Introduction

Knowledge acquisition has emerged as a key technology underpinning the creation of expert systems. A number of schemes have appeared recently which learn structural concepts directly from examples, providing — or at least promising — an attractive alternative to the time-consuming and costly partnership of knowledge engineer and domain expert in certain areas of expert system development. Such systems create generalized representations of a structure by processing a number of specific examples obtained from an "oracle", either a cooperative teacher or direct experience of an external environment. Some place the initiative for constructing new examples with the oracle; others with the system itself; while yet others are mixed-initiative dialogs.

What is meant by concept learning? Dictionary definitions of "concept" are remarkably vague, but have in common the abstract idea of a class of objects, particularly one derived from specific instances or occurrences. "Learning" is also a very broad term, and denotes the gaining of knowledge, skill, and understanding from instruction, experience, or reflection. Notwithstanding the grand and sweeping nature of these ideas, the notion of concept learning as presently used in AI is much more constricted and specific. It denotes the acquisition of structural descriptions from examples of what it is that is being described. In fact it would be more accurate to call this "inductive modeling", which is the process of reasoning from particular facts or individual cases to derive generally-applicable models. The state of the art in knowledge representation is so crude and rudimentary that fine distinctions between the meaning of words like "concept", "description", and "model" disappear completely. Indeed the variety of different, evocative, connotation-laden words used to describe very simple mechanisms is one of the biggest problems in coming to grips with the field of machine learning. Our approach to this difficulty is to be completely honest about the scope of terms, and this paper makes no distinction between "concept", "description", and "model" (see the Glossary for capsule definitions of key terms used in this paper).

Concrete examples of things which might be learned by methods discussed in this paper are:

- the concept "nice", where a dish is nice if it does not contain onions, garlic, or cabbage;
- a description of what it means for a poker hand to be a royal flush;
- the conditions under which it is effective to integrate by parts when evaluating a difficult integral;
- a pallet-stacking procedure for a robot;
- the fact that all numbers with (exactly) three (different) divisors are squares of primes.

While these examples are stated informally, what is learned in each case will be a formal description. It might be a rule which distinguishes between an example and a non-example of the concept, or a specification of a procedure, or the annotation of a property in a frame description. Any concept-learning program will have a description language in which examples can be couched, and a generalization language in which concepts are expressed.

The five examples above are intended to convey to you a general idea of what this paper means by "concept learning". In an (admittedly somewhat weak) sense, you yourself have just learned a new concept by example! What you have gained is knowledge in the sense that you can use it in a rich variety of different ways: apply it to a particular case, reflect on it, criticize it, tell it to a friend, relate it to other ideas, etc. You have clearly *not* merely created a formal description! And this is where you differ from a computer program for concept learning. For a person, learning is not simply a matter of acquiring a description, but involves taking something new and integrating it fully with existing thought processes. To suggest that one does less is to insult their intelligence.

As one of the examples suggests, we include procedural knowledge within the ambit of concept learning. One may cavil at the thought of a procedure such as, say, "how to boil an egg" being described as a concept. However, modern computer science has successfully blurred the distinction between the dynamic, executable procedure and the static, descriptive specification. For example, the idea of "what it means for a list of names to be sorted" certainly is a concept, and we know that such a specification can be executed to produce an

(admittedly inefficient) procedure for performing a sort. Indeed, many concept learning systems work within logic programming languages precisely so that specifications can be executed.

This paper examines the fundamental distinctions which underlie the field of concept learning, relates them to plain questions about the capabilities of learning systems, and critically reviews three representative systems from the perspective of practical knowledge acquisition. Figure 1 illustrates the basic concept learning situation, and the next section discusses some practical questions that it seems reasonable to ask of computer programs which learn concepts. These are discussed under the headings of learning power, teaching requirements, and concept representation. Then three quite different systems are presented to illustrate the state of the art. The first provides an economical way of representing partially-learned concepts, the second relies on a teacher to help structure a potentially enormous search space by designing a suitable training sequence, while the third learns robot procedures from example traces. Finally we return to the plain questions introduced earlier and show that the answers are more complex than might appear at first sight.

Plain questions for concept learning systems

Imagine a system which, when presented with examples couched in a certain language, creates from them a generalized description or concept. Questions that one might reasonably ask about such a system include:

- Does it always produce the correct concept given appropriate examples?
- Does it take long in terms of execution time? — number of examples needed?
- Does it correctly classify examples it has been shown before?
- Can it classify new examples, and can it recover from classifying them incorrectly?
- Can it handle erroneous or noisy examples?

- Can the system use negative examples as well as positive ones?
- Can it generate test cases itself (intelligently)?
- Can it explore on its own, creating examples and testing them without human intervention?
- Is it sensitive to the order in which examples are presented?
- Does it need a cooperative, perhaps skilled, "teacher"?

- Can the system tell when it has achieved the generalization which is its final result?
- Is the class of learnable concepts finite or infinite?
- Can it use concepts it has learned as components of the structural description of others?
- Is a concise and comprehensible representation of the concept generated?
- Can the generalization be used in a variety of different ways?

Although there is plenty of overlap, these questions divide broadly into the three groups shown: those concerning the learning power of the system; those relating to its teaching requirements; and those about the representation of concepts learned. We discuss the questions under these headings.

Learning power. Systems which acquire generalizations do so through some sort of searching process. "Generalization as search" was the title of an influential paper (Mitchell, 1982) which equated the process of generalization with searching through a space of structured descriptions. Since then other paradigms have appeared, notably the newly-fashionable connectionist architectures which employ relaxation methods such as simulated annealing to embed new concepts in highly-connected networks; but even these are effectively searching for energy minima using hill-climbing. Groping around for solutions seems central to the learning process, and it is hard to see how a program which avoided search could reasonably be described as generalizing or learning†.

† For a contrary view see Thompson (1984), who describes a self-referencing definition that embeds a new language construct forever

Generalization as search is essentially the same idea as identification by enumeration, a basic methodology for inductive inference introduced by Gold (1967) in a seminal contribution‡. A number of theoretical results, summarized in Figure 2, characterize the ability to identify classes of languages from example sentences, through the theory of enumerability (countability) of various infinite sets. Identifying a particular language from a given class by examining sample sentences is the same thing as identifying a description from a given class by examining samples of what is being described. Although theoretical results of this kind may seem remote from practical concept learning, they do point out that search is at the root of inductive inference. They may not tell us *how* to search, but they confirm that we will have to do so. The crux of the matter in practice, of course, is not whether sets can be enumerated but whether the space of possibilities can be cut down sufficiently to make searching productive.

The "identification by enumeration" paradigm for inductive inference proposes a system that enumerates all possible descriptions in advance. It selects a description and pronounces it to be the target generalization as long as it is consistent with all examples that come along. Once an example is encountered which clashes with the current description, it moves to the next description, checks it against all examples that have been seen so far (for it remembers each example given), and if it passes that test, adopts it as the new target generalization. Suppose that the presentation is such that all possible examples are included. Is such a system is bound to come up with the correct generalization eventually?

