THE LIMITATIONS OF LOGIC

Robert Kowalski
Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ

Feigenbaum [4], commenting on the Fifth Generation Project, has said that logic is not important, but knowledge is. I agree that knowledge is more important than logic. But logic is important too. Knowledge-based systems need both knowledge and formalism. Although knowledge is more important than formalism, formalism is important because the use of a poor formalism can interfere with the representation of knowledge and can restrict the uses to which that knowledge can be put. I believe that logic is the least restrictive and most appropriate formalism for knowledge-based systems.

Knowledge-based systems combine both complex knowledge and sophisticated formalisms. I believe that this combination of knowledge and formalism accounts for some of the difficulty practicioners have had in explaining what knowledge-based systems are. Problems arise because we confuse knowledge with formalism. Many characterizations of expert systems for example concentrate simply on formalism, on rule-based languages for example and say very little about what makes such formalisms particularly appropriate for expressing and reasoning with knowledge.

Logic is strong on formalism but weak on concepts. It contains no knowledge, and is all form and no content. Indeed the significance of the model theoretic semantics of logic is precisely that: Model theory defines as valid precisely those sentences which are true in any interpretation. As a consequence, logic tells us nothing about the actual world itself.

To use logic to represent knowledge we have to identify a useful vocabulary of symbols to represent concepts. We have to formulate appropriate sentences, with the aid of that vocabulary, to represent the knowledge itself. Logic can help us to test an initial choice of vocabulary and sentences, by helping us to derive logical consequences and identify the assumptions which participate in the derivation of those consequences. It provides us with no help, however, in identifying the right concepts and knowledge in the first place.

A typical AI knowledge representation scheme, such as semantic networks or frames, combines concepts and formalism at the same time. It provides a built-in framework of ready-made concepts to help with the initial representation of knowledge. But it also provides a formalism to go along with the concepts. In the same way that a computer salesman might try to convince us that to run a particular piece of software we need to buy the appropriate hardware, a LISP machine for example, the developer of an AI system typically tries to convince us that to use a particular collection of concepts we need to buy an associated formalism. My thesis is that, in the same we can separate software from the hardware on which it is implemented, we can also separate concepts from formalisms. The same concepts can be implemented in other formalisms, including the formalism of logic.

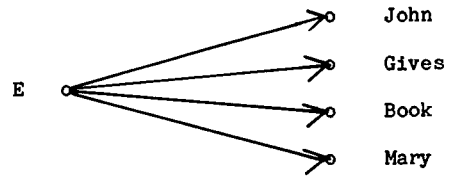
Semantic Networks

Semantic networks, for example, combine the concepts of events and hierarchies with a graphical formalism in which nodes represent individuals and arcs represent binary relationships. The same concepts, however, can be represented in other formalisms. Of particular importance, in my opinion, is the prominence given in semantic networks to the notion of event. The event calculus, which my colleague Marek Sergot and I [11] have developed, borrows concepts about events from semantic networks and implements them within a logic programming framework. Instead of representing the semantics of a sentence

An earlier version of this paper was presented at The Workshop on Knowledge Base Management Systems, held in Chania, Crete, June 1985, to be published by Springer Verlag.

"John gives the book to Mary"

by means of a network



we represent the same "knowledge" either by means of binary relationships

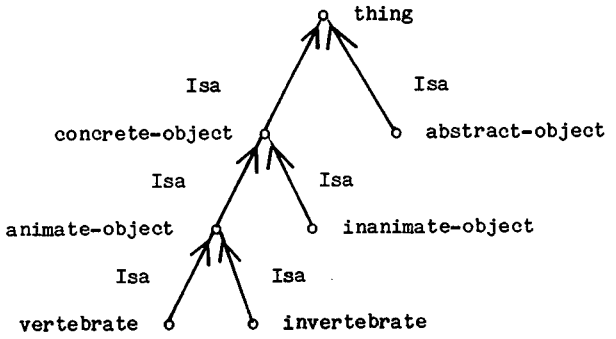
```
Actor(E John)
Act(E Gives)
Object(E book)
Recipient(E Mary)
```

or by means of a single relationship

```
Event(E John gives book Mary).
```

The contribution of semantic networks here has been the identification of events as a concept for building knowledge representations. Of some importance also is its identification of networks as a convenient user-friendly notation. (We shall discuss the relationship between formalism and notation later).

