The University of Alberta

GUIDING CONSTRUCTIVE INDUCTION FOR INCREMENTAL CONCEPT LEARNING FROM EXAMPLES

by

Larry Masato Watanabe

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1987

ISBN 0-315-37806-9

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Larry Masato Watanabe

TITLE OF THESIS: Guiding Constructive Induction for Incremental Concept Learning from Examples

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1987

Larry Masato Watanabe

13 Lismer Crescent

Kanata, Ontario

K2K 1A3

Date: March 28, 1987

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Guiding Constructive Induction for Incremental Learning from Examples** submitted by **Larry Masato Watanabe** in partial fulfillment of the requirements for the degree of **Master of Science**.

[illegible] ....

Supervisor

[illegible] ....

[illegible] ....

[illegible] ....

Date April 14/87

ABSTRACT

LAIR is a system that incrementally learns conjunctive concept descriptions from positive and negative examples. These concept descriptions are used to create and extend a domain theory that is applied, by means of constructive induction, to future learning tasks. Important issues for constructive induction are *when* to do it and *how* to control it. LAIR demonstrates how constructive induction can be controlled by (1) reducing it to simpler operations, (2) constraining the simpler operations to preserve relative correctness, (3) doing deductive inference on an as-needed basis to meet specific information requirements of learning sub-tasks, and (4) constraining the search space by subtask-dependent constraints. In addition, LAIR's rules are relatively complete, in the sense that a correct, usable concept description can be derived, if one exists, without backtracking or maintaining multiple concept descriptions. LAIR also shows how similarity-based learning techniques can be used to acquire and extend domain theories for explanation-based learning systems, and how goal concepts for these systems can be inferred by example rather than explicitly given to the system.

Acknowledgements

I wish to express my gratitude and appreciation to my supervisor, Dr. Renée Elio, for her ideas, suggestions, and guidance. Dr. Elio gave generously of her time and effort at crucial points during this thesis, and created a good research environment, the Lab for Artificial Intelligence Research, in which to implement LAIR. Dr. Elio organized the Analogies Study Group and the Machine Learning Reading Group, in which I found the topic for my thesis.

I would like to thank the members of my examing committee, Dr. Lenhart K. Schubert, Dr. Francis J. Pelletier, and Dr. Emil Gircyzc, for their worthy criticism and suggestions. Special thanks go to Dr. Schubert and Dr. Pelletier for their comments on the logical formalism used in LAIR. Dr. Schubert was also responsible for much of my training in artificial intelligence, and has helped and supported me in numerous ways throughout my Master's program.

Thanks go to my fellow students and for their help, advice and encouragement, and suggestions; in particular, Ajit Singh, Fahiem Bacchus, Brian Wilkerson, and C. S. Law. Thanks also go to the members of the Machine Learning Research Group; Michael Gruninger, Sean Cormack, Peternala Scharf, Bonita Wong, Brant Coghlan, Peter Leung, and John Nesbitt. Michael Gruninger helped with the implementation of LAIR, and gave much useful advice on the InterLisp-D programming environment.

I thank Dr. Wlodek Dobosowiecz for his advice and support, and for providing a challenging opponent for games of GO. Dr. Stan Cabay, my academic advisor, guided me through the pitfalls of first year coursework and the adjustment to graduate school. The staff of the Department of Computing Science helped in many ways, in particular Lynn Gaetz, Debbie Pallant, Amanda Collins, and Paul Godreau.

Most of all, I would like to thank my wife, Kyoko Iwasaki, for her love, support, encouragement, and understanding.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1. Overview

Learning is a feature of human intelligence that is fundamental to intelligent behaviour. The ability of humans to learn allows discovery of new concepts, improvement in performance, and the transfer of ideas and expertise from one human to another. A common definition of learning is the following: learning is any process by which a person or system improves his or her performance on a task or similar tasks. This definition is general enough to cover the many types of learning that people seem to do: learning by being told, learning by doing, learning from examples, learning by analogy, and so on. For researchers interested in understanding human learning or researchers interested in providing machines with learning capabilities, the issue is the same: what are the mechanisms and processes that support what we would characterize as "learning" under our definition?

This work concerns learning from examples. Learning from examples has had a long history of research in both cognitive psychology and artificial intelligence. Learning from examples is a special case of inductive learning where the learner must induce a general concept description from a set of examples. Typically, learning mechanisms examine only those features that are explicitly represented in the example set. This research concerns the role of prior knowledge and deduction in going beyond the very specific concept information that a series of examples provides for the learner. In particular, I present a framework for viewing the inductive process of learning from examples as combination of deductive inference and constrained, learning-specific inductive inference. Deduction becomes important when the positive examples have no apparent commonalities, or the negative examples have no apparent dissimilarities to the positive examples. In these cases, the learner must consider what is known about the features that comprise the examples and try to deduce some commonalities or dissimilarities. I have designed and implemented a system called LAIR (Learning And Inductive Reasoning) that illustrates this framework and how the deductive and learning-specific inductive processes can be guided by the constraints inherent in the learning task.

The remainder of this introduction is organized as follows. First, a brief perspective on machine learning is presented. Next, I consider learning by example in more detail, first reviewing the major research efforts in this area, and then focusing on recent concern with the role of knowledge.

1.2. Historical Perspective on Machine Learning

1.2.1. Parameter Adjustment

One of the earliest methods of learning studied in artificial intelligence is *learning by parameter adjustmen* Two classes of systems used learning by parameter adjustment: neural modelling systems and decision-theoretic systems. The neural modelling systems, also called self-organizing systems, modified themselves to adapt to their environments (Yovits, Jacobi, & Goldstein, 1962; Barr & Feigenbaum, 1982). The best-known neural modelling system, perceptrons (Rosenblatt, 1957), met with some limited success. However, Minsky and Papert's (1969) analysis of the theoretical limitations on the learning capabilities of perceptrons lead to a decline of interest in this approach among artificial intelligence researchers. Work has continued along these lines in the areas of linear systems theory, pattern recognition, and control theory. Recently, connectionist researchers have redefined and extended some of the neural modelling approaches.

The decision-theoretic approach was closely related to and sometimes overlapped the neural modelling approach. In the decision-theoretic approach, a system acquired an arithmetic discriminant function (e.g., a polynomial) from a set of training examples presented to the system by a user or from a *critic* (a system module that extracted positive and negative examples from traces of the system's performance on a task). Along with theoretical work (Nilsson, 1965), applied systems were developed in areas such as game-playing (Samuels, 1958) and pattern recognition (Uhr & Vossler, 1963).

Samuel's checkers player was one system that remains noteworthy for its early success as a learning system, so I will discuss his system in further detail. Samuel's checkers player incorporated several measurements of features of the domain into a single summary statistic, using a static evaluation function

$$g(f_1, \dots f_n) = a_1 f_1 + \dots + a_n f_n$$

where f_i was the value of the ith feature, and a_i was the weight attached to that feature. This function was used in an alpha-beta search procedure to estimate the value of a position.

Samuel's program adjusted the coefficients to make the value of the static evaluation function g more closely approximate the values backed-up by the alpha-beta procedure, on the assumption that backed-up values were more accurate. This can be viewed as one of the earliest examples of the use of a critic to automatically extract positive examples from a performance trace: positive examples corresponded to backed-up values. Samuel also used principles of self-organizing systems to determine the value of the static evaluation function by playing two programs against one another, one version using the old static evaluation function, and the other using the modified evaluation function. If the new version won, the old static evaluation function was replaced by the new one. In addition, Samuel used perturbation methods to get off local maxima by dropping or replacing features.

