Knowledge systems: Principles and practice

by Adrian Walker

We describe what is expected of a knowledge-based expert system, and the components from which such a system is constructed. We give a view of how an interplay between principles and practice has helped the knowledge system field to develop, and we give simple examples to show that reasoning techniques based on formal logic are now starting to provide a useful coupling between scientific and engineering work in the field. The examples are about logic programming, knowledge representation, judgmental reasoning, and about three methods by which a system can acquire the knowledge it needs.

1. Introduction

The rate of publication of articles about expert systems is now higher than ever before. There are survey articles in computer science journals, in popular computing and scientific periodicals, and in the general press. The volume of research publication is also unusually high. Expert systems have so far proved their worth in structure elucidation in chemistry, in helping to find mineral deposits, in helping technicians in hospitals, in suggesting maintenance procedures for locomotives, and in several specialties for which we do not have enough human experts. The commercial potential of the subject is now being recognized.

In a period of such intense activity, it can be healthy to step back from the day to day excitement of new uses of expert systems, new research results, product announcements, and the formation of new companies in the expert systems area. In this article we look at some of the interplay between principles and practice in expert systems; we argue that it is fruitful to combine principles and practice closely, and that logic programming is a strong candidate for bridging the traditional gap between the two. We set out a particular view of where work on expert systems has come from, what is being done now, and of some of the trends for the future. Because of the scope of the subject, our view necessarily focuses on just some of the trends and achievements.

In the next section we describe the properties that we expect of an expert system. Section 3 sketches some central issues in expert systems from a historical point of view. Section 4 outlines the role of Prolog and logic programming in expert systems. In Section 5 we note that an expert system is only as good as the knowledge it contains, and we describe some methods of knowledge representation. Then, in Section 6, we describe some trends in knowledge acquisition. Section 7 is a summary.

2. What is an expert system?

We expect an expert system to act as an intelligent assistant in some task, or to solve an important problem that would otherwise have to be solved by a human expert. However, as the field has developed, we have come to expect more. Because the expert knowledge that people have may change with time, we would like an expert system to be flexible in integrating new knowledge incrementally into its existing store of knowledge. Indeed, we increasingly expect an expert system to assist us in the transfer of knowledge. We would also like it to be able to show its knowledge in a form that is easy for us to read. If we are to take actions with serious consequences in the real world, based on advice given by an

expert system, then we would like the system to provide explanations of its advice [1]. Because the expert knowledge that people have is often incomplete and only partly understood, we would like an expert system to be able to reason with judgmental or inexact knowledge. This knowledge may be *declarative* (about the nature of a task), *procedural* (about how to do the task, and about how to do it efficiently), or both. Because we would like to concentrate on communicating expertise and knowledge, rather than on programming, we would like an expert system to be able to deal with simple sentences in English or other natural languages. Since there are many open research questions in natural language processing, it is perhaps fortunate that we can build useful expert systems whose natural language technology is shallow yet exact; a system can reason reliably and logically with English sentences whose meanings stem from the way in which they appear in rules, rather than from a detailed dictionary, syntax, and semantics of English. Almost paradoxically, these sentences can even contain judgmental phrases, such as "some evidence," yet can still be used by the underlying, exact reasoning method to yield judgmental conclusions. This simplified approach appears to be surprisingly useful. We can build effective expert systems whose English is limited, and we can add more sophisticated natural language processing as it becomes available.

To summarize, here is what we would like an expert system to be able to do. It

- Should solve an important problem that would otherwise have to be solved by a human expert.
- Should be flexible in integrating new knowledge incrementally into its existing store of knowledge.
- Should assist us to elicit, structure, and transfer knowledge.
- Should be able to show its knowledge in a form that is easy for us to read.
- Should provide explanations of its advice.
- Should be able to reason with judgmental or inexact knowledge about the nature of a task or how to do the task efficiently.
- Should be able to deal with simple sentences in English or another natural language.

There are useful expert systems that do not do all of the above, so these points are best read as ways to measure the sophistication of an expert system. They also serve to outline, in a general way, many of the main research goals in the field. In addition, we can now list the main components that we expect to find in most expert systems:

- Facts, plus knowledge, often in the form of rules.
- An inference engine, for reasoning with the facts and rules.
- An explanation generator.
- A knowledge acquisition engine.
- A (possibly limited) natural language processor.

Perhaps because of the variety of things we expect of an expert system, there are many different names for such systems. They are known as knowledge-based systems, intelligent systems, intelligent knowledge-based systems, and so on. It seems likely that *knowledge system* will become the standard term. (For example, many of the expert system projects at Stanford University are now departments of a recently formed Knowledge Systems Laboratory.) We use the terms expert system and knowledge system interchangeably here.

3. From general to specific, and back again

In this section we sketch some central issues in expert systems work over the last three decades, and we outline how the issues are currently treated.

At a meeting in 1956 at Dartmouth College, John McCarthy coined the term "Artificial Intelligence." At the same meeting a system called the Logic Theorist was discussed. It proved theorems in logic [2] and may be thought of as the first, or at least one of the first, expert systems. It seems useful to outline the subsequent evolution of expert systems in the USA in terms of a *thesis*, an *antithesis*, and a *synthesis*. We then look at the corresponding evolution outside the USA.

The *thesis* states that general methods of expert problem solving can be found, and that these can be made computational and can be applied to many different problems. This approach is represented by the Logic Theorist (mentioned above), by the General Problem Solver [3], and by early work in automatic theorem proving, e.g., [4]. Implicit in the thesis is the concept that the *declarative* and *procedural* aspects of how to solve a problem are independent of the task at hand. The idea is that we should just be able to declare what the task is, and the problem solver should then do it. While this thesis works well in principle, the early implementation efforts were very inefficient in practice. One apparent source of difficulty is that there is no obvious place, within the framework prescribed by the thesis, to put specialized procedural knowledge about how to do each task efficiently. In the absence of this knowledge, the problem solver searches for answers along many blind alleys before arriving at a solution.