Semantic networks also focus attention on the concept of hierarchy. The concept of hierarchy, however, can be abstracted from the graphical notation and can be represented in other formalisms. For example, the hierarchy fragment



can be represented in logic either by means of binary relationships or by means of general rules:

```
Isa(vertebrate animate-object)
Isa(invertebrate inanimate-object)
Isa(animate-object concrete-object)
Isa(concrete-object thing)
etc.
```

or

```
Isa(x animate-object) if Isa(x vertebrate)
Isa(x animate-object) if Isa(x invertebrate)
Isa(x concrete-object) if Isa(x animate-object)
Isa(x thing) if Isa(x concrete-object)
```

Notice that in the first representation transitivity of "Isa" needs to be expressed by a general rule.

```
Isa(x y) if Isa(x z) and Isa(z y),
```

whereas in the second representation it comes for free. In both cases the inheritance of "mortality" by anything which is classified as an animate-object is represented by the rule

```
Mortal(x) if Isa(x animate-object)
```

Entity - relationships

Object-oriented programming, abstract data types, and the entity-relationship database model, like semantic networks, promote the concept of object as a way of organising knowledge. Whereas object-oriented programming and abstract datatypes single-mindedly force all knowledge to be stored with and accessed through objects, the entitly-relationship model allows entities to enter into relationships with other entities. Although the entity-relationship model may seem to conflict with the relational model, it now seems to be the consensus in the database community that the two models deal with different levels of knowledge representation and are not in conflict. Relations in the relational model can be used at a lower level as a formalism to implement the concepts of both properties and relationships in the higher level entity-relationship model. For example, the entity John with the properties of being 24 years old, male and born in the U.K. and with the relationship of being married to the entity Mary can all be represented as relationships, which can in turn be expressed in the formalism of logic:

```
Age(John 24)
Birth-place(John U.K.)
Sex(John Male)
Married(John Mary)
```

(Note that object-oriented programming, in contrast with the entity-relationship model, would force the "married" relationship either to be duplicated for both John and Mary or to be made into a separate entity with husband and wife properties).

Thus the entity-relationship model and the allied object-oriented programming and abstract data type models can be regarded as contributing primarily to the level of concepts, whereas the relational model and formal logic operate primarily at the lower level of formalism.

Frames

Frames are another example. Besides the concepts of hierarchy borrowed from semantic networks and of objects taken from object-oriented programming, frames focus attention on the concepts of stereotypes and default reasoning.

Frames encourage us, instead of reasoning from first principles on every occasion, to reason by comparing new occasions with preconceived stereotypes. Default assumptions about the new occasion are made in the absence of countradictions and are withdrawn if contradictory information is later made known.

The concepts of stereotype and default reasoning are useful for building knowledge-based systems. But in the context of frame-based systems they are generally combined with rather loosely defined formalisms associated with forms, slots and fillers. As Pat Hayes has pointed out [6], in many ways these formalisms are closer to logic than many of their predecessors, because a slot is like an argument place of a relation and a filler is like an argument. It should not be surprising therefore if we can implement stereotypes and default reasoning in other formalisms.

Consider, for example, the frame for "bird", represented as a form with slots for holding properties of birds. In the absence of information to the contrary, certain properties may have default values.

bird frame	Isa vertebrate
primary locomotion = default flight	
number of legs = default 2	
etc.	

This might be represented in logic programming formalism by the sentences

Isa(x vertebrate) **if** Isa(x bird)
Primary-locomotion(x flight) **if** Isa(x bird)
  and not [Primary-locomotion(x y) **and** y ≠ flight]
Number-of-legs(x 2) **if** Isa(x bird)
  and not [Number-of-legs(x y) **and** y ≠ 2]

Here the negation symbol "**not**" is interpreted as negation by failure [2]. This gives a good approximation to default reasoning (though, in this case, if executed by PROLOG, would give rise to an infinite loop, which can, however, be eliminated by program transformation techniques [8]).