Despite the limitations of parameter-adjustment approaches, they illustrate certain themes like "strength of association" and "rewarding useful features" that have pervaded theories of human cognition. These ideas are still prominent in many current models of human and machine learning, including connectionist approaches (Feldman, 1984; Hinton, Sejnowski, & Ackley, 1984). In most of these models, feature patterns accumulate strength on one or both of these measures: how frequently they occur in association with the concept, and how useful they have been in identifying the concept (Hinton &

Anderson, 1983; Anderson, 1983; Langley, 1986). The strength of stored patterns then serves as a characteristic that can be considered when there are several competing patterns that the system can use to classify an input pattern.[1] However, strength cannot represent why a feature pattern was associated with the concept, or why a feature pattern was useful in identifying the concept. Therefore, recent systems usually augment strength with additional knowledge. For example, Langley, Klahr and others recognize the importance of a *history trace* that records past decisions and information about the contexts in which those decisions were made.

The problems with strength as knowledge about past performance occur in general parameter adjustment systems. Using parameters, it is difficult to express relationships between objects, properties of objects, and contingent relationships or properties, all of which are standard in most knowledge representation schemes. Further, learning by parameter adjustment is essentially a hill-climbing strategy, and like other hill-climbing strategies, may get stuck on local maxima. However, parameters can be combined with more expressive knowledge representation schemes, and parameter adjustment can be combined with more powerful learning techniques. Simple learning by parameter adjustment remains useful as an auxiliary or default technique that can often be used to advantage where more sophisticated techniques may be too difficult to implement.

1.2.2. Concept Learning

In the early sixties, systems that learned symbolic concept descriptions from examples were developed. Hunt (1966) formulated the conjunctive and disjunctive concept learning tasks within a predicate logic framework, and successfully modelled human concept learning in situations where the examples were describable by features, or binary predicates stating the value of some dimension of an object. Later programs used more expressive

[1] Anderson's ACT theory and models Langley has built using PRISM use complex resolution schemes for choosing among competing patterns, in which strength is only one component.

languages, such as Winston's program ARCH (1975) that used a semantic network representation, Hayes-Roth's SPROUTER (Hayes-Roth & McDermott, 1976; Hayes-Roth, 1977; Hayes-Roth & McDermott, 1978) that used a case frame representation, and systems using predicate-logic representations such as Vere's systems (1975, 1977, 1978, 1980), Mitchell's version-space approach (1977) and Michalski's AQVAL (1973, 1978). About this time, systems using knowledge-intensive approaches began to appear.

1.2.3. Knowledge-Intensive Learning

Concurrent with the development of concept learning systems that manipulated symbolic representations of concepts and examples, researchers in knowledge-based systems (more commonly known as "expert" systems) began to explore the use of automated methods of knowledge acquisition to create the required knowledge bases. These systems were characterized by domain-specific, knowledge-intensive learning techniques, such as those used in the META-DENDRAL program (Buchanan, 1978). META-DENDRAL encoded large amounts of knowledge about chemical structures to learn rules for explaining mass spectrometry data. These rules were used in the DENDRAL program, one of the early successes in knowledge-based systems. A difficulty with these types of learning systems is that little effort was expended on explicating the theory behind the learning procedures, so that few of the techniques developed were transferable to other domains (Michalski et al., 1986).

Although the task-specific systems made little direct contribution to learning theory, the success of the knowledge-intensive approach led to awareness that "in order to acquire new knowledge, it is necessary to already have a great deal of knowledge" (Carbonell et al., 1983). In recent years, general purpose, theoretically justified knowledge-intensive learning systems were developed (Michalski, 1983), along with systems based on new learning paradigms such as operationalization (Mostow, 1983), learning by discovery (Lenat, 1977; 1983a; 1983b), conceptual clustering (Michalski & Stepp, 1983; Stepp &

Michalski, 1986), and learning by analogy (Carbonell, 1983, 1986). Systems were also characterized by having greater control over the learning process: newer systems used techniques such as asking questions of the user (Sammut & Banerji, 1986), and generating new learning tasks (Lenat, 1977; 1983a; 1983b).

The use of knowledge in learning from examples has emerged as a major research issue. The next section discusses some important distinctions in the learning from examples paradigm and the need to include mechanisms for using prior knowledge to achieve more flexible and robust learning.

1.3. Learning from Examples

The ability of people to learn complex concepts from experience with examples is ubiquitous. Suppose we wanted to teach a person the concept "chair" by example. We could provide a positive example like "has four legs, has a seat, has two arms, found in living rooms." Another example might be "has four legs, has a seat, has no arms, found in office." From this, the person might infer that a "chair" is something that has four legs and a seat; where it is found or the number of arms does not matter. In other words, some features occur in both examples; the ones that vary might not be critical to the description. Conversely, we could provide the learner with a negative example like "has four legs, found in living rooms, barks loudly." From this, the learner might infer that a "chair" is something that has four legs, has a seat, and does *not* bark loudly. Features that occur in negative examples are plausible negative features of the description.

The above example included a number of assumptions about the learner and the learning task: it was assumed that both positive and negative examples were presented, and that a concept description can include negative features. Further, examples and concept descriptions were assumed to be representable as sets of features. These are only a few of many assumptions that might be made in a learning task.

The next sections will discuss a number of important distinctions between systems that learn by example, many of which are presented in (Carbonell, Michalski & Mitchell, 1983; Dietterich & Michalski, 1983; Michalski, 1983; Michalski, 1986):

1. What kinds of examples are recognized by the system?
2. What is the source of the examples?
3. What is the representation language?
4. What kinds of knowledge are represented by the system?
5. What is the desired concept description?
6. Is learning incremental?
7. When can new knowledge be inferred?
8. What are the mechanisms for creating and modifying concept descriptions?
9. How do the examples delimit the concept description space?

These distinctions are important because they determine the kinds of learning tasks for which a learning system is useful, and the power and flexibility of the system.

1.3.1. Kinds of Examples

Researchers have distinguished many kinds of examples: positive examples, negative examples, and prototypical examples, to name a few. Learning systems typically recognize only positive and negative examples. Most learning systems can be classified into two categories: those that recognize only positive examples, and those that recognize both positive and negative examples. Systems that use only positive examples require constraints on learning to avoid over-generalization, because the system will not be able to determine from only positive examples whether its concept description is too general. For example, Berwick (1983) identifies the "Subset Principle," a strategy of "timid acquisition," as a useful constraint that avoids over-generalization:

> If possible guesses can be arranged in a subset relationship, then the learner should make the smallest possible guess about what it should learn consistent with the evidence it has seen so far.

A guess G is a subset of a guess H if everything it describes is also described by H. Many other constraints on learning from positive-only examples have been identified, e.g., by Angluin (1978) in learning recursive languages.

Systems that recognize both positive and negative examples can use the information in the negative examples to avoid over-generalization. Typically, the negative examples are used to constrain the generalization process (Michalski, 1983) or to force specialization of the concept description (Winston, 1975; Mitchell, 1977; Langley, 1986). Since the methods such as the "Subset Principle" can also be used in systems that recognize both positive and negative examples, most recent systems recognize both positive and negative examples.

1.3.2. Source of Examples

Two important questions about the source of examples are "What is the source of examples?" and "Is the source helpful?" A source can be external, such as a teacher or the environment, or internal, i.e. the learner itself. An teacher presents positive examples (and possibly negative examples) of the concept to the learner. A helpful teacher tries to infer the state of knowledge of the system in order to provide examples that help the system induce the correct concept description as quickly as possible. A non-helpful teacher generates essentially random examples.