If there are only a finite number of different examples corresponding to any given description, the system will be bound eventually to stumble upon a correct generalization (assuming one exists). However, as Gold (1967) showed, if the class from which generalizations can be selected includes all finite languages and at least one infinite one, the answer is no. He proved this by an intriguing method that develops what has become known as a "frustration" sequence of examples. There exists an infinite sequence of finite languages, such that each is a subset of the next, and all are subsets of the given infinite language. For any learner a presentation sequence can be arranged to "fool" it that the language is one of this sequence, and to fool it an infinite number of times, thereby postponing indefinitely any opportunity for it to identify the infinite language. On the other hand a system which begins by guessing the infinite language will never discover its error if the language is in fact finite, since no examples disconfirm its hypothesis. While the result may be considered interesting by theoreticians only, the possible existence of such a frustrating example sequence is a useful tool in the analysis of practical methods.

Other fundamental questions about learning power ask whether a system correctly classifies all examples it has already seen, and whether there exist examples it has not seen that it can classify correctly. The process of simply storing all examples presented is called "rote learning". Most concept learning systems perform some "generalization", in other words, they can sometimes classify unseen examples. However, most cannot recover from situations in which they harbor a misconception about a novel example. Such a system's learning strategy must be extremely conservative to avoid jumping to any conclusion which might turn out to be untenable. It could only generalize in cases where there was no room at all for doubt. Whether it is desirable that a system should remember all examples it has encountered depends on circumstances. In practice it may not matter that examples are forgotten, for phenomena which are worth learning will always recur. All depends on the cost of obtaining examples and the desirability of minimizing learning time.

Finally, it may seem strange to ask whether a system correctly classifies examples it has already seen — surely it is obvious that any system worth discussing will! But the other side of the coin is whether it can withstand noise or other errors presented to it. A system which faithfully reclassifies all examples shown to it will faithfully reproduce all errors too. In fact, none of the systems considered here can recover from errors in the examples presented. This is a major shortcoming and will severely impact their use for practical knowledge acquisition.

---

within a compiler as "as close to a 'learning' program as I have ever seen".

‡ This arose out of early work in the mid to late 1950s; see Ginsburg (1958), Moore (1956), Gill (1961), Solomonoff (1964). Angluin & Smith (1983) provide an up-to-date survey.

Teaching requirements. The paradigm of generalization by enumeration highlights the importance of the sequence of examples presented and the role of the "teacher" who selects them. It is obvious that learning can be prevented by showing only a subset of examples. However, even if all examples are presented eventually, the frustration argument shows that a teacher can have the power to inhibit learning by pernicious choice of the example sequence.

There is a basic distinction between presenting positive examples only, presenting both positive and negative examples, and allowing the learning system to choose examples itself and have them classified by an informant. In the first case, we assume that all positive examples are included eventually. In the second, we assume that every member of some universe which contains all possible descriptions is shown eventually; this allows an informant to be simulated by simply waiting until a selected example occurs in the presentation sequence. Gold (1967) showed unequivocally what one expects informally, namely that the second and third methods are more powerful than the first. They permit primitive recursive languages, which include the context sensitive, context free, and regular languages, to be learned eventually, whereas with positive presentations only, as noted above, the inclusion of even one infinite language in the set of potential concepts can destroy learnability.

Figure 3 illustrates some of these ideas with reference to a simple, artificial learning problem. Suppose the concept "integer between 1 and N" is to be learned, where N is unknown and may be infinite. Figure 3(a) illustrates the set of possible concepts. Part (b) shows the initial part of sample presentations for each of two target languages, L_{100} and L_∞. Positive presentations contain only positive examples (and eventually every positive example will appear); mixed presentations include all positive and negative examples over some universe (in this case, the integers). Part (c) shows the effect of two different enumeration sequences for the two chosen target languages. With the sequence $L_1, L_2, \cdots, L_\infty$ and target L_{100}, identification by enumeration never even considers L_{101}, L_{102} etc. (Supposing it were to, none could be ruled out if the presentation were positive, but any could be if it were mixed.) With target L_∞, positive and mixed presentations become identical (see part (b)). With the same enumeration sequence the target will clearly never be reached. On the other hand a presentation sequence in which L_∞ occurs early on will quickly identify L_∞. With target L_{100}, identification depends on the kind of presentation. Positive examples will never be able to rule out L_∞, so L_{100} will not be identified correctly. In contrast, a mixed presentation will achieve the desired result. Thus the problem can be solved with an enumeration in which L_∞ occurs early on, if the presentation is mixed.

Although it does not increase formal learning power, the possibility of a system constructing its own examples and having them classified by an informant has considerable potential to speed up learning. This potential can only be realized if the system is able to synthesize "crucial examples" which discriminate effectively between alternative hypotheses consistent with the information seen so far. This ability seems to be one of the chief distinguishing characteristics of people who are good learners. To take the generation of examples one step further, if there is an automatic way of classifying examples as positive or negative, a system can attempt to learn autonomously. This paradigm of "learning by discovery" was exploited most effectively in pioneering work by Lenat (1978; see also Davis & Lenat, 1982) on automating the theory formation process in mathematics. The world of mathematics exists entirely within the computer, and conjectures can be supported or refuted without human intervention, by experiment. Despite some later work on the generation of heuristics (Lenat, 1983) and three-dimensional VLSI design (Lenat *et al*, 1982), the power of automated discovery seems largely untapped. Lenat & Brown (1984) provide a fascinating retrospective on this work.

A skilled teacher will select illuminating examples himself and thereby simplify the learner's task. The benefits of carefully constructed examples were appreciated in the earliest research efforts in concept learning. Winston (1975) showed how "near misses" — constructs which differ in just one crucial respect from examples of a concept being taught — could radically diminish the search required for generalization. Confident that its teacher is selecting examples helpfully, a learning system can assume that any difference between an example being shown and its nascent concept is in fact a critical feature.

Recently, Van Lehn (1983) has formalized this notion of a sympathetic teacher in terms of what he calls "felicity conditions", constraints imposed on or satisfied by a teacher that make learning better than from random examples†. One obvious condition is that the teacher should correctly classify examples as positive or negative, and not (intentionally or unintentionally) mislead the student. Another is that if the absence of something is important, the teacher should point it out explicitly. If in the classic example of the concept of "arch" it is necessary that a gap be left between the pillars, the gap should be explicitly labeled in examples. More generally, the teacher should show all work and avoid glossing over intermediate results. Examples should not by coincidence include features that might mislead the learner: one should not illustrate the concept of married *vs* maiden names using a lady whose original family name happens to coincide with her husband's; one should not illustrate the geometric concept of isosceles triangles with ones that happen to be congruent. Finally, the teacher should introduce one essentially new feature per lesson and not try to teach multiple differences at once — a similar condition to Winston's "near miss" approach.

There is a curious anomalous theoretical result which relates to the sequence of examples presented. Gold (1967) showed that descriptions unlearnable from positive and negative examples *can* be learned under a special condition: the presentation sequence must be sufficiently regular. Moreover, positive examples alone suffice! For instance, descriptions from the class of recursively enumerable languages (which includes the primitive recursive ones mentioned above, and more besides) are not normally identifiable even from positive and negative examples, nor indeed using a helpful informant who is prepared to classify arbitrary examples. But a suitable, highly contrived, presentation sequence of positive examples can render such descriptions learnable. In essence the learner can identify the algorithm used to select the examples, rather than identifying the set from which they are drawn. For example, it is much harder to recognize the Fibonacci numbers from the presentation 34, 5, 13, 1, 8, 1, 21, 3, 2, 55 $\cdots$ than from the normal sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, $\cdots$. There is a sense in which it is this theoretical anomaly that is exploited by a good teacher, who renders learnable classes of descriptions that would be unapproachable from a random presentation.