Notice that another characteristic of the frame-based representation is the use of forms as a notation. This is undoubtedly more user-friendly than the notation of symbolic logic. Our defence of logic as a formalism, therefore, is not a defence of its notation but rather a defence of its abstract syntax, its semantics and its proof procedures.

Thus, to be more precise, I would have to argue that non-logic-based systems contribute to the identification both of useful concepts and of useful, user-friendly notations. Other formalisms, such as formal logic, can be used to implement the same concepts and notations.

In each of the preceeding examples, semantic networks, entity-relationships and frames, concepts are combined with formalism to a lesser or greater extent. The resulting formalisms and their associated notations facilitate expressing those particular concepts, but often hinder the expression of other concepts. The alternative to tying concepts and formalism so closely together is to employ a single universal formalism within which different and even competing concepts can be expressed and integrated. First-order predicate logic with certain embellishments seems to be the best candidate for such a formalism.

Some other systems with concepts which can usefully be reformulated in logic are Hewitt's Open Systems [7] and Schank's Conceptual Dependency Theory [12].

Open Systems

Hewitt regards the requirements of open systems as conflicting with the constraints of logic and logic programming. I believe that he has correctly identified an important class of problems previously neglected by students of logic. But, in my opinion, this neglect is not the result of any inherent limitation of logic.

Open systems consist of multi-actor knowledge-based systems, each with their own internal goals and able to perform actions to accomplish those goals. An actor's goals may be internally incompatible or conflict with the goals of other actors.

Actors in an open system dynamically change both their beliefs and their goals as a result of interacting with other actors and the changing environment. Such changing systems have been studied within the framework of knowledge assimilation in logic-based systems [9]. Logical deduction can assist the process of knowledge assimilation by focussing attention on the logical relationships between new knowledge and the current state of the knowledge-base. It can be used to determine whether the new knowledge logically implies existing knowledge, is implied by it, is inconsistent with it or is logically independent. The detection of these relationships is constrained by the amount of resources which can be expended.

To improve the efficiency of performing deductions, proof procedures attempt to avoid the derivation of irrelevant consequences. As a result an inconsistent set of beliefs can still be useful in practice - both because inconsistencies may not be detected and because the derivation of inconsistency need not lead to the derivation of irrelevant further consequences.

To achieve the power of open systems, however, such logic-based systems need to be augmented with their own internal goals and need to construct and execute plans of action to accomplish their goals [10]. For this purpose an actor needs to have a model of the current state of the environment and of the expected effect its actions have upon it. Both of these can be represented by sentences expressed in formal logic. An actor can use

logical deduction to construct a plan of action to accomplish one or more of its goals. Several such systems of plan-formation have been developed within the formalism of logic. The degree of success of failure of these systems, however, has depended more on the appropriateness of the world model than on its representation in logical formalism. This can be taken as further evidence for the thesis that knowledge is more important than logic.

Actors in open systems need to be able to perform actions to accomplish their own goals. Such an actor can be represented logically by means of a metalevel predicate

Process(input-stream knowledge-base output-stream)

For example, the (over-simplified) case where an actor processes an item of "input" which is at the head of an input-stream

cons(input rest-input-stream)

and does nothing to it if the input is derivable from the knowledge base can be represented by the rule

```
Process(cons(input  rest-input-stream)
        knowledge-base  output-stream)
        if Process(rest-input-stream
                        knowledge-base output-steam)
```

The parallel interaction of many such actors can be represented by a metalevel sentence executed by a parallel logic-programming interpreter, such as PARLOG [3] or concurrent PROLOG [13]. Logic-based systems of this kind have been proposed and investigated by Shapiro and Takeuchi [14] and Furukawa et al [5]. To a large extent these investigations have been motivated by the attempt to implement in logical formalism concepts first identified and highlighted in other, non-logic-based systems. They are an example of the benefits to logic of borrowing concepts from other formalisms.