The second source of examples is the environment. Typically, the system observes some data gathered by monitoring equipment or observes experimental results. Note that we can consider the "monitoring equipment" or "experimental module" to be a teacher that supplies examples to the rest of the system, and that this teacher is non-helpful because it lacks control over the environment and knowledge about the target concept.

A third source of examples is the learner. Typically, the learner creates examples based on its current knowledge state, presents it to a teacher, and asks the teacher whether the example is an instance or a non-instance of the concept. This is advantageous because the learner knows its own knowledge state, whereas a teacher can only make plausible conjectures about the knowledge state of the learner. The disadvantages are that the learner has less knowledge of the concept than the teacher, so the sequence of examples presented may not be the best one for learning the concept.

1.3.3. Representation Language

The representation language determines to a large extent the knowledge that can be represented by the system. Typical representation languages are the predicate calculus, production rules, semantic networks, hierarchical descriptions, frames, feature-values, and scripts. Most of these languages are subsumed in expressive power by the first-order predicate calculus (Schubert, 1976) but are advantageous because knowledge may be easier to organize, represent, or use. For example, most feature-value representations can be viewed as a subset of first-order predicate calculus consisting of binary predicates (features), constants (values), and "$\wedge$" (conjunction), but can be represented easily as vectors or tables. Feature-value representations cannot easily represent structural relationships between two objects such as "a door-knob is part of a door," which can be easily represented in semantic networks (e.g. by using NODE-1 to denote "door-knob," NODE-2 to denote "door," and LINK-1 to denote "part-of"), or in first-order predicate calculus (e.g., "Part-of(door-knob, door)"). Therefore, I will consider only extended feature-value representations that are isomorphic to the predicate calculus, referring to a predicate as a feature, predicate arguments as dimensions, and constants as values.

Regardless of the representation formalisms used, a system cannot learn anything it cannot, in principle, describe. Thus, the representation language circumscribes the kinds of concepts the system can learn.

1.3.4 Kinds of Knowledge

Concept learning requires many kinds of knowledge: knowledge about the world, knowledge about examples, and knowledge about concept descriptions. There is an important distinction between a representation language and knowledge: a representation language must have an *interpretation* in order to represent knowledge. Learning by example is one way in which the knowledge of the system can be extended: a teacher can establish a correspondence between objects and relations known to the system, in the form of examples, and expressions in the language, in the form of concept names. Some researchers consider only those expressions that have an interpretation to be "in the representation language" and refer to the above process as "introducing new terms" (Barr & Feigenbaum, 1982) or "extending the representation language" (Mitchell et al., 1983).

Knowledge about examples comprises the example description and the role of examples in learning. A system uses example descriptions and world knowledge, typically relationships between example features and other features, to infer further features of an example. A system uses knowledge about the role of examples in learning, and knowledge about concept descriptions, to decide what is relevant and how concept descriptions should be changed in response to a positive or negative example.

1.3.5 Desired Concept Descriptions

A concept can be described in many ways. Learning systems usually incorporate some means of distinguishing between desirable concept descriptions and undesirable ones. The most common criteria for a desirable concept description is that it be correct. However, correctness may be relative to the goals of the learning task. Michalski has identified several types of learning tasks associated with various types of concept descriptions, and their corresponding correctness criteria.

The first type of concept description is a *characteristic description*. A characteristic description is constrained to describe all the presented positive instances of the concept, and none of the presented negative instances of the concept. Thus, if D is a characteristic description of y, then "x is an instance of concept y" is logically equivalent to "x is described by description D."

The second type of concept description is a *discriminant description*. A discriminant description is constrained to describe all the instances of the concept, and no instance that is an instance of some fixed set of other classes of objects. This is a weaker form of description than a characteristic description, because a discriminant description may describe a non-instance of the concept that is not an instance of any of the other classes of objects, whereas a characteristic description may not.

The third type of description is a *taxonomic description*. A taxonomic description is a description of a class of objects that subdivides the class into subclasses. Thus, it is possible to infer from the way an instance satisfies a taxonomic description additional information: to what subclass does it belong.

In addition to the above constraints on a concept description, there may exist syntactic constraints on a concept description. For example, if the representation language is first-order predicate calculus, concept descriptions may be constrained to be *conjunctive* rather than *disjunctive*. A conjunctive description uses only the connectives "$\neg$" (for atomic negation) or "$\wedge$" (conjunction). A disjunctive description may also use the connective "$\vee$" (disjunction). Thus, disjunctive descriptions can describe concepts that conjunctive descriptions cannot, but are generally more difficult to learn.

Criteria for desirability may also involve considerations of how the concept description is to be used. Some kinds of concept descriptions may be more efficient to use in a given system than others (Mitchell, 1986). Concept descriptions that are to be used by humans must be comprehensible.

1.3.6. Incremental vs. Non-Incremental Learning

In incremental learning, the learner creates or modifies concept descriptions as each example is encountered. This contrasts with non-incremental learning, in which the learner creates or modifies concept descriptions after all the examples are presented. Incremental learning is important for several reasons. First, non-incremental systems have trouble coping with large sets of examples because of the need to store all the information about each example. Second, it is not clear that any learner (human or machine) will be guaranteed to have all examples at its disposal. An incremental learning system can adjust a current concept to accomodate new examples, whereas a non-incremental learning system might have to relearn the concept with all past and present examples. Third, humans learn concepts incrementally, so computer models of incremental learning may be useful for understanding human learning.

1.3.7. When Can New Knowledge be Inferred?

Learning systems that represent knowledge about the world typically acquire this knowledge before the learning task. I define prior knowledge as knowledge about the world that the learner brings to the learning task. A learning system may also infer new knowledge from prior knowledge and knowledge about examples. The question of when this knowledge is inferred is important. A system that infers new knowledge about an example only immediately after the example is presented must ensure that all the needed knowledge is inferred. If a system is unable to predict accurately what knowledge will be needed, then the system may have to infer a great deal of knowledge that will not be useful. A system that can infer knowledge about an example at any time need only infer enough knowledge to meet the current requirements of the learning task.

1.3.8. Mechanisms for Creating and Modifying Concept Descriptions

The mechanisms that create and modify concept descriptions are critical to learning from examples, because they are essentially the "learning mechanisms." An incremental learning system must know how to create (or induce) a concept description from one example, and how to modify that description as additional examples are presented. A non-incremental learning system must know how to induce a concept description from a set of examples. Both strategies can be modelled as an heuristic search of the space of possible concept descriptions (Mitchell, 1982; Michalski, 1983), where the learning mechanisms are description transformation rules that move between different alternative descirptions of the concept in the search space. There are essentially two classes of description transformation rules: generalization rules that transform a description into a more general description, and specialization rules that transform a description into a more specific description.

1.3.9. How do Examples Delimit the Concept Description?

The way in which examples delimit the concept description is critical to understanding the role that prior knowledge plays in learning. This is an important distinction for understanding the thrust of this research. For example, consider a learning system using a feature-value representation with conjunction, disjunction, complex values, that learns only from positive examples. Many systems presuppose that all the relevant dimensions are already present somewhere in the features that comprise the example. The learner must just identify which ones are the relevant ones. However, one can argue that the hard part of learning is figuring out what is *potentially* relevant. Furthermore, the features in the examples may not really reflect what the concept is. We can consider three different cases:

1. All the relevant features and values are presented. The learner must find the right pattern that corresponds to the desired concept description.
2. All the relevant features are there, but not the relevant values.
3. Neither the relevant features nor values are explicitly in the examples.