Concept representation. One way of characterizing the concept learning problem is in terms of a language in which examples are couched, a language in which generalizations are expressed, and a method for constructing generalizations from examples (Mitchell, 1982). In the beginning, all possible generalizations are candidate descriptions of the concept. During the learning process, certain descriptions are ruled out because they are too narrow (ie do not encompass some known positive examples), while others are eliminated because they are too broad (ie encompass some known negative examples). Eventually either just one acceptable generalization remains, in which case the system has learned the concept (and knows that it has learned it), or the concept is not representable in the generalization language. As described in the next section, an elegant representational structure is available in which to carry out this process. Normally it is not possible for the system to tell when it has achieved the final generalization unless the presentation includes negative examples as well as positive ones; otherwise very general descriptions can never be ruled out.

This view of learning, called "finite identification", is quite a restrictive one because it implies that the set of concepts that can be learned is closed. The ability to converge on the "correct" generalization and know unequivocally that one has converged, carries with it the millstone that if one is wrong there is nothing that can be done about it! An alternative view is that of identification "in the limit", that is, the system continually proposes hypotheses which state the identity of the target concept and is deemed to have learned successfully if a point is reached after which all its guesses are the same (Gold, 1967). While with finite identification one cannot afford to make any mistake in coming to the final conclusion, with identification in the limit one can err — but only a finite number of times. As noted above, a large class of descriptions (the primitive recursive ones) can be learned by enumeration if the presentation eventually includes all positive and negative examples. However, the learner does not know when he has succeeded.

† It will be interesting to observe whether this work will be used as the basis for future lawsuits against teachers!

Another way of assessing the "open-ness" of a concept learning system is to examine whether it can capitalize on what it has already assimilated by utilizing learned concepts as part of the structural description of others. For example, can a system learn the concept of a *court card* and employ it as a component in the concept of *royal flush*? Extension of the language in which concepts are expressed is a very powerful way to structure learning. Obviously it increases dependence on the felicity (and skill) of the teacher. Would the concept of *royal flush* be learnable if *court card* had not already been taught? In theory, perhaps yes — since eventually enumeration would stumble upon the latter concept, or some equivalent. But in practice? Here the insights yielded by the enumeration paradigm begin to evaporate. Search spaces become enormous, and it is no longer a matter of whether they can be enumerated in principle, but of whether one can ever hope to do so in practice. Systems must take advantage of any heuristics available to focus attention, whether they be the teacher himself or specialized knowledge of the learning domain. And such heuristics seem to be inherently unanalyzable.

One way a concept can be used is to expedite new learning. Another is to classify new examples, whether previously seen (rote learning) or novel (generalization). The version space method described in the next section enjoys the ability to tell, for any example presented, whether it is

- a positive instance of the concept
- a negative instance
- unclassifiable according to the current description.

Categorical yes/no decisions will sometimes be possible even on examples which have not already been encountered. As learning proceeds, the third, unclassifiable, category will shrink, and vanish altogether if convergence occurs. For these laudable virtues the method pays the price of a closed mind! It cannot cope with the more open description languages supported by the general enumeration paradigm.

In particular applications, other ways of using learned concepts will be important too. One question of wide relevance is whether the description generated is compact and comprehensible to a person, so that it can be used by an explanation mechanism. Systems can easily create descriptions that can be used successfully for classification but which cannot be articulated in any satisfactory way. A good example is data compression, where schemes exist that learn the lexical structure of English, from examples, to a degree that lets them predict with remarkable accuracy what will come next (Witten & Cleary, 1987). And yet their internal representations, which occupy many megabytes of storage, are completely useless to a person wishing to examine the basis of their success. The need to simplify automatically-generated descriptions has been recognized by researchers who question whether opaque structures can be described as knowledge, no matter how well they function (eg Quinlan, 1986).

Examples of concept-learning systems

In order to make the distinctions drawn above more tangible, we present three learning systems and discuss their capabilities. The three have been chosen to represent different genres. The first provides an economical way of representing partially-learned concepts, narrowing down the space of possibilities as more examples (both positive and negative) are encountered. The second learns concepts and relations expressed in a logic programming language by searching a potentially enormous space. It relies on a teacher to present concepts in the right order so that it can cut down its search by utilizing previously-learned concepts. The third learns robot procedures from example traces of the robot's actions.

Version space. The "version space" approach to concept learning offers a structure for representing all possible generalizations consistent with the examples seen so far, in an economical way (Mitchell, 1982). Suppose all conceivable generalizations could be listed and stored. When an example is presented, this list could be pruned by removing generalizations that are inconsistent with it. For instance, a positive example may rule out certain generalizations that are too restrictive to encompass it. A negative example may eliminate other generalizations so broad that they *do* encompass it (since it is a negative example, the correct generalization must not). A third possibility is that the example is compatible with all remaining generalizations on the list. A positive example is compatible if all generalizations include it; a negative one if they all exclude it. In either case, nothing can removed from the list on the evidence of that example.

As time goes on, the list of remaining generalizations shrinks. Now the system can classify some examples itself. Given an unknown example, it may be compatible with all remaining generalizations (in which case it must be a positive instance of the concept), incompatible with all remaining generalizations (in which case it must be a negative instance), or compatible with some and incompatible with others. In the last case the system cannot tell whether that example is positive or negative. But if it is told its classification, it can use that fact to eliminate some more generalizations from the list. Ultimately, one hopes, the size of the list will shrink to one generalization, which is the "correct" description of the concept. Then all further unknown examples can be classified as positive or negative.

What can go wrong? An example might be given which eliminates *all* generalizations remaining on the list! Then one can conclude that there was no correct description in the original list — in other words, the concept cannot be represented in the given generalization language. Suppose, on the other hand, more than one description remains on the list for ever. It is a common misconception that this represents a concept which is the disjunction (OR) of these descriptions. But a simple example shows that this is not so. Suppose the two descriptions *is-red* and *is-blue* remain. Then an object which *is-yellow* can be unequivocally classified as a negative instance; but neither *is-red* nor *is-blue* ones can be classified positively. Moreover, presenting one as a positive instance rules out the other description, and if that is subsequently presented as a positive instance the system will conclude (correctly) that the correct description cannot be expressed in the given generalization language. If it is required, disjunction must be represented explicitly within the description language (perhaps by having a description *is-red-or-blue*). Multiple descriptions remaining on the list indicate that the representation language makes irrelevant distinctions. Either examples have not been chosen to cover all features that can be represented (eg neither red nor blue objects are ever encountered) or the description language is intrinsically redundant (eg separate descriptions for *is-red-or-blue* and *is-blue-or-red*).

What version space offers is an economical storage structure within which this description-elimination algorithm can be carried out. Instead of listing all compatible descriptions and striking out those that prove incompatible, one can store instead the upper and lower edges of the compatible set. To do this, descriptions are arranged in a lattice structure according to a "more-general-than" relation. At the top are very general descriptions; at the bottom very specific ones. The compatible set lies between an upper boundary and a lower one. When a positive example is encountered which rules out some too-specific descriptions, the lower boundary is lifted just enough to exclude them. When a negative example is encountered which rules out some too-general ones, the upper boundary is lowered accordingly. When the two boundaries meet at one unique description, this is the correct concept. If they try to cross, the concept cannot be represented. The point is that the entire list of compatible descriptions (which can be very long, especially at first) need not be stored; it is effectively represented by its upper and lower boundaries.