This difficulty led Edward Feigenbaum of Stanford University to advocate an *antithesis*. The antithesis states that, rather than looking for generality, we should set out empirically to capture human knowledge and procedures for specific tasks (see Feigenbaum and McCorduck [5] for a recent summary of this view). Essentially, we should be willing to write a new program for each task. This technique led to the first practical expert systems. For example, the

DENDRAL program [6] is a "smart assistant" for a chemist concerned with structure elucidation in organic chemistry. The Meta-DENDRAL program (described in the same paper) is a knowledge acquisition program for DENDRAL. The antithesis approach, and its reduction to practice, has clearly been responsible for the present commercial interest in expert systems. However, the approach is intellectually labor intensive, so far as the acquisition of knowledge is concerned. It is usual for a *knowledge engineer* to study the task at hand, specify appropriate inference engines, and then work with both task experts and programmers to construct a system. Since the conceptual levels dealt with by the task experts and the programmers are usually far apart, success can depend on the skill of the knowledge engineer. Often the experts, knowledge engineer, and programmers must invest years of work in building a useful expert system. Moreover, with the notable exception of DENDRAL, the systems so constructed have not by and large contributed much to our knowledge of the principles of expert systems.

Thus we have a thesis, that general problem solvers are desirable, and an antithesis, that the best practical approach is to build specific systems for specific tasks. The *synthesis* of these two approaches essentially takes the middle ground. The idea is that many tasks have requirements in common, and that these requirements can be met by an expert system "shell," to which we add knowledge about particular tasks. Typical shells, such as Emycin [7] and OPS5 [8, 9], each cover a range of tasks, but no one shell covers them all. Clearly there are variations on this synthesis, one of which is to provide a "toolkit" containing many of the methods used in the various expert system shells. The Expert System Environment/VM [10, 11], KEE [12], and LOOPS [13] systems can be thought of as toolkits of this kind. Since even a toolkit may not support all of the expert systems that we may wish to build, it appears important that such kits be packaged as "open systems," in the sense that the underlying programming language is accessible. It should be easy to write new tools and to link them with the ones provided.

When we described the thesis about generality, we noted that there was no place in the general framework for procedural knowledge about how to do a specific task efficiently. So it is natural to ask how the synthesis avoids the efficiency problems that led to the suspension of the thesis. There seem to be several reasons why individual expert system shells are efficient enough to be useful. First, individual shells do not try to cover such a broad class of problems as the thesis methods, so each shell can be designed to be efficient for the class of problems for which it is intended. Second, the processing speed provided by the underlying hardware has increased significantly since the time the thesis was first proposed, although not enough for direct support of the thesis. Third, although we like to keep the knowledge about a task as declarative as possible, in practice we often program some efficiency know-how into it. However, this last step tends to make the knowledge harder to examine and change. It is usually better to put as much of the procedural knowledge as possible in the inference engine.

In its most general form the thesis calls for more computational power than we can supply, but it allows us in principle to just specify a task to be done, so that we do not have to give a procedure for how to do the task. Thus the thesis is computation intensive. The antithesis, on the other hand, is intellectual-labor intensive. Task experts, knowledge engineers, and artificial intelligence programmers have in some cases spent years building a single specialized system. The synthesis, taking the middle ground, tends to reduce the amount of intellectual work in constructing a system, and tends to result in a system that is reasonably efficient on current computers. Work on computer-assisted acquisition of knowledge, which we describe in Section 6, shows some promise of further reducing the intellectual work needed to build a system, without increasing the computer power needed to run the system after it has been built. Interestingly, the more ambitious knowledge acquisition methods call for very large amounts of computing. Often this is worthwhile: We may be willing to compute for days or weeks to build a system, although we often expect fast response from the system once it has been built.

So far, we have outlined some central trends in expert systems in the USA since the late fifties in terms of a thesis, an antithesis, and a synthesis. The thesis was based on mathematical logic, while the antithesis and synthesis dispensed with the logical approach on the grounds of efficiency. Most of the experimental work in all three phases has been done in the language Lisp, invented by John McCarthy at about the start of the thesis period [14], and elaborated through many different versions since. Interestingly, although Lisp is designed for symbol manipulation, it is a *functional* rather than a *relational* (logical) language; various operations for logic, such as unification and search, must be programmed in Lisp when needed.

Outside of the USA, there was a thesis stage concerned with automatic theorem proving, but there was generally much less of the antithesis-inspired activity in building individual systems. Around the mid-70s, the synthesis happened in the USA; several expert system shells with good but not completely general coverage appeared on the scene. In the meantime research elsewhere also led to some new logical techniques for expert systems, and we discuss these next.

4. Prolog and logic programming

At about the same time that the synthesis made itself known in the USA, a new language called Prolog was designed in Europe [15–17]. Although Prolog was first implemented to support natural language processing [18], it works essentially as an efficiently executable part of mathematical logic, so it

is of interest for expert system reasoning, as well as for other tasks.

By 1981, Prolog was adopted by the Japanese as the basis of their Fifth Generation Project [19]. The technical aims of this ten-year project include fundamental work on both software and hardware for advanced knowledge bases. In 1984, at the end of the first phase of the project, substantial progress was reported on a Prolog-based workstation, a database machine, studies of parallel machines to support Prolog, and in some experimental expert systems [20].

Prolog has a certain hybrid vigor, in that it contains some declarative features from computational mathematical logic [21, 22] and some procedural aspects from conventional programming. Thus it is called a *logic programming* language, and, like the expert system shells and toolkits, it occupies the middle ground between our general thesis and specific antithesis.

Prolog is somewhat weighted towards the thesis idea and towards generality. Many of the mechanisms one needs for an expert system shell are in the language, yet the language has very few features in the conventional sense. (For example, Prolog has a generalized logical pattern match called *unification,* but it is built-in, and one usually does not see it or call it explicitly when writing a program.) Perhaps because of the lack of conventional features, Prolog has a very elegant and practical semantics that seems to be the key to much of its appeal. For example, it is possible to say what a well-written Prolog program should compute, independent of any particular interpreter or compiler, and it is possible to say this precisely without getting into undue complexities. One analytical tool for doing this is model theory in logic [23, 24], which turns out, in many examples, to be quite close to the intuition that most people have about what a program should do. As a very simple example, the Prolog program

```
route(X, Z) ← road(X, Z).
route(X, Z) ← road(X, Y) & route(Y, Z).
road(b, c).      road(c, d).      road(d, e).
```

says that there is a route from X to Z either if there is a road from X to Z, or if there is both a road from X to Y and a route from Y to Z. It also says that there are roads from b to c, c to d, and d to e. The following is a model of the program:

```
road(b, c).   road(c, d).   road(d, e).
route(b, c).  route(c, d).  route(d, e).
route(b, d).  route(c, e).  route(b, e).
```

This is a model in the ordinary sense that every relevant instance of each rule in the program evaluates to *true* in the model. It is also just the collection of commonsense consequences of the rules and facts in the program.