I will clarify these differences by presenting three examples corresponding to each of these cases. In the first case, the system is a non-incremental learner, the problem is to find a characteristic description, only positive examples are presented, and all the relevant features and values are presented. For example, the features might be "Size," "Color," "Locomotion," and "Habitat," which can have values "small, medium, or large," "red, pink, or brown," "flies or runs," and "Florida, tree, or Australia," respectively.

Feature	Values
Size	small, medium, large
Color	red, pink, brown
Locomotion	flies, runs
Habitat	Florida, tree, Australia

The system might be provided with the following positive examples of "Bird":

[Size(x) = small] [Color(x) = red] [Locomotion(x) = flies] [Habitat(x) = tree]
$\Rightarrow$ Bird(x)
[Size(x) = large] [Color(x) = pink] [Locomotion(x) = runs] [Habitat(x) = Florida]
$\Rightarrow$ Bird(x)
[Size(x) = medium,] [Color(x) = brown] [Locomotion(x) = flies] Habitat(x) = tree]
$\Rightarrow$ Bird(x)
[Size(x) = large] [Color(x) = brown] [Locomotion(x) = runs] [Habitat(x) = Australia]
$\Rightarrow$ Bird(x)

From these examples, the system might induce the following descriptions of "Bird":

[Size(x) = small] [Color(x) = red] [Locomotion(x) = flies] [Habitat(x) = tree] $\vee$
[Size(x) = large] [Color(x) = pink] [Locomotion(x) = runs] [Habitat(x) = Florida] $\vee$
[Size(x) = medium,] [Color(x) = brown] [Locomotion(x) = flies] Habitat(x) = tree] $\vee$
[Size(x) = large] [Color(x) = brown] [Locomotion(x) = runs] [Habitat(x) = Australia]
$\Rightarrow$ Bird(x)]

[Size(x) = small, medium, large] [Color(x) = red, pink, brown] [Locomotion(x) = flies, runs] [Habitat(x) = tree, Florida, Australia] $\Rightarrow$ Bird(x)

[Size(x) = small, large] [Locomotion(x) = flies] [Habitat(x) = tree)] $\vee$
[Size(x) = large] [Locomotion(x) = runs)] $\Rightarrow$ Bird(x)

These are only a few of many possible descriptions the system could have induced. The first description simply corresponds to the disjunction of each of the positive examples, and therefore describes no non-instance of "Bird." However, it is also the longest description, and cannot describe any instance of "Bird" that has not already been presented. This is the most conservative description of the concept: it will make no incorrect predictions, but cannot help in classifying any new instance. The second description generalizes the value of each feature to the complex value denoting the set of all values of the feature across all the examples. Note that if the values listed in the table are the only values of these features, then anything with features "Size," "Locomotion," "Color," and "Habitat" will be classified as an instance of Bird. This is the least conservative description of the concept: it will classify many new examples as instances of "Bird," but it is likely to make many incorrect predictions. The third description is intermediate between the first two: it can predict many new instances of "Bird," but not so many that most of them are unlikely to be correct.

To derive any of these descriptions, the learning mechanisms detect *syntactic commonalities* across patterns. If the desired concept description is some combination of these particular features and values, then this kind of approach is sufficient. However, the learning task requires more inference on the part of the learner if some aspect of the concept description is not explicitly represented in the example descriptions. Consider a case where all the relevant features are there, but not all the relevant values. A system is attempting to form a concept that will discriminate "Group1" objects from "Group2" objects, and is given these examples:

[Shape(x) = Triangle] $\Rightarrow$ Group1(x)
[Shape(x) = Circle] $\Rightarrow$ Group2(x)
[Shape(x) = Square] $\Rightarrow$ Group1(x)
[Shape(x) = Oval] $\Rightarrow$ Group2(x)
[Shape(x) = Rectangle] $\Rightarrow$ Group1(x)
[Shape(x) = Ellipse] $\Rightarrow$ Group2(x)

In a very simple case like this, a "Group1" concept might amount to

[Shape(x) $\neq$ Circle, Oval, Ellipse] [Shape(x) = Triangle, Square, Rectangle]
$\Rightarrow$ Group1(x)

and a "Group2" concept might amount to

[Shape(x) $\neq$ Triangle, Square, Rectangle] [Shape(x) = Circle, Oval, Ellipse]
$\Rightarrow$ Group2(x)

Notice these concept descriptions would not be able to deal effectively with a question like "Is a trapazoid in Group 1 or Group 2?" If the system knew that rectangles, squares, and triangles are polygons, and that circles, ovals, and ellipses are round,

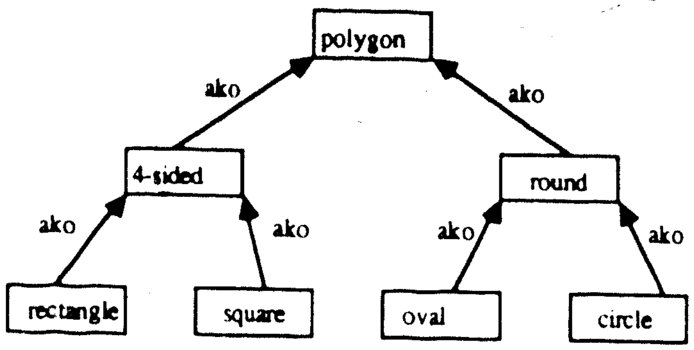

Figure 1.1 Type Hierarchy of Shapes (ako = a kind of)

then [Shape(x) = Polygon] could be inferred to be common to all the Group1 examples, and [Shape(x) = Round] could be inferred to be common to all the Group2 examples. Note that the values "Polygon" and "Round" for the feature "Shape" are not in any of the examples. To introduce a new value into a concept description, the system needs to have prior knowledge and methods of inferring further knowledge from the examples.

We can give the learning system a still more demanding task in which the relevant features (and hence the relevant values) do not appear in any of the examples. This does not necessarily happen out of malicious intent on the part of the teacher. Rather, the teacher (or environment) providing examples may assume that the learner is noticing the "right" aspects of the example. For example, suppose we wanted to teach a robot a description of a chair. We could take the robot to the living room, sit it down in a recliner and tell it "This is a chair." The robot might internalize the following description:

$$\text{Found-in(Place, Obj)} \wedge \text{Living-Room(Place)} \wedge \text{Made-of(Obj, Material)} \\ \wedge \text{Comfortable(Obj)} \wedge \text{Fabric(Material)} \Rightarrow \text{Chair(Obj)}$$

The next time we were at school we could take the robot to the auditorium, point to a chair, and say "That is also a chair." The robot might look at it and internalize the following description:

$$\text{Found-in(Place, Obj)} \wedge \text{Auditorium(Place)} \wedge \text{Made-of(Obj, Material)} \\ \wedge \neg\text{Comfortable(Obj)} \wedge \text{Metal(Material)} \wedge \text{Has(Obj, Legs)} \Rightarrow \text{Chair(Obj)}$$

At this point, we might feel we have provided fine examples of "chair." Unknown to us, and unfortunately for the robot, the features that it has noticed have no apparent commonalities or relation to the concept we are attempting to teach. This becomes clear when we point to a lawn chair, ask if it is a chair, and the robot (observing that it is made of plastic, has legs, is found near the pool, is covered with snow and its comfort is unknown) says "No." Disappointed, we trade this robot in for an advanced model that fortuitously has following knowledge:

