have at least the capabilities of an automatic-deduction system.

C. The Control Problem in Deduction

If the negative conclusions that were widely drawn from the early experiments in automatic theorem-proving were fully justified, then we would have a virtual proof of the impossibility of creating intelligent systems based on the knowledge representation approach, since many types of incomplete knowledge that people are capable of dealing with seem to demand the use of logical representation and deductive inference. A

careful analysis, however, suggests that the failure of the early attempts to do commonsense reasoning and problem solving by theorem-proving had more specific causes that can be attacked without discarding logic itself.

The point of view we shall adopt here is that there is nothing wrong with using logic or deduction per se, but that a system must have some way of knowing, out of the many possible inferences it could draw, which ones it should draw. A very simple, but nonetheless important, instance of this arises in deciding how to use assertions of the form $P \supset Q$ ("P implies Q"). Intuitively, such a statement has at least two possible uses in reasoning. Obviously, one way of using $P \supset Q$ is to infer Q, whenever we have inferred P. But $P \supset Q$ can also be used, even if we have not yet inferred P, to suggest a way to infer Q, if that is what we are trying to do. These two ways of using an implication are referred to as forward chaining ("If P is asserted, also assert Q") and backward chaining ("To infer Q, try to infer P") , respectively. We can think of the deductive process as a bidirectional search, partly working forward from what we already know, partly working backward from what we would like to infer, and converging somewhere in the middle.

Unrestricted use of the resolution method turns out to be equivalent to using every implication both ways, leading to highly redundant searches. Domain-independent refinements of resolution avoid some of this redundancy, but usually impose uniform strategies that may be inappropriate in particular cases. For example, often the strategy

is to use all assertions only in a backward-chaining manner, on the grounds that this will at least guarantee that all the inferences drawn are relevant to the problem at hand.

The difficulty with this approach is that whether it is more efficient to use an assertion for forward chaining or for backward chaining can depend on the specific form of the assertion, or the set of assertions in which it is embedded. Consider, for instance, the following schema:

$$\forall x(P(F(x)) \supset P(x))$$

Instances of this schema include such things as:

$$\forall x(x+1<y \supset x<y)$$

$$\forall x(JEWISH(MOTHER(x)) \supset JEWISH(x))$$

That is, a number x is less than a number y if x+1 is less than y; and a person is Jewish if his or her mother is Jewish.[3]

Suppose we were to try to use an assertion of the form $\forall x(P(F(x)) \supset P(x))$ for backward chaining, as most "uniform" proof procedures would. In effect, we would have the rule, "To infer P(x), try to infer P(F(x))." If, for instance, we were trying to infer P(A), this rule would cause us to try to infer P(F(A)). This expression, however, is also of the form P(x), so the process would be repeated, resulting in an infinite descending chain of formulas to be inferred:

P(A)
P(F(A))
P(F(F(A)))
P(F(F(F(A)))), etc.

If, on the other hand, we use the rule for forward chaining, the number of applications is limited by the complexity of the assertion that originally triggers the inference. Asserting a formula of the form P(F(x)) would result in the corresponding instance of P(x) being inferred, but each step reduces the complexity of the formula produced, so the process terminates:

P(F(F(A)))
P(F(A))
P(A)

It turns out, then, that the efficent use of a particular assertion often depends on exactly what that assertion is, as well as on the context of other assertions in which it is embedded. Kowalski [1979] and Moore [1980] illustrate this point with examples involving not only the distinction between forward chaining and backward chaining, but other control decisions as well.