Because of its relation to mathematical logic, Prolog has a theoretical basis [25–27] as well as a body of associated practical experience. In fact, the theory and practice can be made to mesh quite well [28, 29], and can be usefully extended to include relational databases [30, 31].

One consequence of the simple semantics of Prolog is that it works very well as its own metalanguage. That is, it is rather easy to describe the semantics of Prolog in Prolog. While a Prolog interpreter written in a conventional language may occupy upwards of 10 000 bytes, an interpreter for most of Prolog can be written in Prolog in about 100 bytes (less than half a printed page). This may seem to be just an academic curiosity, until we realize that many inference engines for expert systems are somewhat similar to the Prolog inference engine. For example, by making relatively minor changes to a Prolog interpreter in Prolog, we get the core of the Emycin inference engine; by making some other changes, we get rules by which each object belonging to some class inherits the general properties of the class unless otherwise stated. We pay an efficiency price for this conceptual elegance, since the inference engines that we build in this way are generally slower than if they had been written directly. However, when supported by a fast computer, the performance is adequate for many tasks, the flexibility advantages are overwhelming, and it is possible to compile some aspects of the inference engine if necessary [32]. The inference engine consists of Prolog rules that function as metainformation; they are rules about how to use the rules for particular tasks. If we have procedural information about how to do a range of tasks efficiently (e.g., always use a relevant fact before trying a rule), then we can encode this in the metarules that define the inference engine. Thus, in the metalanguage approach to expert systems in Prolog, there is a natural place for procedural knowledge, and the task knowledge can remain largely declarative. If our target is high performance on small machines, then we can first make a metalanguage prototype of the inference engine on a large machine and then rewrite or compile it as necessary.

We mentioned that Prolog is a simple language with few features. In fact, the main part of Prolog lacks many of the standard constructs of other programming languages. For example, destructive assignment (as in X:=5) is rarely used in Prolog, whereas modification of a working memory is a central feature of OPS5. This tends to guide the programmer away from machine-level state-transition programming and towards a more declarative style. Simultaneously, it appears to make pure Prolog (like pure Lisp) more suitable for implementation on parallel hardware [33], and parallel hardware now seems to be promising for the support of some of the more ambitious tasks, such as real-time control, that are proposed for expert systems. Some expert systems for real-time control can be built by very efficient programming on a sequential machine [34], while others

appear to need more: specifically, the kind of underlying performance that is expected from parallel machines.

So far, we have sketched some of the issues in expert systems over the last three decades, both inside and outside the USA, in terms of a *thesis* about generality, an *antithesis* about specialization and efficiency, and a *synthesis* containing a partial return to generality. We have touched on a spectrum of activities from general to specific, and from theoretical to empirical. An expert system is only as good as the declarative and procedural knowledge that it contains. In the next sections, we look at knowledge representation and at knowledge acquisition.

5. Knowledge representation

In order to use knowledge in a machine, we must first choose a way of representing it. We need a notation that supports what we expect an expert system to do. The notation should make it easy for us to add and change knowledge, should be easy for us to read, and should support explanation generation. In addition, the notation should suggest ways in which it will be used and should allow us to write down different methods of use. The notation should also encourage us to separate declarative knowledge (what a problem is about) from procedural knowledge (how to solve the problem), yet it should support efficient problem solving. Thus we can use these criteria to estimate how useful conventional programming is for expert systems and to compare the various knowledge representations that are used for expert systems.

When we write a program in a conventional language, we are writing down knowledge about how to do a task. When we load the program into a computer, the computer could be said to acquire knowledge. However, it may be quite hard to add to the knowledge or to change it. It can also be hard for a person other than the author to read and understand the program, and it is rare for ordinary programs to provide explanations of what they do. On the plus side, ordinary programming notation does suggest how it is to be used, since the meaning of a program can (in principle) be worked out in terms of how it changes the state of the computer it runs on. Yet, most conventional programming languages encourage us to mix declarative and procedural knowledge so much that it can be difficult to separate the two. On the plus side again, the knowledge can often be used very efficiently.

The kind of task for which we build an expert system is, almost by definition, complicated. When we describe a task in English, we usually keep the description to a readable length by relying on knowledge that the writer and the reader already have in common. If the task is simple, it is usually feasible to write a suitable program from the description, although there may be many suitable programs. However, if the task is complicated and specialized, we cannot rely so much on shared knowledge. An English description of readable length is no longer enough, while if we make the description long and pedantic, we get lost in the details. So there are several notations for knowledge that stand somewhere between English and programs. They are useful to the extent that they allow us to write down knowledge in a form that both people (who need not be programmers) and computers can use.

Some of our methods for representing knowledge for expert systems make use of the notion of a *hierarchy*, in which lower items are normally assumed to inherit some of the properties of higher items. For example, if a manager in a company is interested in expert systems, then we assume that the people who work in his or her group have the same interest, unless we are told that one of them specializes in hardware. Some notations for knowledge are often used without explicit reference to a hierarchy, for example, *rules* and *nets*. Others, such as *frames* and *objects*, are centered around the hierarchical notion. Although the various notations have been designed separately, and have been used for many different purposes, there is increasing evidence that each one can usefully be written down in logic. We look at some of this evidence next.

Knowledge is represented as *rules* in systems such as Emycin [7] and Syllog [30]. For example, an Emycin-style rule reads

```
if "plant is wilting" and
    not "leaves have yellow spots"
        then "there is not enough water" : 60.
```

The number "60" indicates that we have 60% confidence in the rule and is used in a numerical form of judgmental reasoning. A Syllog rule reads

```
site eg_number has eg_type rock in suitable form
eg_group fossils have been found at site eg_number
eg_group fossils are characteristic of the eg_p period
known reserves in eg_type rock from the eg_p period
-------------------------------------------------------------------------
some evidence for oil at site eg_number
```

Here, the conclusion below the line is established if all of the premises above the line are true. The "eg_" items are variables that make the rule general.

In Emycin, judgmental reasoning proceeds by chaining rules together to form deductions, and by using a built-in algorithm to combine the numerical confidences at each step in the chain. Thus it is a design assumption that there should be some way of assigning confidence numbers to rules so that the built-in algorithm will give reasonable answer confidences. The algorithm is a part of the Emycin shell, and most of its operation is hidden from the user. In writing a knowledge base it is not always easy to assign suitable numbers to rules, perhaps because one cannot easily tell without experimentation just what their effect will be on the answer confidences. [35] describes an interesting approach to this problem.