Conceptual Dependency Theory

Conceptual dependency theory combines concepts about reducing the semantics of complex events to the semantics of a few primitive acts with a pictorial formalism. To take a specific example, the acts of "giving" and "taking" can both be reduced to special cases of the primitive act of transferring possession. In the case of "giving", the actor is the donor; in the case of "taking", the actor is the recipient. Schank describes these reductions of "giving" and "taking" in English and implements them in LISP. He uses his graphical formalism for representing concrete events, but has no formalism other than LISP for describing the reduction of events in general.

The reduction of "giving" and "taking" to "transfer-possession" can, however, be represented by means of logic programs which have both declarative and procedural interpretations:

```
Act(x transfer-possesion) if Act(x giving)
Donor(x y) if Act(x giving) and Actor(x y)
Actor(x y) if Act(x giving) and Donor(x y)
Act(x transfer-possesion) if Act(x taking)
Recipient(x y) if Act(x taking) and Actor(x y)
Actor(x y) if Act(x taking) and Recipient(x y)
```

Interpreted as logic programs these rules will be used backwards only when needed. Like many other declarative programs, however, when executed by PROLOG, they can go into infinite loops. These loops can be avoided by program transformations, or by applying more sophisticated proof procedures (employing loop-detection perhaps). In any case, by separating the declarative knowledge from its mode of use we obtain potentially greater flexibility and power than we have with the corresponding LISP routines, which can use the *same knowledge in only one, previously* anticipated, way.

Notice that rules which express properties of "transfer possession" such as

```
Possesses(y z after(e)) if Act(x transfer-
                                  possession)
                           and Recipient(x y)
                           and Object(x z)
Start(after(e) e)
```

i.e. "The recipient of an event of "transfer possession" possesses the object of the event for some, possibly indeterminate period of time after(e), which starts at e".

are automatically inherited by "giving" and "taking".

Thus concepts which have been originally introduced within the context of systems with non-logical formalisms can be rationally reconstructed in logical formalism and gain greater clarity and power as a result. Once knowledge has been represented explicitly in logical terms, it can be used to derive arbitrary logical consequences, in ways not originally anticipated and not catered for in the original non-logical formalisms. The price that sometimes has to be paid for this greater power, however, is that more flexible uses of the knowledge may require the application of more powerful proof procedures than are currently available. This can be partially alleviated by the use of program transformations, but in the longer term will require the development of more powerful and more efficient proof procedures.

The practice of logic itself benefits from such borrowing of concepts from non-logical systems. Non-logical systems, by comparison with logic, are more concept-oriented and can tell us therefore about the kinds of knowledge which need to be represented in any formalism. Logic can not progress without applications. Non-logical systems can help identify **the** concepts that are needed for building such applications.

Non-classical Logic

The combination of concept and formalism which is characteristic of A.I. systems not based on logic is also a feature of non-classical logics.

Logicians themselves can be as inclined as A.I. practitioners to invent different formalisms for different concepts. Thus we have temporal logics for dealing with time, relevance logics for relevant implication and fuzzy logics for uncertainty. According to the methodology associated with non-classical logic, to determine what logic is needed for a given application, it is necessary to analize the application in detail, identify the concepts needed and find a logic which formalizes those concepts. If the analysis is mistaken or a change needs to be made to the application for some other reason, then the entire application may need to be reformulated in another, more appropriate logic. Even if the better logic has already been developed and is available for the purpose, the process of complete reformalization creates an intolerable discontinuity in the knowledge representation process. This methodology is the complete opposite of the process of formalisation by top-down, successive refinement which is the hallmark of good practice in software engineering.

The inadequacy of this methodology is even more apparent with complex applications which require a multiplicity of different concepts associated with different logics. There are only two ways of tackling such applications - either by developing a methodology which allows different formalisms to be combined within a single application; or by abandoning special-purpose logics for the right universal formalism in the first place. The first alternative is workable for many applications of intermediate complexity where the problem can be decomposed into relatively self-contained subproblems each of which can be tackled with a single formalism. It will not work, however, for more complex problems, where several different concepts are intimately connected, as they might be, for example, within a single natural language sentence involving time, uncertainty and obligation:

"Tomorrow I will probably need to change my mind".