In some cases, control of the deductive process is affected by the details of how a concept is axiomatized, in ways that go beyond "local" choices such as that between forward and backward chaining. Sometimes logically equivalent formalizations can have radically different behavior when used with standard deduction techniques. For example, in the blocks world that has been used as a testbed for so much AI

research, it is common to define the relation "A is ABOVE B" in terms of the primitive relation "A is (directly) ON B," with ABOVE being the transitive closure of ON. This can be done formally in at least three ways:[4]

$$\forall x,y(\mathrm{ABOVE}(x,y) \equiv (\mathrm{ON}(x,y) \vee \exists z(\mathrm{ON}(x,z) \wedge \mathrm{ABOVE}(z,y))))$$

$$\forall x,y(\mathrm{ABOVE}(x,y) \equiv (\mathrm{ON}(x,y) \vee \exists z(\mathrm{ABOVE}(x,z) \wedge \mathrm{ON}(z,y))))$$

$$\forall x,y(\mathrm{ABOVE}(x,y) \equiv (\mathrm{ON}(x,y) \vee \exists z(\mathrm{ABOVE}(x,z) \wedge \mathrm{ABOVE}(z,y))))$$

Each of these axioms will produce different behavior in a standard deduction system, no matter how we make such local control decisions as whether to use forward or backward chaining. The first axiom defines ABOVE in terms of ON, in effect, by iterating upward from the lower object, and would therefore be useful for enumerating all the objects that are above a given object. The second axiom iterates downward from the upper object, and could be used for enumerating all the objects that a given object is above. The third axiom, though, is essentially a "middle out" definition, and is hard to control for any specific use.

The early systems for problem solving by theorem-proving were often inefficient because axioms were chosen for their simplicity and brevity, without regard to their computational properties--a problem that also arises in conventional programming. To take a well-known example, the simplest procedure for computing the nth Fibonacci number is a doubly recursive algorithm whose execution time is proportional to 2^n, while a

slightly more complicated, less intuitively defined, singly recursive procedure can compute the same function time proportional to n.

D. Prospects for Logic-Based Reasoning Systems

The fact that the issues discussed in this section were not taken into account in the early experiments in problem solving by theorem-proving suggests that not too much weight should be given to the negative results that were obtained. As yet, however, there is not enough experience with providing explicit control information and manipulating the form of axioms for computational efficiency to tell whether large bodies of commonsense knowledge can be dealt with effectively through deductive techniques. If the answer turns out to be "no," then some radically new approach will be required for dealing with incomplete knowledge.

III LOGIC AS A PROGRAMMING LANGUAGE

A. Computation and Deduction

The parallels between the manipulation of axiom systems for efficient deduction and the design of efficient computer programs were recognized in the early 1970s by a number of people, notably Hayes [1973], Kowalski [1974], and Colmerauer [1978]. It was discovered, moreover, that there are ways to formalize many functions and relations so that the application of standard deduction methods will have the effect of executing them as efficient computer programs. These observations have led to the development of the field of logic programming and the creation of new computer languages such as PROLOG [Warren, Pereira, and Pereira, 1977].

As an illustration of the basic idea of logic programming, consider the function APPEND, which appends one list to the end of another. This function can be implemented in LISP as follows:

```
(APPEND A B) =
(COND ((NULL A) B)
      (T (CONS (CAR A) (APPEND (CDR A) B))))
```

What this function definition says is that the result of appending B to the end of A is B if A is the empty list, otherwise it is a list whose

first element is the first element of A and whose remainder is the result of appending B to the remainder of A.

We can easily write a set of axioms in first-order logic that explicitly say what we just said in English. If we treat APPEND as a three-place relation (with APPEND(A,B,C) meaning that C is the result of appending B to the end of A) the axioms might look as follows:

$$\forall x(\mathrm{APPEND}(\mathrm{NIL},x,x)$$

$$\forall x,y,z(\mathrm{APPEND}(x,y,z) \supset \forall w(\mathrm{APPEND}(\mathrm{CONS}(w,x),y,\mathrm{CONS}(w,z))))$$ [5]

The key observation is that, when these axioms are used via backward chaining to infer APPEND(A,B,x), where A and B are arbitrary lists and x is a variable, the resulting deduction process not only terminates with the variable x bound to the result of appending B to the end of A, it exactly mirrors the execution of the corresponding LISP program. This suggests that in many cases, by controlling the use of axioms correctly, deductive methods can be used to simulate ordinary computation with no loss of efficiency. The new view of the relationship between deduction and computation that emerged from these observations was, as Hayes [1973] put it, "Computation is controlled deduction."