Living-Room(x) $\Rightarrow$ People-Gathering-Place(x)
Auditiorium(x) $\Rightarrow$ People-Gathering-Place(x)
Pool(x) $\Rightarrow$ People-Gathering-Place(x)
Found-in(x, y) $\wedge$ Comfortable(x) $\wedge$ People-Gathering-Place(y) $\Rightarrow$ Sit-On(People, x)
Has(x, Legs) $\Rightarrow$ Function(Support, Something, x)
Found-In(x, y) $\wedge$ Function(Support, Something, x) $\wedge$
People-Gathering-Place(x) $\Rightarrow$ Sit-On(People, x)
Sit-On(x, y) $\Rightarrow$ Function(Support, x, y)
Made-of(x, y) $\wedge$ Fabric(y) $\Rightarrow$ Comfortable(x)
Made-of(x, y) $\wedge$ Metal(y) $\Rightarrow$ Uncomfortable(x)

Giving this robot the same experience, we are pleased when it identifies the lawn chair as a chair and, upon inquiry, reports this:

"A chair is an object whose function is to support people and is found in places where people tend to gather"

This robot illustrates *constructive induction:* introducing new features into a concept description that are not present in any of the examples (Michalski, 1983). This can be done by using prior knowledge about the features in the examples to infer new features from those already present. This essentially means altering the knowledge representation space, and it is important for several reasons. First, many concept learning problems cannot be solved without a change of representation. In the "Chair" example, an important relevant feature is "Function," so the knowledge representation space has to be changed to include that feature. Second, constructive induction eases the teaching task. Rather than requiring the teacher to present all relevant and potentially relevant features of an example, the teacher

At this point, we might feel we have provided fine examples of "chair." Unknown to and unfortunately for the robot, the features that it has noticed have no apparent nmonalities or relation to the concept we are attempting to teach. This becomes clear en we point to a lawn chair, ask if it is a chair, and the robot (observing that it is made plastic, has legs, is found near the pool, is covered with snow and its comfort is known) says "No." Disappointed, we trade this robot in for an advanced model that tuitously has following knowledge:

Living-Room(x) ⇒ People-Gathering-Place(x)
Auditiorium(x) ⇒ People-Gathering-Place(x)
Pool(x) ⇒ People-Gathering-Place(x)
Found-in(x, y) ∧ Comfortable(x) ∧ People-Gathering-Place(y) ⇒ Sit-On(People, x)
Has(x, Legs) ⇒ Function(Support, Something, x)
Found-In(x, y) ∧ Function(Support, Something, x) ∧
    People-Gathering-Place(x) ⇒ Sit-On(People, x)
Sit-On(x, y) ⇒ Function(Support, x, y)
Made-of(x, y) ∧ Fabric(y) ⇒ Comfortable(x)
Made-of(x, y) ∧ Metal(y) ⇒ Uncomfortable(x)

Giving this robot the same experience, we are pleased when it identifies the lawn ir as a chair and, upon inquiry, reports this:

"A chair is an object whose function is to support people and is found in places where people tend to gather"

This robot illustrates *constructive induction:* introducing new features into a concept cription that are not present in any of the examples (Michalski, 1983). This can be done using prior knowledge about the features in the examples to infer new features from se already present. This essentially means altering the knowledge representation space, . it is important for several reasons. First, many concept learning problems cannot be ved without a change of representation. In the "Chair" example, an important relevant ture is "Function," so the knowledge representation space has to be changed to include t feature. Second, constructive induction eases the teaching task. Rather than requiring teacher to present all relevant and potentially relevant features of an example, the teacher

Chapter 2
Related Research

Learning by example has received a good deal of attention. This section reviews four systems that have most directly set the stage for this work: Winston's ARCH, Mitchell's LEX, Sammut and Banerji's MARVIN, and Michalski's INDUCE.

2.1. ARCH

Winston's seminal ARCH program (Winston, 1975) learned structural descriptions of concepts from examples. Structural descriptions represent structural relations between objects, such as "x is a part-of y." ARCH represented example descriptions, concept descriptions, and assertions about description elements in a semantic network. Examples consisted of a description of a concept and a classification (positive or negative).

ARCH initialized its concept description to the description of the first positive example. Subsequent examples were used to generalize the description or to form constraints on the description. First, ARCH determined a mapping between the description of the new example and the concept description. Next, ARCH determined and ranked the differences between the concept description and the example description. If the example was positive, then ARCH tried to replace the difference in the concept description by a generalization of the differences. These generalizations were determined by rules that incorporated knowledge about the type of relationship being generalized and the way in which it could be generalized. Some of these generalization rules were:

climb-tree heuristic: This heuristic was applied when the example and concept differed along an "a-kind-of" (AKO) dimension in a type hierarchy. The closest common parent of the AKO difference in the example and in the concept was chosen as the generalization. For example, if one example include Object A that was a-kind-of Square, and the concept description specified that Object A that was a-kind-of Oval, and the type hierarchy was the one shown below, then the a-kind-of dimension of A would be generalized to Polygon in the concept description, because Polygon is the closest common parent of Square and Oval.

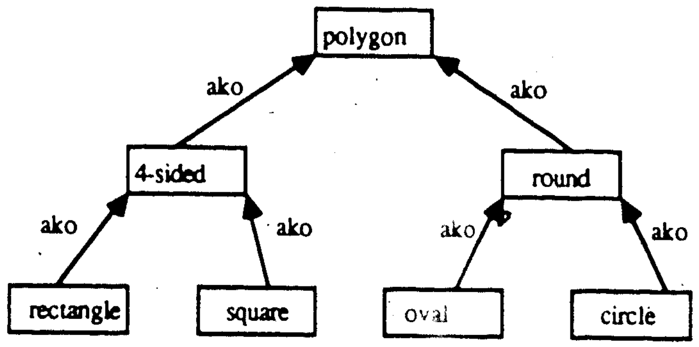

Figure 2.1 Type Hierarchy for ARCH

close-interval heuristic: This heuristic was applied when the example and concept differed along a linear dimension, such as location. The smallest interval including the interval or point corresponding to the differences in the example and concept was chosen as the generalization. For example, if Object A in the concept description was located at point (3, 4), and object A in the example was located at point (3.5, 5), then the location of A in the concept description would be generalized to ([3, 3.5], [4, 5]), which can be interpreted as "object A has x-coordinate in the interval [3, 3.5] and y-coordinate in the interval [4, 5]."

If a generalization was not found for a difference, then the difference was simply dropped from the concept description.

ARCH also included heuristics for specializing a description when a negative example was encountered. The differences were ranked, and the most important difference (possibly according to some evaluation function) was chosen. A difference could be either a feature of the example that was not present in the concept description, or a feature of the concept description that was not present in the example. For the first case, the "forbid" heuristic was used to assert that the feature was unacceptable in an instance of the concept, and for the second case, the "require" heuristic was used to assert that the feature was required in an instance of the concept. ARCH kept track of these forbidden and required features.

An important conceptual contribution of ARCH was the idea of "near-misses" to avoid the need to rank differences in the specialization step. A near-miss was an example that differed from the concept description on only one feature, so there could be no ambiguity about what was the most important aspect of the current example. A second important function of near-misses was to eliminate ambiguity during the matching process. Since near-misses were very close to the correct concept description, there was usually a candidate match that was clearly best. This avoided the problem of multiple subgraph isomorphisms, that is, the problem that arises when the training example matches the concept description in more than one way (Dietterich & Michalski, 1983). Winston's program was also notable for taking advantage of discourse conventions in student-teacher interactions, called *felicity conditions* (see also VanLehn, 1983). For example, if a teacher knew of ARCH's reliance on near-misses, the teacher would present only this kind of negative example. The assumption of a helpful teacher that knows the learner's knowledge state has been used in many later concept learning systems. As observed in our earlier example of teaching a robot the concept of "chair," the presentation and selection of examples may not be optimal or influenced by the learner's knowledge state.