In my opinion the second alternative is better. We need a universal formalism, which is not tied to specific concepts, but within which different concepts can be represented and integrated. These concepts can be borrowed from other more specialized logics, extracted from non-logical systems or formulated specially for the problem at hand.

Classical Logic

There may be several candidates for the universal language; and it may not be obvious how to choose between them. My own belief is that the best candidate is classical first-order logic. Some extensions and even some restrictions will undoubtably be necessary. The Horn clause subset of first-order logic augmented with negation by failure, upon which logic programming is based, is such a restriction; the amalgamation of object language and metalanguage is such an extension. Amalgamation logic, however, does not really go beyond first-order logic, but simply gives more of it - at both the object language and metalanguage levels.

First-order logic makes a good candidate for the universal language, because it is the only logic which has been extensively applied, both inside and outside computing. It is the only formalism which has demonstrated its adequacy for formalising the foundations of mathematics. In computer science it is the only formalism which has been used not only for knowledge representation and problem-solving in Artificial Intelligence, but also for program specifications, databases, formal grammars and computer programs.

Building first-order logic on top of systems which efficiently implement the Horn clause subset of logic has an advantage, because the procedural interpretation of Horn clauses potentially gives such systems the efficiency of a computer programming languages.

Negation as failure:

not P holds if P fails to hold

for example, can be implemented very simply and very efficiently on top of Horn clause proof procedures. It gives a correct implementation of classical negation [2] under the assumption that the formalization contains a **complete** characterisation of the predicate P. Even with this assumption, however, negation as failure does not always give a complete implementation of classical negation. Nonetheless it can be used to implement conditions which have the expressive power of full first-order logic (even if they do not necessarily have its full deductive power). Consider for example, the definition of the subset relation:

x subset of y **if for all** z
z is in y **if** z is in x.

can be reduced to Horn clause form augmented with negation as failure:

x subset of y **if not exists** z
z is in x **and not** z is in y.

Executed backwards, logic programming style, with negation interpreted as failure, this behaves as a procedure which shows

x is a subset of y by
testing each element z in x and
showing each such z is in y.

This is a correct interpretation of the original definition of "subset", provided the "knowledge-base" contains a complete characterization of the "is in" relation. However, as it stands, the interpretation is incomplete because it can only be used to test whether x is a subset of y and not to generate subsets x of y or supersets y of x

The use of first-order logic at both the object level and the metalevel adds greatly to expressiveness and problem solving power. It can be used, not only at the ordinary object level, but also at the metalevel for programs and databases which manipulate and describe object level programs and databases. It can be used, in particular, to describe and implement knowledge

assimilation and multi-actor belief systems. It is even possible to devise an amalgamation of object language and metalanguage [1] which can self-referentially apply to itself - for an editor which can be used to edit itself or a compiler which will compile itself.

Classical first-order logic as presented in traditional logic books, however, is not necessarily the best starting point for its practical application. Indeed it might even be argued that the very success of symbolic logic applied to mathematics has contributed to its failure to be applied more widely outside of mathematics. The style of logic which has proved useful for foundations of mathematics, is bottom-up and reductionist, with all concepts reduced to the bare minimum. This is the opposite of the approach needed for most knowledge-based applications, which is top-down and concept-rich.

The bottom-up, reductionist use of logic, which is adequate for foundations of mathematics is not even useful for its practice. The notion of subset, for example, which is so central to the mathematical practice of set theory and which is the mathematical basis of ISA-hierchies is eliminated in the foundations of set theory in favour of the primitive membership relation. Logic as it has been applied to the foundations of mathematics teaches us not to worry about identifying useful concepts, but rather to eliminate them in favour of primitive concepts. Such primitives, however, are virtually impossible to use in practice.

The tradition of mathematical logic has other characteristics which can make it ill-suited for complex non-mathematical applications. It places inordinate emphasis on consistency and completeness and inhibits the process of trial and error, which is needed for developing such applications and which is an essential ingredient of expert systems methodology in particular.