In Syllog, there is no built-in algorithm for judgmental reasoning. When needed, judgments are written and used explicitly (and usually symbolically) rather than implicitly and numerically in the Emycin style. For example, the phrases "suitable form" and "some evidence" in the Syllog rule above carry judgment. If we carry the judgmental reasoning explicitly in this way, we are not forced to assume that there is a uniform method (e.g., Bayesian) for weighing evidence; the way in which evidence is weighed may be quite different from rule to rule. Although the lack of a built-in algorithm for combining numerical confidences appears at first sight to be a disadvantage, it seems very natural in practice to use judgmental English phrases in the syllogisms, and to show in explanations how the judgments are combined at each step.

For the Prospector system [36], knowledge is written down as a *net* with nodes and arcs. The nodes represent geological evidence or hypotheses, and the arcs represent causal linkages among the nodes. We can think of part of a net as represented inside the computer by the logical facts, such as

```
arc(favorable_level_of_erosion,
    favorable_environment, 5700, 0.0001)
arc(preintrusive_throughgoing_fault_system,
    favorable_environment, 5, 0.7)
```

This describes two arcs in a net. The first arc is from a node called "favorable level of erosion" to one called "favorable environment," and it bears numbers for judgmental reasoning. The second arc is similar, and both arcs can provide evidence to support the conclusion "favorable environment." For another method of representing knowledge in networks, see [37].

We can think of a *frame* [38] as something like a form that we can fill in, which may have a place in a hierarchy of forms. For example, a form about a kind of house might be filled in like this:

```
house has
    street_name : main
    number_of_bathrooms : 2
    wall_color : white
```

Then we might fill in a form about a more particular kind of house like this,

```
type_A_house has
    number_of_bedrooms : 8
```

and about an individual house like this,

```
house1 has
    street_name : delaware
    wall_color : green
```

Now given that house1 is a type A house, and that a type A house is a kind of house, we can reason that house1 has 8 bedrooms and 2 bathrooms, and that, exceptionally, its walls are green rather than white. The notation is a succinct one for describing many houses, provided that the houses can be grouped according to common features.

An *object* can be thought of as a process or agent that receives a message, changes its internal state, and sends out a message [39]. An object can also be used as a template to generate other objects with similar properties. Objects form an intuitively appealing way of representing knowledge in situations where there are many agents (say, people and computers) that work together on a task. In fact, there is a language called Smalltalk [40] that is based on objects. In our frames example, we looked at the description of a house, and if we did not find a property there, we looked at the descriptions of the kind of house. Similarly, if an object receives a message, it may handle it according to its own procedure, or it may look among more generic objects for a suitable procedure. For example, Shapiro and Takeuchi [41] use logic to describe a screen management program for a computer terminal. Windows on the screen correspond to objects that can send messages to one another; a window with a caption is a special case of a window, which in turn is a special case of a rectangular area of the screen.

We have seen that symbolic knowledge can be represented in several ways, including *rules*, *nets*, *frames*, and *objects*, and we have argued that logic provides a natural common notation. Many kinds of knowledge that are largely symbolic contain a few numbers, as in our examples above. Sometimes it is useful to express knowledge, particularly knowledge about how to search a space of possible actions for the best action to take, almost purely numerically; see [42]. In the numeric case logic is also a strong candidate notation. However, numeric knowledge is traditionally written down as functions rather than as logical relations—that is, there is an implied direction of computation from input numbers to an output number. Writing the same knowledge in logic allows us, in principle, to also specify the output and generate the corresponding sets of inputs. This generalization is very powerful and often works directly in Prolog for the symbolic case. However, in the numeric case Prolog must usually be supported by extra knowledge about how to solve equations. Thus the ability to treat symbolic knowledge reversibly is to a large extent built into Prolog, while the same ability for numeric knowledge must in most cases be added by extra programming. For example, in [43] it is shown how to isolate a variable in an equation symbolically (and automatically) so that one can then use the equation numerically in the usual way.

We have looked briefly at some of the useful ways of representing knowledge in a computer, and we have argued that logic is a useful common notation (see also [44, 45]). Next we shall turn to methods for transferring knowledge

from people to the machine. Before doing so, however, it is important to note that a choice of representation sets the framework in which the machine acquires knowledge. An unsuitable representation can make the task very difficult, while a suitable one can allow acquisition techniques that work well for one task to work for an entirely new subject; see, e.g., [46].

6. Some trends in knowledge acquisition

As we mentioned, expert systems are also known as knowledge systems, and it is clear that they are useful only insofar as they contain knowledge. For a person it is an open question how much knowledge is innate and how much is acquired. For a computer, however, the answer is clear. All knowledge must be acquired, since nothing significant is built in during manufacture.

If we think for a moment about human learning, it is clearly difficult for a person to acquire a full slate of commonsense and specialized knowledge; it takes about a quarter of a lifetime, and this fact may influence our research approaches to machine learning [47]. We can calculate that, roughly speaking, a person may be able to change the information in his or her brain at a rate (in bits per second) that is at least comparable to, and may be much greater than, the rate at which the fastest current computer can change its information. So, if the human potential is in fact used for learning, and if artificial methods are no faster than the natural ones, we may expect the task of general knowledge acquisition for computers to be very difficult. It could require substantially faster computers than we now possess.

However, it already seems clear that we can write useful knowledge acquisition programs that can help us to build expert systems. We can set up some useful forms of machine learning, although what is currently practical falls far short of the kinds of learning that people can do. Once we have chosen to represent knowledge so that it is easy to examine and change, we can set up several methods by which a machine can acquire it. It is useful to group the methods like this:

- Learning by being told.
- Learning by induction from examples.
- Learning by observation and discovery.

In the following subsections we look at each of these methods of knowledge acquisition.

- *Learning by being told*

This is the simplest form of knowledge acquisition, and it can be surprisingly useful. We simply tell the system facts and rules about the task at hand. For example, in the knowledge system shell Syllog [48], we can simply add English-like syllogisms that contain knowledge about a particular subject, such as airline reservations or manufacturing planning [49]. We can easily examine the knowledge, and when the system answers a question, it can also provide an explanation of its answer. The implemented Syllog system does partial checking of the incoming knowledge. Checking can also be done in Emycin-style systems that use numbers for judgmental reasoning [50].

We can make learning by being told more helpful, and thus hopefully use up less of the time of a human expert, by having an expert system assist in pinpointing faulty or missing rules. This process has been called "interactive transfer of expertise" [51]. Once a faulty rule has been found, it can be specialized (e.g., by adding a premise) or generalized. If a rule is missing, it can be very useful if the system can suggest what kind of premises and conclusions should be added. Shapiro [52] describes some methodical techniques for computer-assisted debugging of Prolog programs that essentially interview a person at the terminal to find out, by example, what he or she has in mind. Notably, Shapiro's techniques are not only implemented as a program but also have a clear theory, based on models in logic, to back them up. The techniques extend to inductive inference of programs from examples, as discussed in the next subsection.