ARCH efficiently learned conjunctive concepts from examples. ARCH represented prior knowledge in the form of type hierarchies, and used the climb-tree heuristic on this prior knowledge to introduce values into the concept description that were not in any of the examples presented. ARCH also addressed the problem of matching concept descriptions against examples, implementing a matcher that was able to find the correct match between the concept description and examples presented by a helpful teacher. However, ARCH did not represent any other kinds of prior knowledge, including relationships between different features, so ARCH could not do constructive induction. In addition, ARCH addressed the important problem of finding a match between a concept description and a training example.

2.2. LEX

LEX is a program that learns heuristic problem-solving strategies in the domain of symbolic integration (Mitchell, Utgoff & Banerji, 1983). Below are some methods, or operators, that are used in LEX:

OP1	$\int r * f(x)\,dx \Rightarrow r \int f(x)\,dx$
OP2	$\int u\,dv \Rightarrow uv - \int v\,du$
OP3	$1 * f(x) \Rightarrow f(x)$
OP4	$\int f1(x) + f2(x)dx \Rightarrow \int f1(x)\,dx + \int f2(x)\,dx$
OP5	$\int \sin(x)\,dx \Rightarrow -\cos(x) + C$
OP6	$\int \cos(x)\,dx \Rightarrow -\sin(x) + C$
OP7	$\int x\hat{}r\,dx \Rightarrow [x\hat{}r\,(r + a)] / (r + 1) + C$

Each of these operators has a condition specifying when it *can* be used. LEX learns when an operator *should* be used, e.g., a context in which it will be a good idea to use each operator. So one concept to be learned is "when it's best to apply OP1"; another concept is "when it's best to apply OP2." LEX calls these *heuristics*. The examples from which these heuristics are learned are solution paths in which the operators occurred. If an operator was used in a "good" solution path (determined by an evaluation function), then LEX uses the instantiated preconditions of the operator as a positive example of the heuristic. These conditions are expressed in a language based on a grammar for algebraic expressions containing indefinite integrals. The grammar is shown in the following diagram, with terminal nodes of the hierarchy corresponding to terminal symbols of the language, and rewrite rules expressed as edges between a higher level node and a lower level node.

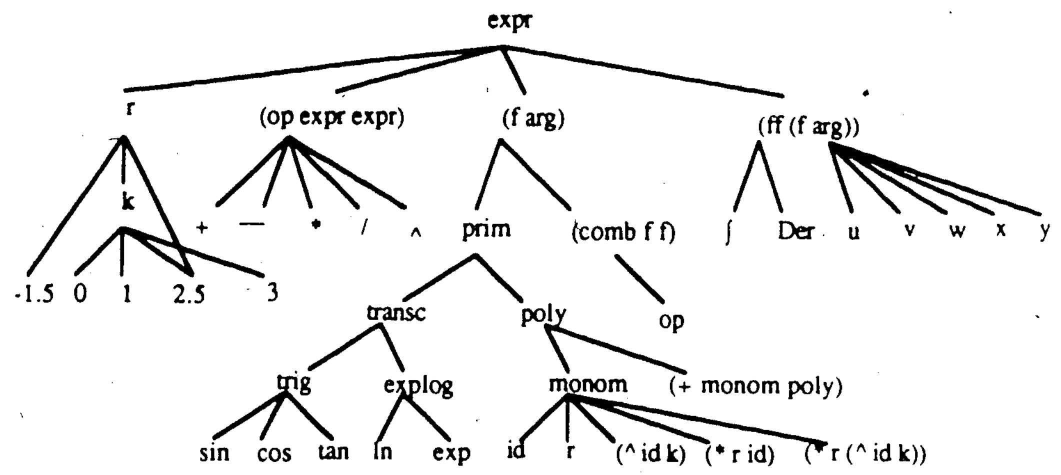

Figure 2.2 LEX Grammar

LEX consists of four modules: a problem generator, a problem solver, a critic, and a generalizer. The problem generator determines what a useful practice problem would be. The problem solver uses available heuristics to solve the problem. The critic analyzes the search steps performed in obtaining a solution, and extracts positive and negative examples of a good application of an heuristic from the solution. The generalizer uses these examples to propose and refine new domain-specific heuristics to improve performance on subsequent problems.

The important aspect of LEX is its use of *version spaces*, a method of compactly representing all alternative plausible descriptions of a concept. LEX maintains sets S and G that delimit the most specific and most general descriptions of an heuristic. For example, the following might be given to LEX's generalizer as a positive example of when OP2 was successfully applied:

$\int 3x \cos(x)\,dx$ -> Apply OP2 with $u = 3x$ and $dv = \cos(x)\,dx$

LEX will initialize the version space for this heuristic to

S: $\int 3x \cos(x)\,dx$ -> Apply OP2 with $u = 3x$ and $dv = \cos(x)\,dx$
G: $\int f1(x)\, f2(x)\,dx$ -> Apply OP2 with $u = f1(x)$ and $dv = f2(x)\,dx$

Note the most specific description of the heuristic is the actual example itself. The most general description is prior knowledge, describing all situations in which it is possible to apply the operator. *S* always contains exactly one element, but *G* may contain arbitrarily many elements. The sets *S* and *G* determine an upper and lower boundary on the possible versions of the heuristic: the operator is assumed to apply in any situation that is more specific than some element of *G* and more general than the element in *S*. For the above example, u can take any value that is a successor of f1(x) and an ancestor of 3x in the language hierarchy, and dv can take any value that is a successor of f2(x) and an ancestor of cos(x) in the language hierarchy, since all of these heuristics are within the area bounded by *S* and *G* in the version space.

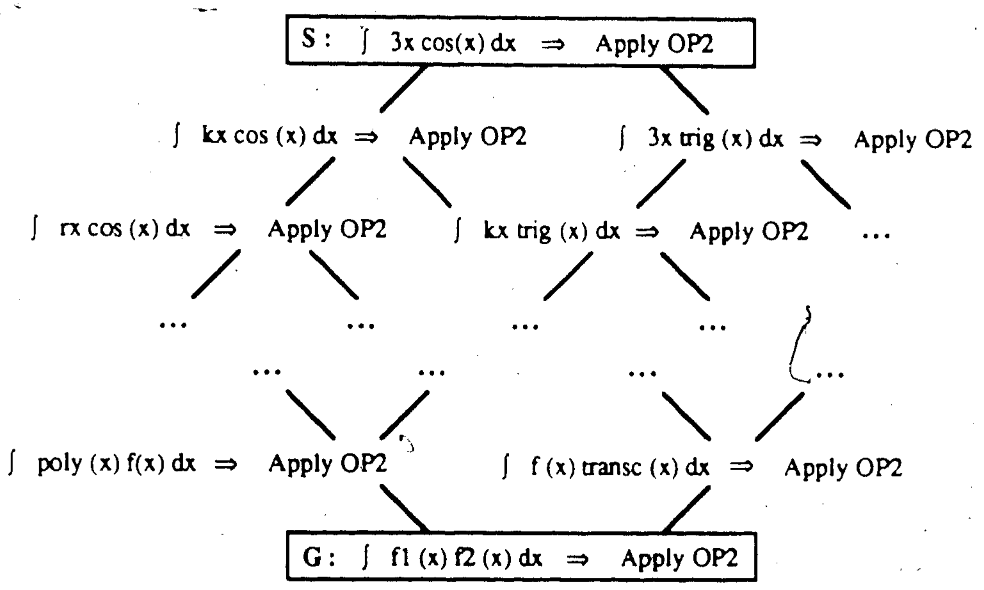
S : ∫ 3x cos(x) dx ⇒ Apply OP2
∫ kx cos (x) dx ⇒ Apply OP2
∫ 3x trig (x) dx ⇒ Apply OP2
∫ rx cos (x) dx ⇒ Apply OP2
∫ kx trig (x) dx ⇒ Apply OP2
∫ poly (x) f(x) dx ⇒ Apply OP2
∫ f (x) transc (x) dx ⇒ Apply OP2
G : ∫ f1 (x) f2 (x) dx ⇒ Apply OP2