The mathematical tradition of logic, however, is not an inherent characteristic of logic itself. Logic is sufficiently neutral with respect to both concepts and methodologies that it can integrate different concepts and it can adapt itself to different methodologies including that associated with top-down, trial and error development of knowledge.

The universal formalism whose adoption I have advocated is an elaboration of classical first-order logic. I have argued in its favour on theoretical grounds. Until recently the theoretical arguments might have been overshadowed by problems of efficiency. Advances in logic programming technology, however, have reached the stage where logic-based implementations of concepts are often as least as efficient as implementations in special-purpose languages. The application of compiler technology to the implementation of ISA-hierchies and inheritance formulated in logic, for example, compares well with their implementation in conventional programming languages.

Finally, we should note that the adoption of first-order logic as a universal formalism does not preclude the continuing use of other languages. First-order logic can coexist with other formalisms and can interoperate with them. Existing applications implemented in other formalisms can be incorporated within larger systems implemented in first-order logic, provided such applications can be viewed logically from the outside. Taking a well-structured, top-down point-of-view, there is no need to look inside. The actual implementation itself can be viewed as a compiled version of its rational reconstruction formulated in first-order logic.

So, the universal logic language can coexist with other languages, especially in the short term. It can even benefit from them by borrowing their concepts to facilitate the formalization of knowledge in logical terms. It can also be of benefit to other languages by helping to liberate their concepts from their formalisms and, by representing them in the formalism of logic, enabling those concepts to interoperate with other concepts liberated from other formalisms.

Acknowledgements

An earlier version of this paper was presented at The Workshop on Knowledge Base Management Systems, held in Chania, Crete, June 1985, to be published by Springer Verlag.

References

[1] Bowen K. A. and Kowalski R. A., [1982]. "Amalgamating language and metalanguage in logic programming", in Logic Programming, (K. L. Clark and S-A. Tarnlund, Eds.), Academic Press, London.

[2] Clark K. L., [1978]. "Negation as failure", in Logic and Data Bases, (H. Gallaire and J. Minker, Eds.), Plenum Press, New York.

[3] Clark K. L. and Gregory S., [1985]. "PARLOG : parallel programming in logic", to appear in ACM Trans. on Programming Languages and Systems. 1986.

[4] Feigenbaum E. A., [1982]. "Innovation and symbol manipulation in Fifth Generation Computer Systems", in Fifth Generation Computer Systems, pp. 223-224, (T. Moto-Oka, Eds.), North Holland, Amsterdam.

[5] Furukawa K. et al., [1984]. "Mandala : A logic based knowledge programming system", in Proceedings of the International Conference on Fifth Generation Computer Systems, pp. 613-622, Ohmsha Ltd., Tokyo.

[6] Hayes P. J., [1979]. "The Logic of Frames", in Frame Conceptions and Text Understanding, pp. 46-61, (D. Metzing, Eds.). Walter de Gruyter and Co., Berlin.

[7] Hewitt C., [1985]. "The Challenge of Open Systems". BYTE, April 1985, pp. 223-242.

[8] Hogger C. J., [1984]. "Introduction to Logic Programming". Academic Press. London.

[9] Kowalski R. A., [1979]. "Logic for Problem Solving". Elsevier North-Holland, New York.

[10] Kowalski R. A., [1985]. "Logic-based Open Systems", Department of Computing, Imperial College, London.

[11] Kowalski R. A. and Sergot M. J., [1984]. "Towards a logic-based calculus of events", to appear in New Generation Computing, Vol. 4, No. 1, February 1986, Ohmsha Ltd., Tokyo, and Springer Verlag, Berlin.

[12] Schank R. C., [1975]. "Conceptual Information Processing", North Holland, Amsterdam.

[13] Shapiro E. Y., [1983]. "A subset of Concurrent Prolog and its interpreter", in ICOT Technical Report TR-003, Institute New Generation Computing Technology, Tokyo.

[14] Shapiro E. Y. and Takeuchi A., [1983]. "Object oriented programming in Concurrent Prolog", in New Generation Computing 1, Springer Verlag. Berlin.