The storage economy achieved by this method depends on the shape of the description lattice. If it is a shallow, flat structure, little is gained. In the limit, if there is no generalization relation between descriptions the lattice degenerates into a simple list of all descriptions at the same level (since they are mutually incomparable) Then the method becomes equivalent to storing the compatible set explicitly. On the other hand a deep, narrow lattice will gain a lot. In theory the lattice could even have infinite depth by containing infinite chains. Then explicit storage of the infinite set would be impossible. Of course, if the correct description lay on an infinite chain one may never converge on it, for an infinite number of examples could be required to eliminate all other

descriptions on the chain. In other words, finite convergence cannot necessarily be guaranteed†.

In practice, systems which work with infinite numbers of descriptions find ways other than version spaces to store concepts. This is because although version spaces can handle lattices of infinite depth, they cannot cope with lattices of infinite width. The reason is that the upper and lower edges of the compatible set are stored explicitly, and if the lattice has infinite width then at some point an unbounded list must be stored. Such a lattice would be necessary if there were an infinite number of descriptions, none of which was a generalization of any of the others — then they would all occur at the same level of the lattice. This seems to be the most common way for infinite description spaces to arise in actual applications.

Hierarchical learning. Our second concept-learning system, MARVIN (Sammut & Banerji, 1983, 1986), learns in a space which includes an infinite number of possible descriptions that cannot be structured into a generalization lattice of finite width. It can make use of already-learned concepts as part of the descriptions of new ones. It relies on the teacher structuring the concepts to be taught into an appropriate sequence, simple ones first. (Otherwise if its current language is inadequate to describe the concept it will form the wrong generalization.) It selects its own examples to test hypotheses.

The best way to explain the system is with an example. Figure 4 shows a transcript of a session adapted from Kolokouris (1986). User input is underlined. The teacher begins with the initial concept *letter*, and gives a couple of examples (lines 1–12). MARVIN forms the concept of a *letter* as either *a* or *b*. (The other 24 could be taught in just the same way, but the dialog becomes repetitive.) Then the concept *letter-list*, a list of *letter*s, is introduced (13–41). The first example given is the null list, which is not decomposable and is therefore simply stored. The second, *[a]*, can he broken down into two components using the decomposition functions *head* and *tail*, with which the system is already acquainted. This leads to the hypothesis that a *letter-list* is something whose head is a *letter* and whose tail is the null list (24–26). MARVIN synthesizes the example *[b]*, using another *letter*, as a test, and it is confirmed by the teacher (line 28). A follow-on hypothesis is that since the null list is already a *letter-list*, a *letter-list* is anything whose head is a *letter* and whose tail is a *letter-list*. This in fact is the correct description. An example, namely *[a, a]*, is generated which satisfies this new hypothesis but is not included in the old version of the concept, and it is presented to the teacher (line 35).

An important strength of MARVIN is its ability to synthesize crucial examples which test differences between its tentative hypothesis and the previous, already accepted, one. However, it jumps to a firm conclusion based on this one example alone, which may be precipitate. Also, it (this version at least) is not very good at generating general examples. When finding a *letter-list* with more than one element it selected *[a, a]* — rather a special case. It would have been better not to choose the same *letter* for each element, and perhaps to create a list of several elements rather than just two. If the intended concept were "lists of less than three letters" or "lists of letters which are all the same", MARVIN would have overgeneralized wildly on the basis of the one test case *[a, a]*. Concepts like these, however, could no doubt be taught by a sufficiently resourceful teacher, using a different teaching sequence.

MARVIN searches all possible ways that existing concepts can be used to create a description for the new one. Suppose it were taught the concept *a-or-b*, *a-or-b-or-c*, etc, and then the new concept *letter-list* were illustrated using *[a]*. At one point during the generalization it would have to consider whether a *letter-list* was an *a* followed by the null list, an *a-or-b* followed by the null list, an *a-or-b-or-c* followed by the null list, etc. Much of the search could be avoided in this case by an appreciation that a generalization hierarchy is involved. For instance, if something is not an *a-or-b* then neither can it be an *a-or-b-or-c*. No doubt the program could be modified to avoid this needless search. However, it is clear that blindly investigating all ways that existing

† Even convergence in the limit cannot necessarily be guaranteed. The example of Figure 3 can be viewed as an infinite generalization chain $L_1 \subset L_2 \subset L_3 \subset \cdots \subset L_\infty$. If L_∞ is the target language then the version space method will not converge, for it will never resolve the ambiguity between L_N, L_{N+1}, $\cdots$, L_∞ for N equal to the largest number encountered as a (positive) example. Negative examples won't help — there are none! But as Figure 3 shows, the problem *is* identifiable in the limit from a mixed presentation using an appropriate enumeration.

concepts could be combined to form new ones is inherently a computationally explosive process.

Lines 43 onward of Figure 4 illustrate the search process further by showing how to start teaching the concept *concatenate*, which takes two lists and joins them into a third (or, equivalently, splits a list into two parts). The first example (line 44), *[]* concatenated with *[a]* gives *[a]*, is generalized incorrectly to

- a *letter-list* joined with *[a]* gives *[a]*. (1)

Since the example generated is rejected (line 60), the system tries to restrict this description to see if it can be patched up (lines 61–80):

- a *letter-list [X]* joined with *[a, X]* gives *[a]*

(rejected in line 70) and

- a *letter-list [X]* joined with *[a]* gives *[a, X]*.

This last attempt is rejected in line 80 through a white lie by the teacher†. MARVIN unfortunately chooses an example with *X=[a]*, and indeed it is true that *[a]* joined with *[a]* is *[a, a]*. But had it chosen a more general example, say *X=[b]*, it would have discovered that *[b]* joined with *[a]* is not *[a, b]*.

At this point hypothesis (1) can be specialized no further and is abandoned. Lines 81–100 investigate the new hypothesis

- a null list joined with *[letter]* gives *[a]*

(rejected in line 90) and its restriction

- a null list joined with *[X]* gives *[X]*.

This is accepted in line 100 — at last MARVIN is making some progress! After another false track (lines 101–110) there is a further correct generalization (lines 111–120) to

- a null list joined with a *letter-list X* gives a *letter-list X*.

Now (line 121) the system has extracted all it can from this example, and the teacher gives another — *[a]* joined with *[b]* gives *[a, b]*.

The key step will be introducing a recursive call to the part of *concatenate* that has already been learned:

- a list with head *H* and tail *X*, when joined to a list *Y*, gives a list with head *H* and tail *Z* where *concatenate(X, Y, Z)* is true.

However, the transcript of Figure 4 stops long before this point is reached. One fruitless generalization is shown (lines 130–137) to indicate that the futile-seeming interactions of earlier on are by no means over. To discover the crucial recursive step even for so simple an example, a great deal of searching is needed. The recursive call involves 3 parameters and 6 components have been identified in the example (eg lines 123–128). There are 120 ways to select and order these parameters. And when other possible transformations are considered as well, the search space explodes combinatorially.

† Had the teacher accepted this example, MARVIN could never have recovered from its misconception that this description correctly characterizes *concatenate*.