The ideas of logic programming have produced a very exciting and fruitful new area of research. However, as with all good new ideas, there has been a degree of "over-selling" of logic programming and, particularly, of the PROLOG language. So, if the following sections

focus more on the limitations of logic programming than on its strengths, they should be viewed as an effort to counterbalance some of the overstated claims made elsewhere.

B. Logic Programming and PROLOG

To date, the main application of the idea of logic programming has been the development of the programming language PROLOG. Because it has roots both in programming methodology and in automatic theorem-proving, there is a widespread ambivalence about how PROLOG should be viewed. Sometimes it is seen as "just a programming language," although with some very interesting and useful features, and other times it is viewed as an "inference engine," which can be used directly as the basis of a reasoning system. On occasion these two ways of looking at PROLOG are simply confused, as when the (false) claim is made that to program in PROLOG one has simply to state the facts of the problem one is trying to solve and the PROLOG system will take care of everything else. This confusion is also evident in the terminology associated with the Japanese fifth generation computer project, in which the basic measure of machine speed is said to be "logical inferences per second." We will try to separate these two ways of looking at PROLOG, evaluating it first as a programming language and then as an inference system.

To evaluate PROLOG as a programming language, we will compare it with LISP, the programming language most widely used in AI.[6] PROLOG incorporates a number of features not found in LISP:

Failure-driven backtracking

Procedure invocation by pattern matching (unification)

Pattern matching as a substitute for selector functions

Procedures with multiple outputs

Returning and passing partial results via structures containing logical variables

These features and others make PROLOG an extremely powerful language for certain applications. For example, its incorporation of backtracking, pattern matching, and logical variables make it ideal for the implementation of depth-first parsers for language processing.[7] It is probably impossible to do this as efficiently in LISP as in PROLOG. Moreover, having pattern matching as the standard way of passing information between procedures and decomposing complex structures makes many programs much simpler to write and understand in PROLOG than in LISP. On the other hand, PROLOG lacks general purpose operators for changing data structures. In applications where such facilities are needed, such as maintaining a highly interconnected network structure, PROLOG can be awkward to use. For this type of application, using LISP is much more straightforward.

To better understand the advantages and disadvantages of PROLOG relative to LISP, it is helpful to consider that PROLOG and LISP both contain a purely declarative subset, in which every expression affects the course of a computation only by its value, not by "side effects." For example, evaluating (2+3) would normally not change the

computational state of the system, while evaluating (X←3) would change the value of X. In comparing their "pure" subsets, one finds that PROLOG is strictly more general than LISP. These subsets can both be thought of as logic programming languages, but the logic of pure LISP is restricted to recursive function definitions, while that of PROLOG permits definitions of arbitrary relations. This is what gives rise to the use of backtracking control structure, multiple return values, and logical variables. Pure PROLOG, then, can be thought of as a conceptual extension of pure LISP.

The creators of LISP, however, recognized that "although this language [pure LISP] is universal in terms of computable functions of symbolic expressions, it is not convenient as a programming system without additional tools to increase its power," [McCarthy et al, 1962, p. 41]. What was added to LISP was a set of operations for directly manipulating the pointer structures that represent the abstract symbolic expressions forming the semantic domain of pure LISP. LISP thus operates at two distinct levels of abstraction; simple things can be done quite elegantly at the level of recursive functions of symbolic expressions, while more complex tasks can be dealt with at the level of operations on pointer structures. Both levels, though, are conceptually coherent and, in a sense, complete.