In principle we would like an expert system to check what it is told thoroughly. Basically, there are three things that can happen when we present a system with a new item of knowledge [53]:

a. The new item is already deducible from the current knowledge. Efficiency issues aside, we may set up the system to reject the item, with a suitable message to the user.
b. The new item is inconsistent with the current knowledge. Either the item is to be rejected, or the knowledge is to be changed before the item is accepted. Rejection is straightforward. Alternatively, we can set up the system to add the item automatically as an exception to the knowledge, or to hold a dialog about what action to take with the person providing the knowledge.
c. The new item is neither deducible from the current knowledge, nor inconsistent with it. The item is added to the knowledge. However, there may now be redundancies, e.g., if we have added a general rule that covers a number of facts. So we may wish to edit these out of the new knowledge.

We have spoken so far about adding an item to a knowledge base. In relational database systems [54] items are also deleted, and this presents some additional theoretical and practical problems. Indeed, Kowalski and Sergot [55] propose that, rather than deleting an item, one can add to the knowledge base the fact that the item ceased to be true at

some point in time. This is intuitively appealing, perhaps because people do not automatically forget past facts or knowledge. It also has far-reaching consequences for database normalization theory. However, in practice it needs very large archival memories, and it may have to be combined with some measures to limit the amount of knowledge that is stored.

The steps a–c above can be written as a logic program that assimilates new information into a knowledge base. The declarative version of the program is short and clear, about a page of Prolog, and it works well for small knowledge bases. However, it is not efficient in general for large knowledge bases. There are some techniques available for writing a longer, more procedural logic program that is more efficient [56]. However, for certain knowledge bases, some of the checking that the program does is combinatorially hard.

In this situation the natural choice is to only do partial checking, or (equivalently) to run the full checking program with a resource bound on steps a and b, and to do step c if the resources are used up. This can result in an inconsistent knowledge base. In principle, any answer to any question can be obtained from a logic knowledge base that is inconsistent. Fortunately, most logic programming interpreters and compilers, including the Prolog and Syllog inference engines, impose relevance conditions that prevent a local inconsistency from causing a global one.

So there is really a continuum here. Some knowledge systems provide no checking of the incoming knowledge at all, placing the entire burden on the users. Some, such as Syllog, do some checking, thus moving part of the burden from users to the machine. In future, additional machine speed, as well as more efficient methods, should allow us to increase the amount of checking that is done by the machine. We mentioned that, even with a consistent knowledge base, it is important that a system be able to explain the answers that it gives to questions. In the absence of full checking, explanations also give us a way of verifying the knowledge while a system is being built and when a system is in use.

• *Learning by induction from examples*

It is now generally accepted that people often have expertise that they find difficult to write down explicitly. A person may be very good at a task, but may find it difficult to tell someone else, or a knowledge system shell, how to do the task. One reason for making the language of a system shell declarative and English-like, and for providing explanations, is that some experts may actually experiment directly with the shell and thus may be able to make their implicit knowledge explicit.

On the other hand, an expert can usually provide a wealth of examples about how to do a task for which we would like to build a knowledge base. A particular experimental study [57] found that a knowledge base induced automatically from examples of expert behavior gave better advice than one that was built "by being told" by the expert. In general, the difficulty is to make the leap from examples containing some underlying pattern to general rules that summarize the examples and are capable of dealing with new examples that have not been seen before. Logically, this is not a deduction (given K, and that K implies E, conclude E), but an induction (given E, find a "suitable" K such that K implies E). Here, the examples are represented by E, and the knowledge base that is to be found is K.

In thinking about criteria for "suitable" knowledge bases K, we can immediately rule out two particular inductive inferences from the examples E. A case that is much simpler than a real knowledge induction problem helps to illustrate this. Suppose we are given as examples E just the numbers

1 4 9 16 25 36 49 64

and we are asked for a K which generalizes this. Most people would say

K consists of the squares of the integers.

Intuitively, this is suitable. However, it is much easier to recognize suitability in particular cases than to capture it as a general concept. At least we can avoid two kinds of induction that are almost always unsuitable. The first is that K is just E, i.e., the induced knowledge base is just a look-up table of the examples that have been seen. This is unsatisfactory because we usually want a knowledge base that is smaller than the examples from which it was induced, and because no new examples can be handled. (In our simple case above, K would not imply 81.) The second inductive inference is that K is the most general possible knowledge base that implies E; that is, it implies everything in the domain from which E is a sample. (In our simple case, K would consist of all of the integers.) Although such a knowledge base is often much smaller than the examples, it tends to be vacuous, in the sense that it indicates that anything is possible.

Shapiro [52] shows how to induce Prolog programs automatically from examples of their desired behavior. There can be many programs that cover a set of examples, but which one to choose is not the only concern. The number of examples needed to produce a program and the computer time needed for the induction process must also be weighed. Shapiro gives an induction algorithm that can be used with different search strategies. In one case a strategy that needs many examples yields a short program, while a strategy that sometimes needs fewer examples can (with an adverse ordering of the examples) yield an arbitrarily long program. Kitakami et al. [58] describe a way of combining knowledge acquisition by being told with the inductive approach of Shapiro.

Several criteria have been used to strike a balance between small, overly general inductions on the one hand, and large, overly specific ones on the other hand. In the Ockham system [59], a Bayesian measure was used to steer a search through a space of causal graphs. Quinlan [60] introduces a system in which a decision tree is induced from examples, and different trees are ranked by the amount of information that is gained from the questions asked. Relational databases can be compressed, as described in [61], by replacing several entries by the name of the class to which they belong. For example, "cat" and "dog" could be replaced by "mammal." However, they could also be replaced by "pet." The resulting compressed databases are inductive generalizations, and their succinctness can be compared by running the compression backward and seeing how close we get to the original database. Mitchell's version space technique [62] provides a compact representation for all of the inductive hypotheses that are compatible with a collection of examples and nonexamples of a concept. The idea is to store the most general hypotheses that do not imply any nonexample, and the most specific hypotheses that do not exclude any example. The admissible hypotheses then lie in a "version space," which is partially ordered by generality, between these extremes. If a balance criterion is added, then the hypotheses that satisfy it can be found from the version space.