The kind of tasks that have been reported for MARVIN (Sammut & Banerji, 1983, 1986) include teaching

- the concept of a *suit* as one of *hearts, diamonds, clubs, spades*
- the concept of a *value* as one of *ace, 2, 3, 4, · · · , jack, queen, king*
- the concept of a *playingcard* as a *value, suit* pair
- the concept of a *pair* as two *playingcards* of equal *value*

- arithmetic concepts, representing numbers as strings of bits, comparing and adding them

- what lists are; how to append them, reverse them, sort lists of numbers using the insertion method

- simple grammars

- the concept of arch (of course!).

Synthesizing what amount to general-purpose computer programs is a heady undertaking involving enormous search spaces and great reliance on the teacher's skills. This illustrates a very tricky point in evaluating concept learning systems. Almost anything may be teachable, but teaching sequences may be difficult and unnatural to construct. One might even regard a description of an algorithm in a programming language as a degenerate case of a teaching sequence and eliminate the need for a learner altogether! The dividing line between teacher and programmer becomes almost indiscernible. For example, Shapiro's (1983) MIS system also searches through a space of programs, but requires the user to specify which subprograms should be called and whether each parameter is input to or output from the procedure. This radically reduces the combinations that need be considered — but the search space is still immense. Another way of automatically assisting those who create programs is to provide debugging tools which use examples rather than a control-based model of computation. One can analyze examples on which a program fails and automatically generate a series of questions like "should function X produce result Y when called with argument Z" to narrow down the error. Such techniques were pioneered by Shapiro (1983) and have been refined in Lloyd (1986).

Given the magnitude of the undertaking it is not surprising that pioneering systems like MARVIN jump to conclusions on scanty evidence, generate atypical examples to use as test cases, cannot reconsider generalizations once they are made, assume a perfect teacher, and generally lack robustness.

Constraint-limited generalization. Andreae's (1984a, 1984b) NODDY acquires robot procedures from examples, complete with control information which is not explicitly present in the examples. It copes with problems of action sequencing, and also handles real numbers representing angles and distances. It makes use of an explicit, pre-programmed, generalization hierarchy, and pre-programmed information on about 30 basic mathematical and set-theoretic operators that may be combined to create complex generalizations.

Examples are traces of the desired procedure. The first trace is taken to be the initial version of the procedure. As further traces are seen they are merged with the nascent procedure, generalizing it in various ways. The system cannot reconsider generalizations it has made, and therefore adopts a conservative policy of requiring considerable evidence before generalizing. The elements which make up the traces are called "descriptors".

The principal problem is to meld two example traces of execution of a procedure into one augmented state-transition representation which encompasses them both. In the first stage of generalization, if two descriptors are identical and unique within each example trace, they are assumed to emanate from the same state. In particular each trace begins with a *start* descriptor which forces the initial states of the two sequences to be merged, and ends with *stop* which merges final states. This builds one state model from the two traces.

However the parts corresponding to each still remain largely separate, since descriptors are rarely identical (apart from *start* and *stop*). The second stage of generalization examines states which are different but whose predecessor states are the same, or whose successor states are the same, and attempts to unify the descriptors associated with each. If they can be unified, then the states are coalesced. Unification is done through the generalization hierarchy, and succeeds if the two descriptors have a common generalization. Since two states may be unified only if their successors or predecessors have been unified, the process proceeds from states matched at the first stage, propagating both backward and forward. Finally, a third stage examines states which are different but whose predecessors *and* successors are the same, and again attempts to unify the descriptors but this time with a more liberal unification procedure. This uses the same generalization hierarchy as before, but involves synthesizing functions which unify parameters of the descriptors. Synthesis is accomplished by searching a function space, possibly utilizing numbers which have been "mentioned" in nearby descriptors as components of the function. This introduces variables into the state-transition representation, effectively creating a form of augmented state-transition network. It is a very expensive process in terms of search time, and is only undertaken when there is strong evidence (identical predecessor and successor states) that the descriptors match†.

Figure 5 shows an example of NODDY in action. The system is to be taught how to get from the start point (–6, 0) to the end point (0, 0), circumnavigating obstacles in the way by retreating perpendicularly to the collision surface, moving a short distance parallel to it, and then continuing. Part (a) shows three example traces, both as diagrams and in the form in which they are given to the system. Primitive actions are *start*, *stop*, and a relative *move* with polar coordinates. Observations are also given to the system: its current position *at*(x, y), and any contact with an obstacle *contact* Ω (Ω being the angle of contact). From these three examples, one with no obstacle and two with single obstacles in different orientations, the system can generate the desired procedure shown in Figure 5(c). Note that the procedure is capable of avoiding several obstacles and even feeling its way round a large obstacle of any (concave) shape.

Inspection of the procedure shows that it uses an additional primitive not in the example traces. The action *move–until–contact–toward*(x, y) moves towards a point specified in absolute coordinates, stopping on contact with an obstacle. It was introduced through the generalization hierarchy shown in Figure 5(d). This hierarchy conveys what the system knows about the various actions available. (Also known are the transformations between Cartesian and polar coordinates, not shown.)

When the first two traces are merged, the first stage of generalization unifies the *start* and *stop* states. The second stage examines the two states immediately following *start*, and the two immediately preceding *stop*, and tries to merge them. The generalization hierarchy is used at this point. For example, the *move* 6@ 90 will be merged with *move* 3@ 90 into *move–until–contact* 6@ 90 since the shorter move ends with *contact*. The result of this stage of generalization is to merge the middle state of the first trace with both the second and penultimate state of the second trace, forming a single state out of three. This is shown in Figure 5(b) as the state *at(?, ?); move-until-contact-toward(0, 0)*, in other words, from anywhere, move toward the goal until contact is made. As can be seen, this is the point at which the loop in the procedure appears. When the third trace is considered, state merging is prevented by different parameters appearing in the *move* actions. However, there are strong structural reasons for attempting to merge corresponding states, and so NODDY embarks upon the third stage of generalization, namely functional induction. It rationalizes the difference between the *move* parameters because they are functions of the contact angle already observed. This produces the final procedure.

There are two important differences between NODDY and the other two concept-learning systems. Firstly, it is capable of using pre-specified domain knowledge, in the form of the generalization hierarchy for actions and prior information about operators needed for functional induction. This combination of knowledge-based and enumerative generalization is likely to prove a potent one in the future. Secondly, it introduces the idea of

† In fact, the procedure does not require both predecessor and successor states to be identical, but is applied to parallel chains of states which begin and end in the same state and, if successful, merges corresponding members of the chains. Consequently the merging is only done if both predecessor and successor states end up being identical.

grading the search space by looking harder for generalizations in situations in which some justification already exists. The first stage only merges identical states. The second uses these as anchor points to try to unify others using more powerful (and expensive) techniques. The third uses a more stringent criterion of anchoring and embarks on a very powerful (and incredibly expensive) search for functions which account for differences in parameters.

Conclusions

Concept learning is billed as a way of acquiring knowledge from a domain expert who has no knowledge engineering skills. This paper fuses the theory and practice of concept learning, both in general and through detailed consideration of three representative systems. Figure 6 gives plain answers to the questions raised at the beginning for these systems; however, by now it will be clear that most involve issues of such complexity that simple yes/no judgements are quite inadequate. The answers should be viewed as a starting point for further discussion, not a conclusion.