Figure 2.3 Version Space

As the figure shows, *S* and *G* are compact ways of representing a space in which the true concept description lies. The idea is to use examples to shrink this space by moving *S* and *G* closer to each other until (if LEX is lucky), the space has only description in it. Positive examples cause *S* to become more generalized, and negative examples cause *G* to become more specialized. Suppose *s* is the element of *S*, and s_{term} is the binding of *term* in *s*. Then *S* is generalized by generalizing each s_{term} to the nearest common ancestor of the previous value of s_{term} and the binding of *term* in the positive example. Negative examples cause *G* to be specialized as follows: each element of *G* that is more general than the negative example is specialized in all possible ways that make it rule out the negative example. Each way of specializing a *g* corresponds to specialization of a different term of *g*, and is done as follows: suppose g_{term} is more general than the binding of *term* in the negative example. Then one way of specializing *g* consists of replacing g_{term} by the highest node in the language hierarchy that is not an ancestor of the binding of *term* in the negative example, and is a successor of g_{term}. For example, recall that OP2 is

OP2 $\int u\, dv \Rightarrow uv - \int v\, du$

and suppose *S* and *G* for its version space are:

S: ∫ 3x cos(x) dx -> Apply OP2 with u = 3x and dv = cos(x) dx
G: ∫ f1(x) f2(x) dx -> Apply OP2 with u = f1(x) and dv = f2(x) dx

Next, suppose LEX is presented with the following positive training instance:

∫ 3x sin(x) dx -> Apply OP2 with u = 3x and dv = sin(x) dx

G has only one element (call it *g*), and *g* is more general than this training instance. Therefore, LEX specializes *G* by replacing *g* with two specializations of *g*, one with a more specific binding for u, and the other with a more specific binding for dv. As Figure 2.2. shows, the highest node in the language hierarchy that is not an ancestor of 3x and is a successor of f1(x) is poly(x), so LEX adds the specialization

∫ poly(x) f2(x) dx -> Apply OP2 with u = poly(x) and dv = f2(x) dx

The highest node in the language hierarchy that is not an ancestor of 3x dx and is a successor of f2(x) dx is transc(x) dx, so LEX adds the specialization

∫ f1(x) transc(x) dx -> Apply OP2 with u = f1(x) and dv = transc(x) dx

The new *S* and *G* are:

S: ∫ 3x trig(x) dx -> Apply OP2 with u = 3x and dv = trig(x) dx
G: ∫ poly(x) f2(x) dx -> Apply OP2 with u = poly(x) and dv = f2(x) dx
  ∫ f1(x) transc(x) dx -> Apply OP2 with u = f1(x) and dv = transc(x) dx

The method of generalization described above is essentially the same as the climb-tree heuristic of Winston. A limitation of LEX is that this is the only way concepts can be generalized. Thus, LEX cannot find generalizations that do not have explicit representations as nodes in the type hierarchy. For example, "sin or cos" is a concept that is more general than "sin" and "cos," but if the immediate parent of "sin" and "cos" in the type hierarchy is "trig," then the system will generalize to "trig." Mitchell is aware of this problem and regards it as a fundamental problem of his system (Mitchell, Utgoff & Banerji, 19083). Another limitation of LEX is that prior knowledge can only be represented in single hierarchies. If we view each hierarchy as representing some "aspect" of the mathematical terms known to LEX, then LEX can only represent one aspect of each term. Finally, note that there are many ways of specializing G, so G could be as large as the number of terms raised to the number of negative examples. For example, if we had an operator with ten terms, then G might contain ten thousand elements after four negative examples. In practice, G does not become so large, because the presentation of positive examples will "rule out" members of G.

An interesting aspect of LEX's domain of integration is that there is no matching problem. In order to apply an operator to a formula, LEX must determine a binding of variables in the operator to terms in the formula. LEX is also able to learn disjunctive concepts by creating a new heuristic to cover examples that are not covered by any of its previous heuristics. Subsequently,

> This new heuristic will be updated by all subsequent negative instances associated with operator O, and by any subsequent positive instances associated with operator O and to which at least some member of its version space applies (Mitchell, Utgoff & Banerji, 1983).

However, Bundy et al. note that

> The new shell (heuristic) gets preferential treatment when it comes to allocating the new positive instances between the shells, whereas the old shell gets preferential treatment when it comes to allocating the old positive instances. There is no reason at all to assume that this is the correct division of the positive instances ... the ad hoc nature of the division of the positive examples makes it very unlikely that LEX will learn the correct disjunctive rule (Bundy, Silver & Plummer, 1985).

Despite the shortcomings of version spaces, version spaces remains noteworthy for its contribution to understanding how constraints on a description space can represent a concept, and how such constraints can be learned from positive and negative examples.

2.3. MARVIN

MARVIN is a program that learns concept descriptions by using prior knowledge and by askng questions (Sammut & Banerji, 1986). Concepts are represented as sets of *Horn clauses*, which are expressions in the first-order predicate calculus having form

$$P(X) \leftarrow Q(X) \,\&\, R(X) \,\&\, S(X)$$

An object X is identified as an instance of the concept P if the predicates Q, R, and S are true of X.

Since the predicate P(X) can occur in the right hand sides of other concept recognition rules, these rules can be used as prior knowledge to learn later concepts. MARVIN can recognize instances of concepts through a PROLOG theorem prover that essentially tries to prove the concept predicate, P(X), from an example description, which is a set of propositions.

Sammut and Banerji sometimes treat sets of propositions as descriptions. The description can be viewed as a predicate that denotes all instances in which the description formula is true. Sammut and Banerji define an *elaboration* of a set of propositions S under a set of rules R as the set including all propositions from S and one other proposition derived (by the PROLOG theorem prover) by a rule in R from S. The set All-Elaborations is the set of all propositons derivable from S by R.

MARVIN has two rules for transforming descriptions. The first is a replacement operation, where the antecedents of an instantiated rule are deleted from the description, and the consequents of an instantiated rule are added to the description. This is essentially the same as Michalski's rule of constructive generalization (Michalski, 1983). However, they also define a *specialization* rule that does multiple replacements in parallel. In other words, two replacement operations can be applied even though the first replacement would delete propositions that would be required for the second replacement to take place. This is important because it allows several features of an example to be inferred from a single feature of an example.

MARVIN also is able to generate examples and ask the teacher for a classification (positive or negative) of the example. These examples are descriptions called *trials*, and are denoted $T_0, T_1, \ldots T_n$. These trials are generated by application of different replacement or specialization operations on the initial trial T_0. MARVIN tries to find a trial that is more general than previous trials but is still *consistent* with the *target* T', or desired concept description. Since MARVIN does not know the target, the teacher's classification is used to determine if the trial is consistent with the target. Consistency is defined as follows:

Trial T is *consistent* with the target T' if any object that satisfies T also satisfies T'.