We noted in Section 5 that there are many different representations of knowledge, but that logic is a useful common notation. As we have seen, there are many different criteria for judging the quality of an inductive inference. It appears that further empirical work is needed to relate these criteria and to try to find some useful common ground amongst them.

- *Learning by observation and discovery*

So far, we have looked at learning by being told and at learning by induction from examples, both of which are techniques for acquiring knowledge for subsequent use in an expert system. In this section, we look at the extent to which a system can be said to discover new knowledge.

In learning by being told, the system is given facts plus general rules about how to use the facts, which together amount to a knowledge base K that implies the advice that we wish it to give. Generally, K is very much smaller than an explicit listing of the advice.

In learning by induction from examples, the knowledge acquisition part of the system is given a collection of examples of good and bad advice (so labeled) from which it should induce a knowledge base K that implies the good advice, refrains from implying the bad advice, and gives correct advice on examples that are not in the original collection. In order to do this, the knowledge acquisition engine needs a guidance criterion (let us call it G) to choose a "good" knowledge base K that implies sensible consequences.

For learning by discovery, we equip the knowledge acquisition system with a minimal initial knowledge base k, some operators O for adding information to k, and some guidance G about what operators to apply in what circumstances. We then let the acquisition system run, applying O to k, guided by G. If we have chosen k, O, and G well, the system will discover a larger knowledge base K containing some conjectures that can turn out to be useful. For example, in Lenat's AM system [63], k consists of some simple non-numerical knowledge about mathematical sets; O contains some operators such as

if $f(x, y)$ is a function in the knowledge base, add to the knowledge base the function $g(z) = f(z, z)$;

and the guidance G is a prioritized agenda.

Equipped in this way, the AM system produced a K containing, amongst other conjectures (not all of which were interesting), de Morgan's laws and the unique factorization theorem, although nothing resembling either of these was present in k, O, or G. In fact, it also made some interesting numerical conjectures that were unknown to Lenat at the time he wrote AM. Unfortunately, when the program moved away from the symbolic domain with which it had been primed, and into the numeric domain, it made more uninteresting conjectures as well.

Lenat then observed that, since the program could make interesting conjectures in a domain such as mathematics, it should also be able to discover useful new guidance heuristics G. This led Lenat to formulate metaheuristics—heuristics about how to find heuristics—such as

if a heuristic is occasionally useful but usually bad, then add specializations of the heuristic,

and even to have the system apply this heuristic to itself.

In learning by being told, the expertise given to the system usually contains rules that are general in the sense that they contain variables. In learning by induction from examples, the knowledge acquisition part of a system will often generalize the examples it is given by replacing constants with variables (perhaps with range restrictions). A notable feature of learning by discovery is that variables ranging over function or predicate names are sometimes used; that is, viewed as logic, the process is second order. For example, Lenat's operator that specializes $f(x, y)$ to $f(z, z)$ is intended to apply to *any* function of two variables. Another study [64] describes a program that discovers the geographical concept of an equator. The program is primed with some geographical facts and also with some second-order logical knowledge. McCarthy [65] gives a second-order logic technique, called circumscription, that can be used to discover some general properties of a situation. Thus, while our first two kinds of learning can often be stated in first-order logic, there is some evidence that guidance for

discovery is naturally expressed in second-order logic. Since we have little experience so far in practical computing with second-order logic, it seems likely that simulation using metalanguage techniques in first-order logic [53] will be useful for learning by discovery.

7. Summary

We have described some of the interplay between principles and practice in expert systems.

Behind early work in the field, there was a thesis that general problem solving engines could be built, and that it would suffice to add declarative knowledge about a task to an engine to get an expert system. However, the thesis did not have a niche for procedural knowledge about how to do a task efficiently. Consequently, the thesis led to very slow computation on early machines and still cannot be supported directly on current computers.

There followed the Feigenbaum antithesis that we should collect declarative and procedural knowledge for a specific task and be willing to write a new expert system for each task. The antithesis has the first practical expert systems to its credit and has been mainly responsible for the current commercial interest in expert systems. However, the antithesis approach is intellectually labor-intensive for task experts and for knowledge engineers whose job it is to collect and codify the expertise.

As computer power increased, a practical synthesis of the two earlier approaches appeared. The common elements of various ranges of expert system tasks are collected into expert system shells. These shells each cover a range of tasks efficiently, but no one shell is as general as the extreme form of the original thesis. Together, the shells support a wide range of tasks, so they can be collected together into toolkits for building expert systems. Many of the shells and most toolkits must still be primed by knowledge engineers, but the time taken to build an expert system is reduced.

The original thesis had a strong flavor of mathematical logic to it. About the time of the expert system shell synthesis, the Prolog language for logic programming appeared. Prolog is a restricted form of logic for which there are efficient interpreter/compilers. It appears to be very suitable for implementation on parallel machines. Although it is a language, Prolog is at a conceptual level much closer to a shell or a toolkit than to a conventional programming language. Prolog is very simple, with few language features, but an important technique called metalanguage programming allows us to tailor expert system shells. This technique provides separate niches for procedural and declarative knowledge, and the separation further eases the knowledge engineering task. Because of the connection with logic, Prolog allows us to build useful bridges from theory to the actual practice of building and using expert systems.

Even with the separation of declarative and procedural knowledge, it is clear that more can be done to ease the transfer of knowledge from human experts to expert systems. A good choice of knowledge representation is important, and, now that it can be supported efficiently enough, logic seems to be a highest common notation for the representations in current use. An expert system can acquire knowledge by being told (in which case we like it to help us by checking the consistency of what we say), by induction from examples, or by semi-autonomous learning from observation and discovery. A problem with induction from examples is that we usually cannot get explanations automatically from an induced expert system. Learning by being told or by induction from examples is normally a first-order logic activity, while discovery is often guided by statements in second-order logic. However, metalanguage techniques allow us to handle some second-order logic at the first-order level.

As expert systems become more useful, it may be good to keep in mind that there are several levels of detail at which they can be built. At the least detailed level, we supply simple rules and facts that describe English (or other natural language) abstractions that people use to make decisions. As we go further into detail, we may wish to simulate certain theories about our own cognitive processes (e.g., by using situation-action rules) or we may wish to simulate approximately some aspect of the real world (e.g., a mechanical device that the expert system is to diagnose). At the limits of feasible detail, we may actually simulate events in our brains at the level of individual neurons, or the detailed functioning of a mechanical device that we wish to diagnose. We have achieved most of our expert system successes so far with very little detail. It is a fascinating question whether this trend will continue, or whether we shall find it more useful to be more detailed in future.