Established theoretical results for identifying languages by enumeration, summarized in Figure 2, delimit the possible from the impossible in concept learning. They show that, in general, there is no alternative to searching a space of candidate descriptions. When presented in an easily identifiable sequence, positive examples alone enable recursively enumerable description classes to be learned. In general, however, positive examples are suitable only for learning finite languages, while the inclusion of negative ones enables primitive recursive descriptions to be learned. Additional conditions which are satisfied by the teacher may speed the search for a suitable generalization. Concepts represented in expert systems must be humanly readable, so that the user can both obtain explanations about the system's reasoning, and tune its knowledge directly. Practical systems must build on concepts already learned.

The version space method explicitly stores the finite upper and lower lattice edges of the plausible generalization set, attempting to merge the two as examples are presented. MARVIN and NODDY search a lattice infinitely wide and high, but pay the penalty of needing more constraints to guide them. MARVIN relies on the teacher first presenting rudimentary concepts which can be used later to help represent more complex ones. This allows it to bias search in favour of representations built upon existing concepts. To guide its search, NODDY has both domain knowledge and knowledge of possible component functional relationships. It generalizes conservatively, in three stages, with the power of generalization and amount of justification required increasing in concert through the stages.

Shortcomings in current concept learning systems include a lack of techniques for generating "typical" examples, and the inability to structure concepts automatically for efficient later use. A more major deficiency is their complete inadequacy to handle the imperfect nature of real applications. There is no robustness against misconception, error and noise in examples. Moreover, teaching shares too many of the drawbacks of programming. Users will want to think only about their problem domain, and not dwell upon the machine's limitations. Finally, techniques for enumeration and knowledge-based search must be combined. The practical shortcomings of representative existing systems are apparent from Figure 6. Although holding great promise for the future, concept learning is not yet a mature technology suitable for practicing knowledge engineers.

References

Andreae, P.M. (1984a) "Constraint limited generalization: acquiring procedures from examples" *Proc American Association on Artificial Intelligence,* Austin, TX, August.

Andreae, P.M. (1984b) "Justified generalization: acquiring procedures from examples" PhD Thesis, Department of Electrical Engineering and Computer Science, MIT.

Angluin, D. and Smith, C.H. (1983) "Inductive inference: theory and methods" *Computing Surveys, 15* (3) 237-269, September.

Davis, R. and Lenat, D.B. (1982) *Knowledge-based systems in artificial intelligence.* McGraw Hill, New York, NY.

Gill, A. (1961) "State-identification experiments in finite automata" *Information and Control, 4,* 132-154.

Ginsburg, S. (1958) "On the length of the smallest uniform experiment which distinguishes the terminal states of a machine" *J Computing Machinery, 5,* 266-280.

Gold, E.M. (1967) "Language identification in the limit" *Information and Control, 10,* 447-474.

Kolokouris, A.T. (1986) "Machine learning" *Byte,* 225-231, November.

Lenat, D.B. (1978) "The ubiquity of discovery" *Artificial Intelligence, 9,* 257-285.

Lenat, D.B., Sutherland, W.R., and Gibbons, J. (1982) "Heuristic search for new microcircuit structures: an application of artificial intelligence" *AI Magazine,* 17-33, Summer.

Lenat, D.B. (1983) "EURISKO: a program that learns new heuristics and domain concepts" *Artificial Intelligence, 21,* 61-98.

Lenat, D.B. and Brown, J.S. (1984) "Why AM and EURISKO appear to work" *Artificial Intelligence, 23,* 269-294.

Mitchell, T.M. (1982) "Generalization as search" *Artificial Intelligence, 18,* 203-226.

Moore, E.F. (1956) "Gedanken experiments on sequential machines" in *Automata studies,* edited by C.E.Shannon and J.McCarthy, pp 129-153. Princeton University Press, Princeton, NJ.

Quinlan, J.R. (1986) "Simplifying decision trees" *Proc Workshop on Knowledge acquisition for knowledge-based systems,* Banff, Canada, November.

Sammut, C. and Banerji, R. (1983) "Hierarchical memories: an aid to concept learning" *Proc International Machine Learning Workshop,* 74-80, Allerton House, Monticello, IL, June 22-24.

Sammut, C. and Banerji, R. (1986) "Learning concepts by asking questions" in *Machine learning Volume 2,* edited by R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, pp 167-191. Morgan Kaufmann Inc, Los Altos, CA.

Shapiro, E.Y. (1983) *Algorithmic program debugging.* MIT Press, Cambridge, MA.

Solomonoff, R.J. (1964) "A formal theory of inductive inference Parts I and II" *Information and Control, 7,* 1-22 and 224-254.

Thompson, K. (1984) "Reflections on trusting trust" *Communications of the Association for Computing Machinery, 27* (8) 761-763, August.

Van Lehn, K. (1983) "Felicity conditions for human skill acquisition: validating an AI-based theory" Research Report CIS-21, Xerox PARC, Palo Alto, CA, November.

Winston, P.H. (1975) "Learning structural descriptions from examples" in *The psychology of computer vision,* edited by P.H.Winston. McGraw Hill, New York, NY.

Witten, I.H. and Cleary, J.G. (1987) "Inductive modeling for data compression" *Fourth International Symposium on Modelling and Simulation Methodology,* January 21-23.

Woods, W.A. (1970) "Transition network grammars for natural language analysis" *Communications of the Association for Computing Machinery, 13* (10) 591-606, October.

Glossary

Concept, description, model. Class of objects characterized intensively by a structural description.

Generalization, identification, induction, inductive inference, learning. The process of acquiring an intensive description (see *concept*) from an extensive one (see *oracle*).

Enumeration. Counting. A set is enumerable if it can be put into one-to-one correspondence with the integers.

Felicity condition. Constraints imposed on or satisfied by a teacher that make learning better than from random examples.

Finite identification. The learner stops the presentation of examples when it thinks it has received enough and states the identity of the concept.

Frustration sequence. A sequence of examples chosen especially to subvert a particular learner, possibly with knowledge of how it operates.

Identification by enumeration. The learner selects a description and pronounces it to be the desired generalization as long as it is consistent with all examples seen so far, moving on to the next description if not.

Identification in the limit. The learner continually proposes hypotheses stating the identity of the concept and a point is reached after which all guesses are the same.

Informant presentation. The learner can choose examples and have them classified as positive or negative.

Language. A set of sentences. Identifying a language from a given class by examining sample sentences is the same thing as identifying a description from a given class by examining samples of what is being described.

Mixed presentation. A sequence of positive and negative examples which eventually cover all members of some universe.

Oracle. Source of examples which characterize a concept extensively, whether a positive, mixed, informant, or teacher presentation.

Positive presentation. A sequence of positive examples which eventually include all instances of the concept.

Primitive recursive. A more restricted class than recursively enumerable, but one which nevertheless embraces all context-sensitive (and hence all context-free) languages.

Recursively enumerable. A set is recursively enumerable if its members can be enumerated (and the enumeration function is computable).

Rote learning. Learning by simply storing all examples encountered.

Teacher presentation. A sequence of examples chosen by a teacher who is assumed to satisfy some felicity conditions.