PROLOG also has extensions to its purely logical core that most users agree are essential to its use as practical programming language. These extensions, however, do not have the kind of uniform conceptual

basis that the structure manipulation features of LISP do. Such features as the "cut" operation for terminating backtracking, "assert" and "retract" for altering the PROLOG database, and predicates that test whether variables are free or bound are all powerful and useful devices, but they do not share any common semantic domain of operation. There is nothing categorically objectionable about any of these features in isolation, but they do not fit together in a coherent way. The result is that, while PROLOG provides a very powerful set of tools, the effective use of those tools depends to a greater extent than with many other languages on the ingenuity of the programmer and his acquaintance with the lore of the user community.[8]

This suggests that if PROLOG is really to replace LISP as the language of choice for AI systems, it should be given a more powerful and more conceptually coherent set of nonlogical extensions to the basic logic-programming paradigm, analogous to LISP's nonlogical extensions to the recursive-function paradigm. This suggestion would no doubt be resisted by purists who see the present nonlogical features of PROLOG as already departing too far from the semantic elegance of a system where the correctness of a program can be judged simply by whether all of its statements are true; but that is an idealized vision whose practical realization is doubtful.[9]

C. PROLOG as an Inference System

Whatever its merits purely as a programming language, much of the current enthusiasm for PROLOG undoubtedly stems from the impression that, because a PROLOG interpreter can be viewed as an automatic theorem-prover, PROLOG itself can be used as the reasoning module of an intelligent system. This is true to an extent, but only to a limited extent. The major limitation is that all practical logic programming systems to date, including PROLOG, are based, not on full first-order logic, but on the Horn-clause subset of first-order logic.

The easiest way to view Horn-clause logic is to say that axioms must be either atomic formulas such as ON(A,B) or implications whose consequent is an atomic formula and whose antecedent is either an atomic formula or a conjunction of atomic formulas:

$$(ON(x,y) \wedge ABOVE(y,z)) \supset ABOVE(x,z)$$

Furthermore, the only queries that can be posed are those that can be expressed as a disjunction of conjunctions of atomic formulas:

$$(ON(A,B) \wedge ON(B,C)) \vee (ON(C,B) \wedge ON(B,A))$$

These limitations mean that no negative formulas--e.g., $\neg ON(A,B)$--can ever be asserted or inferred, and no disjunction can be inferred unless one of the disjuncts can be inferred. Thus, Horn-clause logic gives up two of the main features of first-order logic that permit reasoning with incomplete knowledge: being able to say or infer that one

of two statements is true without knowing which is true, and being able to distinguish between knowing that a statement is false and not knowing that it is true.

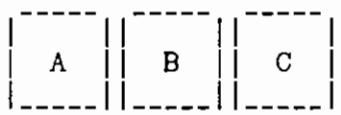
The question of quantification is more complicated. Horn-clause logic does not permit quantifiers per se, but it does allow formulas to contain function symbols and free variables, and there is a result (Skolem's theorem) to the effect that with these devices, any quantified formula can be replaced by one without quantifiers. However, this quantifier-elimination theorem does not apply to most logic programming systems, because of the way they implement unification (pattern matching).

According the usual mathematical definition of unification, a variable cannot be unified with any expression in which it is a proper subexpression. That is, x will not unify with F(G(x)), because there is no fully instantiated value for x that will make these two expressions identical. The test for this condition is usually called "the occur check." The occur check is computationally expensive, though, so most logic programming systems omit it for the sake of efficiency. There is a mathematically rigorous foundation for unification operation without the occur check, based on infinite trees, but this version of unification is not compatible with the quantifier-elimination techniques usually used in automatic theorem-proving. In particular, without the occur check, a logic programming system cannot properly distinguish between formulas that differ only in quantifier scope, such as,

$\forall x(\exists y(P(x,y)))$ and $\exists y(\forall x(P(x,y)))$. That is, the system cannot distinguish between the statement that every person has a mother, and the statement that every person has the same mother.