The economically successful expert systems so far have each addressed a specialized task, such as finding mineral deposits. It is worth noting that most human experts specialize too, in professions such as geology. However, each human expert also has commonsense knowledge about the world in general and knows how to consult experts in subjects other than his or her own. While we each find it easy to do commonsense reasoning, no one so far has produced an account of *how* we do so (or even of the declarative knowledge we might be using) that is sufficient for us to write a "commonsense expert system." We can speculate that, as in the case of specialized expert systems, good progress will be made where there is an interplay between theory (influenced by logic) and specific empirical work in building prototype commonsense systems.

Acknowledgments

It is a pleasure to acknowledge that many conversations with colleagues have helped to shape this paper. Three anonymous referees, Se-June Hong, and John Sowa have kindly taken the time to make detailed comments. Of

course, debit for any remaining shortcomings belongs to the author.

References

1. D. Michie, "Game Playing Programs and the Conceptual Interface," *ACM Sigart Newsletter*, No. 80, 64–70 (1982).
2. A. Newell, J. C. Shaw, and H. A. Simon, "Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics," *Computers and Thought*, E. A. Feigenbaum and J. Feldman, Eds., McGraw-Hill Book Co., Inc., New York, 1963, pp. 109–133.
3. A. Newell and H. A. Simon, "GPS, A Program That Simulates Human Thought," *Computers and Thought*, E. A. Feigenbaum and J. Feldman, Eds., McGraw-Hill Book Co., Inc., New York, 1963, pp. 279–293.
4. C. Green, "Application of Theorem Proving to Problem Solving," *Proceedings of the First International Joint Conference on Artificial Intelligence*, 1969, pp. 219–239.
5. Edward Feigenbaum and Pamela McCorduck, *The Fifth Generation*, Addison-Wesley Publishing Co., Reading, MA, 1983.
6. B. Buchanan and E. Feigenbaum, "DENDRAL and META-DENDRAL: Their Applications Dimension," *Artif. Intell.* **11**, 5–24 (1978).
7. B. Buchanan and E. Shortliffe, *Rule-Based Expert Systems*, Addison-Wesley Publishing Co., Reading, MA, 1984.
8. C. L. Forgy, "OPS5 User's Manual," *CMU-CS-81-135*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, July 1981.
9. L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5*, Addison-Wesley Publishing Co., Reading, MA, in press.
10. P. Hirsch, M. Meier, S. Snyder, and R. Stillman, "PRISM: Prototype Inference System," *AFIPS Conf. Proc.* **54**, 121–124 (1985).
11. P. Hirsch, W. Katke, M. Meier, S. Snyder, and R. Stillman, "Interfaces for knowledge-base builders' control knowledge and application-specific procedures," *IBM J. Res. Develop.* **30**, No. 1, 29–38 (1986, this issue).
12. R. Fikes and T. Kehler, "The Role of Frame-Based Representation in Reasoning," *Commun. ACM* **28**, No. 9, 904–920 (1985).
13. M. Stefik, D. G. Bobrow, S. Mittal, and L. Conway, "Knowledge Programming in LOOPS: Report on an Experimental Course," *The Artificial Intelligence Magazine*, pp. 3–13 (Fall 1983).
14. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, MA, 1962.
15. R. A. Kowalski, "Predicate Logic as a Programming Language," *Proceedings of IFIP 74*, North-Holland Publishing Co., Amsterdam, 1974, pp. 569-574.
16. P. Roussel, *Prolog: Manuel de Reference et d'Utilisation*, Groupe d'Intelligence Artificiel, Université d'Aix-Marseille, Luminy, September 1975.
17. L. M. Pereira, F. C. M. Pereira, and D. H. D. Warren, "User's Guide to Decsystem-10 Prolog," *Occasional Paper No. 15*, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1978.
18. A. Colmerauer, "Metamorphosis Grammars," *Natural Language Communication with Computers*, Lecture Notes in Computer Science, L. Bolc, Ed., Springer-Verlag New York, 1978, pp. 133–189.
19. *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Institute for New Generation Computer Technology, Tokyo, Japan, 1981.
20. *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Institute for New Generation Computer Technology, Tokyo, Japan, 1984.
21. J. A. Robinson, *Logic: Form and Function*, North-Holland Publishing Co., Amsterdam, 1979.
22. J. A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle," *J. ACM* **12**, 23–41 (1965).
23. M. H. van Emden and R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language," *J. ACM* **23**, No. 4, 733–742 (1976).
24. K. R. Apt and M. H. van Emden, "Contributions to the Theory of Logic Programming," *J. ACM* **29**, No. 3, 841–862 (1982).
25. J.-L. Lassez and M. Maher, "The Denotational Semantics of Horn Clauses as a Production System," *Proceedings of AAAI Conference*, Washington, DC, 1983, pp. 229–231.
26. J. Jaffar, J.-L. Lassez, and J. Lloyd, "Completeness of the Negation as Failure Rule," *Proceedings of the International Joint Conference on Artificial Intelligence*, Karlsruhe, W. Germany, 1983, pp. 500–506.
27. J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag New York, 1984.
28. D. Brough and A. Walker, "Some Practical Properties of Logic Programming Interpreters," *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Institute for New Generation Computer Technology, Tokyo, Japan, 1984, pp. 149–156.
29. K. R. Apt, H. Blair, and A. Walker, "Towards a Theory of Declarative Knowledge," Research Report, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1985, to appear.
30. A. Walker, "Data Bases, Expert Systems, and Prolog," *Artificial Intelligence Applications for Business*, W. Reitman, Ed., Ablex Publishing Co., Norwood, NJ, 1984.
31. Ghica van Emde Boas and Peter van Emde Boas, "Storing and Evaluating Horn-Clause Rules in a Relational Database," *IBM J. Res. Develop.* **30**, No. 1, 80–92 (1986, this issue).
32. A. Walker, "Automatic Generation of Explanations of Results from Knowledge Bases," *Research Report RJ-3481*, IBM Research Laboratory, San Jose, CA, 1982.
33. H. Diel, N. Lenz, and H. M. Welsch, "An Experimental Computer Architecture Supporting Expert Systems and Logic Programming," *IBM J. Res. Develop.* **30**, No. 1, 102–111 (1986, this issue).
34. R. L. Ennis, J. H. Griesmer, S. J. Hong, M. Karnaugh, J. K. Kastner, D. A. Klein, K. R. Milliken, M. I. Schor, and H. M. Van Woerkom, "A Continuous Real-Time Expert System for Computer Operations," *IBM J. Res. Develop.* **30**, No. 1, 14–28 (1986, this issue).
35. William F. Eddy and Gabriel P. Pei, "Structures of Rule-Based Belief Functions," *IBM J. Res. Develop.* **30**, No. 1, 93–101 (1986, this issue).
36. R. Duda, J. Gaschnig, and P. Hart, "Model Design in the Prospector Consultant System for Mineral Exploration," *Expert Systems for the Microelectronic Age*, D. Michie, Ed., Edinburgh Press, Scotland, 1979, pp. 153–167.
37. John F. Sowa and Eileen C. Way, "Implementing a Semantic Interpreter Using Conceptual Graphs," *IBM J. Res. Develop.* **30**, No. 1, 57–69 (1986, this issue).
38. M. Minsky, "A Framework for Representing Knowledge," *The Psychology of Computer Vision*, P. H. Winston, Ed., McGraw-Hill Book Co., Inc., New York, 1975.
39. C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *Artif. Intell.* **8** (1977).
40. A. Goldberg and D. Robson, "Smalltalk-80: The Language and Its Implementation," Addison-Wesley Publishing Co., Reading, MA, 1983.
41. E. Shapiro and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," *New Generation Computing* **1**, No. 1, 25–48 (1983).
42. J. Pearl, *Heuristics, Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley Publishing Co., Reading, MA, 1984.
43. A. Bundy and B. Welham, "Using Meta-Level Inference for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation," *Artif. Intell.* **16**, 189–212 (1981).
44. Franz Guenthner, Hubert Lehmann, and Wolfgang Schönfeld, "A Theory for the Representation of Knowledge," *IBM J. Res. Develop.* **30**, No. 1, 39–56 (1986, this issue).