List of figures

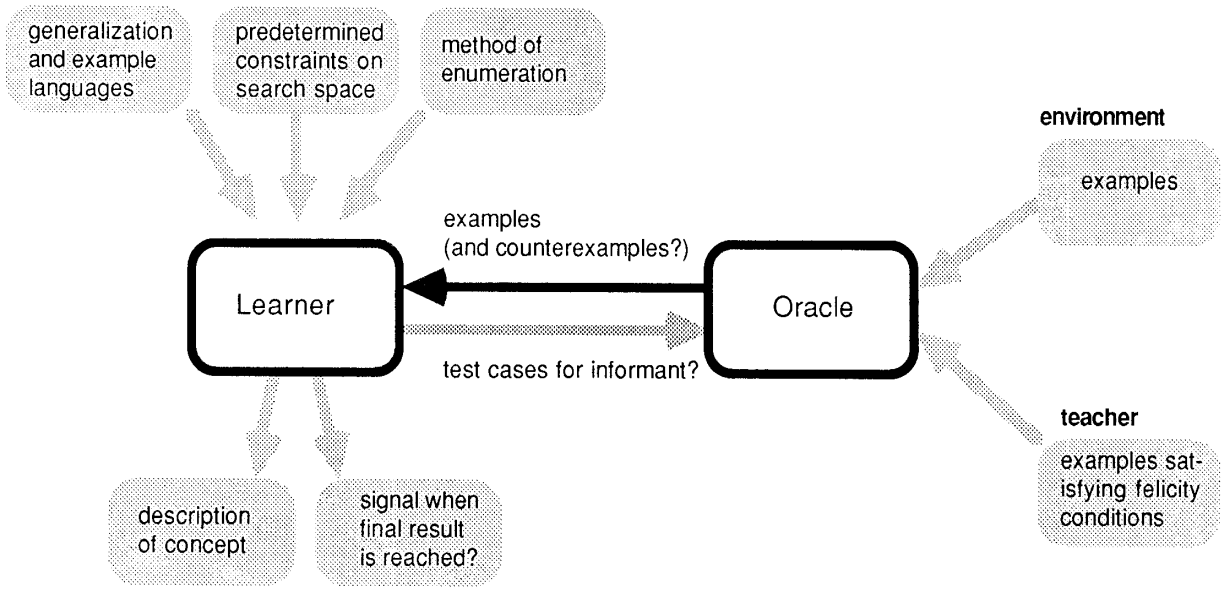

Figure 1 The concept learning situation

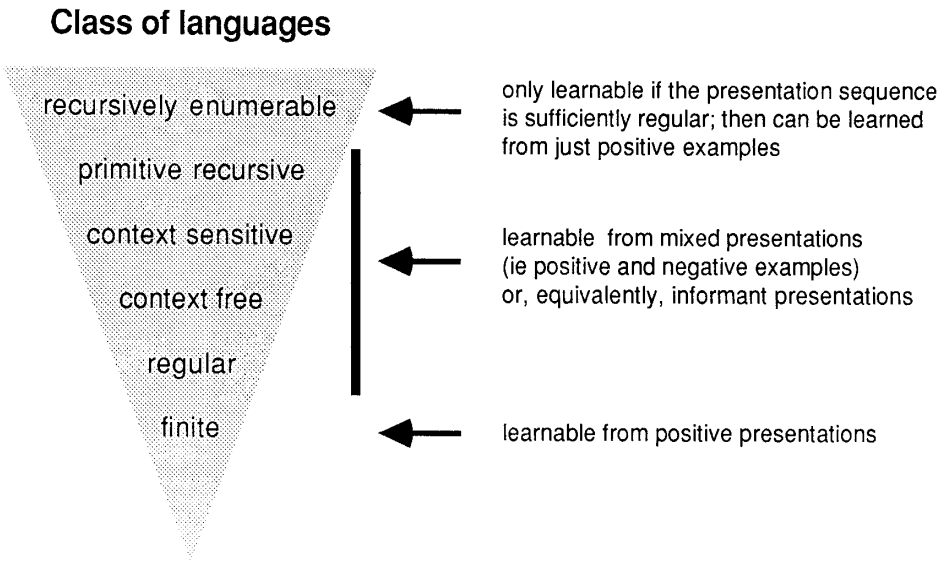

Figure 2 Theoretical results on identification in the limit

(a)

Description	Language
L_∞	{1, 2, 3, . . . }
• • •	• • •
L_{101}	{1, 2, 3, . . . 101}
L_{100}	{1, 2, 3, . . . 100}
L_{99}	{1, 2, 3, . . . 99}
• • •	• • •
L_3	{1, 2, 3}
L_2	{1, 2}
L_1	{1}

(b)

target language	positive presentation	mixed presentation
L_{100}	53,25,2,97,46, . . . (all ≤ 100)	¬2576, ¬198, 53, ¬1007, 25, . . .
L_∞	2576,198, 53,1007, 25, . . . (any numbers)	2576,198, 53,1007, 25, . . .

(c)

enumeration sequence	target L_{100} positive	target L_{100} mixed	target L_∞ positive	target L_∞ mixed
$L_1 L_2 \cdots L_\infty$	ends up with L_{100} as generalization L_{101} L_{102} . . . are never even considered		never reaches a limit but continues to approach it for ever	
$L_\infty L_1 L_2 \cdots$	ends up — incorrectly — with L_∞ as generalization	finds correct generalization, L_{100}	ends up with L_∞ as generalization	

Figure 3 (a) A problem which is identifiable in the limit but only from a mixed presentation