These restrictions are so severe that PROLOG is almost never used as a reasoning system without using the extra-logical features of the language to augment its expressive power. In particular, the usual practice is to define negation in the system, using the "cut" operation, so that $\neg P$ can be inferred by having an attempt to infer P terminate in failure. Making this extension permits the implementation of nontrivial reasoning systems in PROLOG in a very direct way,[10] but it amounts to making "the closed-world assumption": any statement that cannot be inferred to be true is assumed to be false. To adopt this principle, though, is to give up entirely on trying to reason with incomplete knowledge, which is the main advantage that logic-based systems have over their rivals.

To see what one gives up in making the closed-world assumption, consider the following problem, adapted from Moore [1980, p. 28]. Three blocks, A, B, and C, are arranged as shown:



A is green, C is blue, and the color of B is unstated. In this arrangement of blocks, is there a green block next to a block that is

not green? It should be clear with no more than a moment's reflection that the answer is "yes." If B is green, it is a green block next to the nongreen block C; if B is not green then A is a green block next to the nongreen block B.

To solve this problem, a reasoning system must be able to withold judgment on whether block B is green; it must know that either B is green or B is not green without knowing which; and it must use this fact to infer that some blocks stand in a certain relation to each other, without being able to infer which blocks these are. None of this is possible in a system that makes the closed-world assumption.

This is not to say that using PROLOG as a reasoning system with the closed-world assumption is always a bad thing to do. For applications where the closed-world assumption is justified, using PROLOG in this way can be extremely efficient--possibly more efficient than anything that can be programmed in LISP (for much the same reasons that top-down parsing is so efficient in PROLOG). But not all situations justify the closed-world assumption, and where it is not justified, the fact that PROLOG can be viewed as a theorem-prover is irrelevant. The usefulness of PROLOG in such a case will depend only on its utility as a programming language for implementing other inference systems.

IV CONCLUSIONS

In this paper we have reviewed three possible applications of formal logic in artificial intelligence: as a tool for analyzing knowledge-representation formalisms, as a source of representation formalisms and reasoning methods, and as a programming language. As an analytical tool, the mathematical framework developed in the study of formal logics is simply the only tool we have for analyzing anything as a representation. There is little more to say, other than to note all the efforts to devise representation formalisms that have come to grief for lack of adequate logical analysis.

The other two applications are more controversial. A large segment of the AI community believes that any representation or deduction system based on standard logic will necessarily be too inefficient to be of any practical value. We have argued that such negative conclusions are based on experiments in which there was insufficient control of the deductive process, and we have presented a number of cases in which better control would lead to more efficient processing. Moreover, we have argued that when an application involves incomplete knowledge of the problem, only systems based on logic seem adequate to the task.

The use of logic as a basis for programming languages is the most recent application of logic within AI. We had two major points to make in this area. First, current logic programming languages (i.e., PROLOG) need to be more developed in their nonlogical features before they can really replace LISP as the primary language for developing intelligent systems. Second, as they currently exist, logic programming languages are suitable for direct use as inference systems only in a very restricted class of applications.

After thirty years, where does the use of logic in AI now stand? In all fairness, would one have to say that its promise has yet to be proven--but, of course, that is true for most of the field of AI. It may be that, if the promise of logic is to be fulfilled, it will have to come in a remerging of two of the main themes explored in this paper: automatic deduction and logic programming. Logic programming grew out of the realization that, if automated reasoning systems are to perform efficiently, the information they are given must be carefully structured in much the same way that efficient computer programs are structured. But, instead of using that insight to produce more efficient reasoning systems, the developers of logic programming applied their ideas to more conventional programming problems. Perhaps the time is now right to take what has been learned about the efficient use of logic in logic programming, and apply it to the more general use of logic in automated reasoning. This just might produce the kind of basic technology for reasoning systems on which the development of the entire field depends.

NOTES

[1] Or at least a belief; most people in AI don't seem concerned too much about truth in the actual world.