45. Jean Fargues, Marie-Claude Landau, Anne Dugourd, and Laurent Catach, "Conceptual Graphs for Semantics and Knowledge Processing," *IBM J. Res. Develop.* **30**, No. 1, 70–79 (1986, this issue).
46. D. B. Lenat and J. S. Brown, "Why AM and Eurisko Appear to Work," *Proceedings of the National Conference on Artificial Intelligence (AAAI83)*, Washington, DC, 1983, pp. 236–240.
47. H. A. Simon, "Why Should Machines Learn?", *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds., Tioga Publishing Co., Palo Alto, CA, 1983, pp. 25–37.
48. A. Walker, "Syllog: An Approach to Prolog for Non-Programmers," *Logic Programming and Its Applications*, M. van Caneghem and D. H. D. Warren, Eds., Ablex Publishing Co., Norwood, NJ, 1985.
49. C. Fellenstein, C. O. Green, L. M. Palmer, A. Walker, and D. J. Wyler, "A Prototype Manufacturing Knowledge Base in Syllog," *IBM J. Res. Develop.* **29**, No. 4, 413–421 (July 1985).
50. M. Suwa, A. C. Scott, and E. H. Shortliffe, "Completeness and Consistency in a Rule-Based System," *Rule-Based Expert Systems*, B. Buchanan and E. Shortliffe, Eds., Addison-Wesley Publishing Co., Reading, MA, 1984, pp. 159–170.
51. R. Davis, "Interactive Transfer of Expertise," *Rule-Based Expert Systems*, B. Buchanan and E. Shortliffe, Eds., Addison-Wesley Publishing Co., Reading, MA, 1984, pp. 171–205.
52. E. Shapiro, *Algorithmic Program Debugging*, MIT Press, Cambridge, MA, 1982.
53. K. A. Bowen and R. A. Kowalski, "Amalgamating Language and Metalanguage in Logic Programming," *Report 4/81*, School of Computer and Information Science, Syracuse University, New York, 1981.
54. E. F. Codd, "Relational Completeness of Data Base Sublanguages," *Courant Computer Science Symposium 6: Data Base Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1971, pp. 65–98.
55. R. Kowalski and M. Sergot, "A Logic-Based Calculus of Events," *Report*, Department of Computing, Imperial College, London, 1985.
56. A. Walker and S. Salveter, "Automatic Modification of Transactions to Preserve Database Integrity Without Undoing Updates," *Report 81/026*, Department of Computer Science, State University of New York at Stony Brook, 1981.
57. R. S. Michalski and R. L. Chilausky, "Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *Policy Analysis and Information Systems* **4**, No. 2 (June 1980).
58. H. Kitakami, S. Kunifuji, T. Miyachi, and K. Furukawa, "A Methodology for Implementation of a Knowledge Acquisition System," *Proceedings of the 1984 International Symposium on Logic Programming*, Atlantic City, NJ, pp. 131–142.
59. A. Walker, "On the Induction of a Decision Making System from a Database," *Report CBM-TR-80*, Department of Computer Science, Rutgers University, NJ, 1977.
60. J. R. Quinlan, "Learning Efficient Classification Procedures and Their Application to Chess End Games, *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds., Tioga Publishing Co., Palo Alto, CA, 1983, pp. 463–482.
61. A. Walker, "On Retrieval from a Small Version of a Large Data Base," *Proceedings of the Sixth International Conference on Very Large Data Bases*, Montreal, Canada, 1980, pp. 47–54.
62. T. M. Mitchell, P. Utgoff, and R. Banerji, "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds., Tioga Publishing Co., Palo Alto, CA, 1983, pp. 163–190.
63. D. B. Lenat, "The Role of Heuristics in Learning by Discovery: Three Case Studies," *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds., Tioga Publishing Co., Palo Alto, CA, 1983, pp. 243–306.
64. W. Emde, C. U. Habel, and C.-R. Rollinger, "The Discovery of the Equator, or Concept Driven Learning," *Proceedings of the International Joint Conference on Artificial Intelligence*, Karlsruhe, W. Germany, 1983, pp. 455–458.
65. J. McCarthy, "Circumscription—A Form of Non-Monotonic Reasoning," *Artif. Intell.* **13**, 27–39 (1980).

Received June 7, 1985; revised August 1, 1985

Adrian Walker *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Walker is manager of principles and applications of logic programming at the Thomas J. Watson Research laboratory in Yorktown Heights. He joined IBM at the San Jose, California, Research laboratory in 1981, and worked on the R* distributed database system and on logic programming and expert systems. He moved to Yorktown in 1984. Dr. Walker obtained his Ph.D. in computer science from the State University of New York in 1974, and held the posts of assistant professor at Rutgers University and member of technical staff at Bell Laboratories, Murray Hill, New Jersey, before joining IBM.