(b) Example presentation sequences

(c) The effect of different enumeration sequences

```
What is the name of the concept? letter
Show me an example of 'letter': a
X is a 'letter' if:
     X is a
Show me an example of 'letter': b
X is a 'letter' if:
     X is b
Show me an example of 'letter': no
X is a 'letter' if:
     X is a
X is a 'letter' if:
     X is b

What is the name of the concept? letter-list
Show me an example of 'letter-list': []
X is a 'letter-list' if:
     X is []
Show me an example of 'letter-list': [a]
X is a 'letter-list' if:
     the head of X is a
     the tail of X is []
The head of X is an instance of 'letter'
The description can be generalized to:
X is a 'letter-list' if:
     the head of X is an instance of 'letter'
     the tail of X is []

Is '[b]' an instance of 'letter-list'? yes
The tail of X is an instance of 'letter-list'
The description can be generalized to:
X is a 'letter-list' if:
     the head of X is an instance of 'letter'
     the tail of X is an instance of 'letter-list'

Is '[a,a]' an instance of 'letter-list'? yes
Show me an example of 'letter-list': no
X is a 'letter-list' if:
     X is []
X is a 'letter-list' if:
     the head of X is an instance of 'letter'
     the tail of X is an instance of 'letter-list'

What is the name of the concept? concatenate
Show me an example of 'concatenate': ([],[a],[a])
X1,X2,X3 is a 'concatenate' if:
     X1 is []
     the head of X2 is a
     the tail of X2 is []
     the head of X3 is a
     the tail of X3 is []
X1 is an instance of 'letter-list'
The description can be generalized to:
X1,X2,X3 is a 'concatenate' if:
     X1 is an instance of 'letter-list'
     the head of X2 is a
     the tail of X2 is []
     the head of X3 is a
     the tail of X3 is []

Is '[a],[a],[a]' an instance of 'concatenate'? no
The tail of X2 is the same as X1
The description can be restricted to:
X1,X2,X3 is a 'concatenate' if:
     X1 is an instance of 'letter-list'
     the head of X2 is a
     the tail of X2 is the same as X1
     the head of X3 is a
     the tail of X3 is []

Is '[a],[a,a],[a]' an instance of 'concatenate'? no
The tail of X3 is the same as X1
The description can be restricted to:
X1,X2,X3 is a 'concatenate' if:
     X1 is an instance of 'letter-list'
     the head of X2 is a
     the tail of X2 is []
     the head of X3 is a
     the tail of X3 is the same as X1

Is '[a],[a],[a,a]' an instance of 'concatenate'? no
The head of X2 is an instance of 'letter'
The description can be generalized to:
X1,X2,X3 is a 'concatenate' if:
     X1 is []
     the head of X2 is an instance of 'letter'
     the tail of X2 is []
     the head of X3 is a
     the tail of X3 is []

Is '[],[b],[a]' an instance of 'concatenate'? no
The head of X3 is the same as the head of X2
The description can be restricted to:
X1,X2,X3 is a 'concatenate' if:
     X1 is []
     the head of X2 is an instance of 'letter'
     the tail of X2 is []
     the head of X3 is the same as the head of X2
     the tail of X3 is []

Is '[],[b],[b]' an instance of 'concatenate'? yes
The tail of X2 is an instance of 'letter-list'
The description can be generalized to:
X1,X2,X3 is a 'concatenate' if:
     X1 is []
     the head of X2 is an instance of 'letter'
     the tail of X2 is an instance of 'letter-list'
     the head of X3 is the same as the head of X2
     the tail of X3 is []

Is '[],[a,a],[a]' an instance of 'concatenate'? no
The tail of X3 is the same as the tail of X2
The description can be restricted to:
X1,X2,X3 is a 'concatenate' if:
     X1 is []
     the head of X2 is an instance of 'letter'
     the tail of X2 is an instance of 'letter-list'
     the head of X3 is the same as the head of X2
     the tail of X3 is the same as the tail of X2

Is '[],[a,a],[a,a]' an instance of 'concatenate'? yes
Show me an example of 'concatenate': ([a],[b],[a,b])
X1,X2,X3 is a 'concatenate' if:
     the head of X1 is a
     the tail of X1 is []
     the head of X2 is b
     the tail of X2 is []
     the head of X3 is a
     the tail of X3 is [b]
The head of X1 is an instance of 'letter'
The description can be generalized to:
X1,X2,X3 is a 'concatenate' if:
     the head of X1 is an instance of 'letter'
     the tail of X1 is []
     the head of X2 is b
     the tail of X2 is []
     the head of X3 is a
     the tail of X3 is [b]
```

Figure 4 Transcript of a session with MARVIN

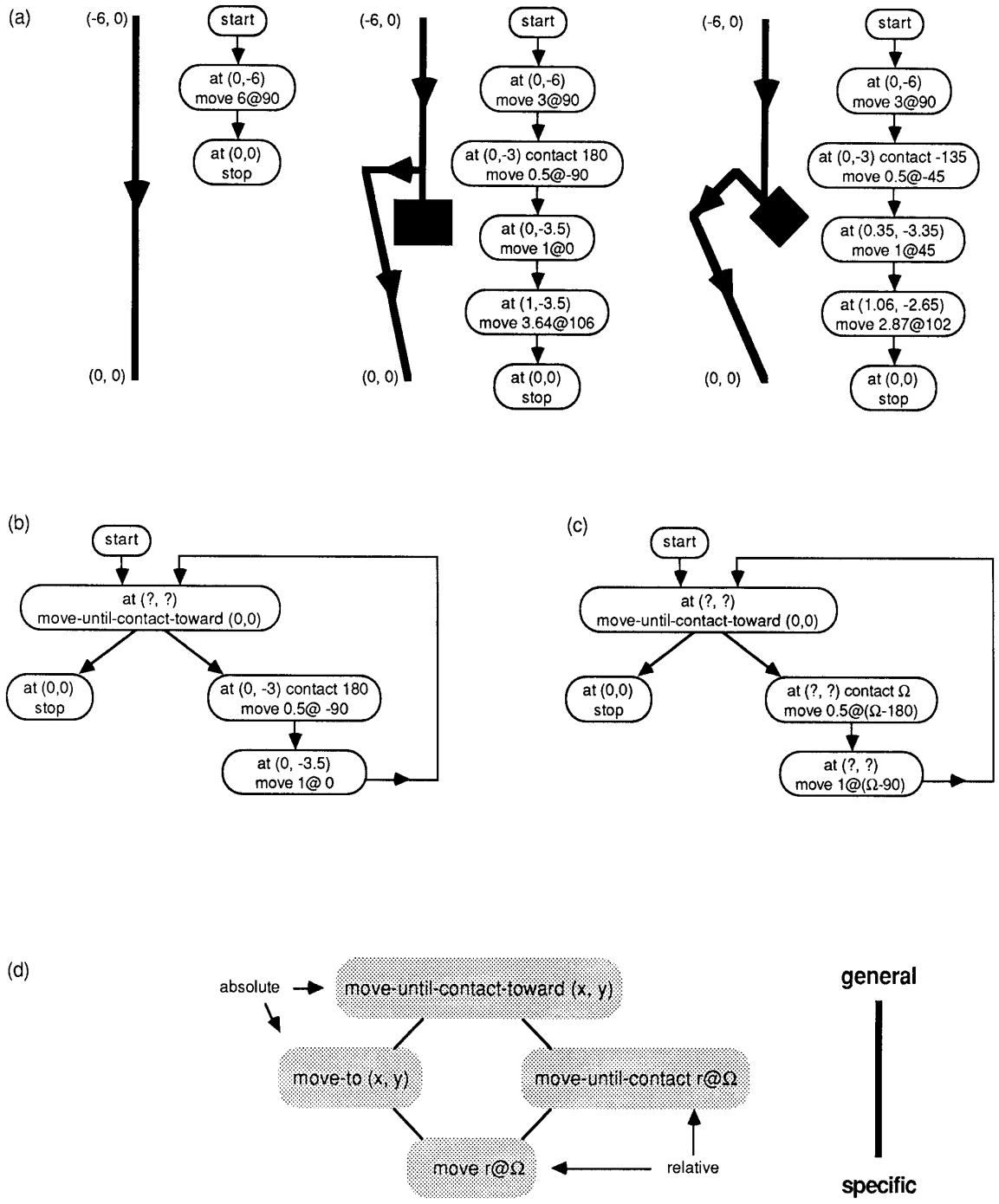

Figure 5 NODDY

(a) Example robot traces

(b) Partially complete procedure

(c) Final, desired, procedure

(d) Generalization hierarchy

	Version space	MARVIN	NODDY
Always produces correct concept	yes	no	yes
Takes a long time: execution time	no	yes	yes
number of examples	no	yes	no
Classifies old examples correctly	yes	yes	yes
Classifies novel examples	yes	yes	yes
Recovers from misconception	no	no	no
Handles erroneous examples	no	no	no
Can employ negative examples	yes	no	no
Generates test cases itself	no	yes	no
Explores on its own	no	no	no
Sensitive to the order of examples	no	yes	yes
Needs a teacher	no	yes	yes
Knows when it has finished	yes	yes	no
Finite or infinite set of concepts	usually finite	infinite	infinite
Uses concepts as components of others	no	yes	no
Generates concise representation	yes	yes	yes

Figure 6 Plain answers to plain questions