[2] We will assume basic knowledge of first-order logic. For a clear introduction to first-order logic and resolution, see Nilsson [1980].

[3] I am indebted to Richard Waldinger for suggesting the latter example.

[4] These formalizations are not quite equivalent, as they allow for different possible interpretations of ABOVE, if infinitely many objects are involved. They are equivalent, however, if only a finite set of objects is being considered.

[5] To see the equivalence between the LISP program and these axioms, note that CONS(w,x) corresponds to A, so that w corresponds to (CAR A) and x corresponds to (CDR A).

[6] The fact that the idea of logic programming grew out of AI work on automated inference, of course, gives AI no special status as a domain of application for logic programming. But because it was developed by

people working in AI, and because it provides good facilities for symbol manipulation, most PROLOG applications have been within AI.

7
This is in fact the application for which it was invented.

8
To be fair, this last statement is true of LISP as well, especially with regard to recent extensions, such as "flavors." But it seems that with PROLOG one is forced into this domain of semantic uncertainty sooner than with LISP.

9
One can make a plausible argument that the advent of massively parallel computer architectures will change this situation. For the type of problem that would normally be solved by an algorithm that changes data structures, using an imperative language typically requires fewer computation steps than using a declarative language but creates more timing dependencies. Thus parallel architectures and declarative languages are well matched, because the architecture provides the greater computational resources required by the language, and the language provides the lack of timing dependencies required to take advantage of the architecture. It remains to be seen, however, for how wide a class of problems the speedups due to parallelism outweigh the additional computation steps required.

10
Ironically, it is necessary to go outside the purely logical subset of PROLOG to do this!

REFERENCES

Colmerauer, A. [1978] "Metamorphosis Grammars," in Natural-langauage Communication with Computers, L. Bolc, ed. (Springer-Verlag, Berlin, Germany).

Hayes, P. J. [1973] "Computation and Deduction," Proc. 2nd Symposium on Mathematical Foundations of Computer Science, Czechoslovak Academy of Sciences, pp. 105-116 (September 1973).

Hayes, P. J. [1977] "In Defence of Logic," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, pp. 559-565 (22-25 August 1977).

Kintch [1974] The Representation of Meaning in Memory (Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey).

Kowalski, R. [1974] "Predicate Logic as a Programming Language," in Information Processing 74, pp. 569-574 (North-Holland Publishing Company, Amsterdam, The Netherlands).

Kowalski, R. [1979] Logic for Problem Solving (Elsevier North Holland, Inc., New York, New York).

McCarthy, J., et al [1962] LISP 1.5 Programmer's Manual (The MIT Press, Cambridge, Massachusetts).

Moore, R. C. [1980] Reasoning from Incomplete Knowledge in a Procedural Deduction System (Garland Publishing, Inc., New York, New York).

Newell, A. [1980] "The Knowledge Level," Presidential Address, American Association for Artificial Intelligence, AAAI80, Stanford University, Stanford, California (19 August 1980), in AI Magazine, Vol. 2, No. 2, pp. 1-20 (Summer 1981).

Nilsson, N. J. [1980] Principles of Artificial Intelligence (Tioga Publishing Company, Palo Alto, California).

Robinson, J. A. [1965] "A Machine-Oriented Logic Based on the Resolution Principle," Journal of the Association for Computing Machinery, Vol. 12, No. 1, pp. 23-41 (January 1965).

Warren, D. H. D., Pereira, L. M., and Pereira, F. C. N. [1977] "PROLOG--The Language and Its Implementation Compared with LISP," in Proc. Symposium on Artificial Intelligence and Programming Languages (ACM); SIGPLAN Notices, Vol. 12, No. 8; and SIGART Newsletter, No. 64, pp. 109-115 (August 1977).

Woods, W. A. [1975] "What's in a Link: Foundations for Semantic Networks," in Representation and Understanding, D. G. Bobrow and A. Collins, eds., pp. 35-82 (Academic Press, Inc., New York, New York).