Sammut and Banerji do not test the consistency of a trial T by asking questions about each object that satisfies T but by asking questions about *crucial objects*. A crucial object is defined as an object that satisfies T_{i+1} but negates each element of

$$\text{All-Elaborations}(T_i) - \text{All-Elaborations}(T_{i-1})$$

where T_{i-1} is a generalization of T_i. This is useful since the number of crucial objects is less than the total number of objects, and if a crucial object is an instance of the target concept, then trial T_i is consistent.

MARVIN constructs crucial objects for a trial T_{i+1} as follows:

To construct an example for a trial T_{i+1}:

1. To construct an example from P & Q: Construct P, Construct Q.
2. To construct an example from an atomic predicate P when there is a set of clauses
   $\{P \leftarrow B_i\}$ in memory:
   a. Select a B_i such that
      $B_i \cap (\text{All-Elaborations}(T_i) - \text{All-Elaborations}(T_{i+1})) = \text{NIL}$
   b. Construct an example using the selected B_i.
3. To construct an example from an atomic predicate P when there is no clause
   $P \leftarrow B_i$, add P to the set of predicates representing the example.

There are number of points that can be made about MARVIN's approach to concept learning. First, MARVIN is essentially model-driven: only one example is input to the system, after which examples are created by MARVIN according to its model of prior knowledge, the set of Horn-clauses that make up its database. This means that all of the information that can appear in the concept must be implicit in the first example, because MARVIN has no means to accept further information from the teacher other than example classifications. While MARVIN is able to take an active role in learning concepts, the teacher is relegated to a passive role in teaching MARVIN. Further, this prevents the addition of negated predicates to the concept description, because any trial is a generalization of the initial trial. In Winston's terminology, "forbidden" features cannot be added to a concept description because no negative examples are presented. Since negated predicates or "forbidden" features are an important aspect of most concept descriptions, this limits the expressive power of the concept descriptions that MARVIN can learn.

Second, MARVIN requires All-Elaborations to be computed at each trial. This is essentially equivalent to considering the entire knowledge base, and it can be an expensive process if there are a large number of Horn-clauses that are relevant to the example. Further, this step means that MARVIN's concept learning problem could have been solved by treating the elaborated example descriptions as the examples, and using learning methods that do not use constructive induction.

Third, a large number of crucial objects could be generated for any one trial. Note that in step 2a. of the algorithm for constructing an example for a trial, that there are many choices for B_i. Since the algorithm is recursive, this means that there could be an exponential number of possible crucial objects.

Fourth, MARVIN's replacement and specialization rules require the deletion of the information that they use. This restricts the kind of descriptions that MARVIN can learn, as shown in the following example. Suppose MARVIN knew that "something with four legs and a top is a table," and was trying to learn the concept "a table with a steel leg" from the example "something with four legs and a top, where one of the legs is made of steel":

MARVIN's Knowledge:

$$\text{Table}(X) \leftarrow \text{Leg}(X, Y1) \mathbin{\&} \text{Leg}(X, Y1) \mathbin{\&} \text{Leg}(X, Y2) \mathbin{\&} \text{Leg}(X, Y3) \mathbin{\&} \text{Leg}(X, Y4) \mathbin{\&} \text{Top}(X)$$

Initial Example:

$$\text{Leg}(A, B1) \mathbin{\&} \text{Leg}(A, B2) \mathbin{\&} \text{Leg}(A, B3) \mathbin{\&} \text{Leg}(A, B4) \mathbin{\&} \text{Steel}(B4) \mathbin{\&} \text{Top}(A)$$

Desired Concept Description:

$$\text{Table-With-Steel-Leg}(X) \leftarrow \text{Table}(X) \mathbin{\&} \text{Leg}(X, Y1) \mathbin{\&} \text{Steel}(Y1)$$

MARVIN's best attempt:

$$\text{Table-With-Steel-Leg}(X) \leftarrow \text{Table}(X) \mathbin{\&} \text{Steel}(Y1)$$

MARVIN is unable to learn the desired concept because use of the "Table" knowledge in a replacement operation eliminates the knowledge about which leg is steel.

Fifth, MARVIN does not need to deal with the problem of matching examples to concept descriptions. Since MARVIN generates all but the first example that is presented, MARVIN knows how objects in the examples correspond to objects in the concept description. Since matching is a difficult problem for systems that learn from examples presented by a teacher, this is an important advantage of learning by asking questions.

In summary, despite MARVIN's weaknesses, it makes a number of important contributions to machine learning. First, it demonstrates a program whose language grows in descriptive power as it learns new concepts. Second, it can use those concepts as prior knowledge in learning new concepts. Third, it shows how a system can learn by asking questions of the teacher, and how this method avoids the matching problem. Finally, it implements these ideas in a program that can learn many useful concepts in a variety of domains.

2.4. INDUCE

INDUCE is a general purpose learning program developed by Michalski, Larson, Chilausky, and Stepp (Larson & Michalski, 1977; Michalski & Larson, 1978; Michalski & Chilausky, 1980; Michalski, 1983; Michalski & Stepp, 1983). INDUCE has been used in systems to classify soy-bean diseases and to learn hierarchies for classifying Spanish folk songs. INDUCE can be tailored for a specific domain by incorporating large amounts of domain-specific rules and knowledge within its representation scheme. Michalski approaches concept formation as an heuristic beam search through the space of concept descriptions for a set of descriptions that correctly classify the examples. Michalski uses forward inference on example descriptions to generate descriptors that are not present in any of the examples presented to the system, and has coined the term "constructive generalization" to describe this process.

INDUCE uses an annotated predicate calculus (APC) as a representation language, which comprises first-order predicate calculus, type constraints and second-order assertions about predicates. Some of the second-order information is implicitly represented by the use of specialized operators and connectives. For example, the connective "::>" represents the implication linking a concept description with a concept name (e.g., wings & flys & sings ::> bird). The second-order information increases the efficiency of the induction process and the quality of the inductive assertions.

Unlike first-order predicate logic, APC includes *relational selectors*. A relational selector is a special kind of ternary predicate where the second argument is one of {=, <, ≤, ≥, >}, and the third argument can be a complex expression such as "red ∨ blue." The syntax is somewhat different from standard first-order predicate logic syntax, and is best explained by example. To represent "the color of object-1 is red or blue," one can write

[Color(object-1) = red ∨ blue]

This is advantageous because INDUCE does not treat the arguments such as "=" and "red ∨ blue" as primitive, but can also reason about them and their relationships to other expressions.

Michalski uses rules of *generalization* and *specialization* to generate concept descriptions from examples. Specialization rules map a concept description D to a concept description D' that denotes a subset of the denotations of D. Generalization rules map a concept description D to a concept description D' that denotes a superset of the denotations of D.

Michalski also distinguishes between *selective* generalization and *constructive* generalization. A constructive generalization rule corresponds to what I term constructive induction: it introduces descriptors into the concept description that are not present in any of the examples presented to the learner. Selective generalization essentially creates a more general concept description by manipulating the descriptors that are present in the example. In the following examples, CTX stands for some arbitrary expression. One of Michalski's selective generalization rules is the *dropping condition* rule. The symbol for "generalizes to" in Michalski's notation is |<:

CTX & S ::> K |< CTX ::> K

For example, "[location = living-room] & [function = sitting] ::> chair" can be generalized to "[function = sitting] ::> chair." In Michalski's framework, *climbing generalization tree* is an example of selective generalization:

$$\begin{array}{lcl} CTX \,\&\, [L = a] \;::>\; K & & \\ \vdots & & \\ CTX \,\&\, [L = b] \;::>\; K & & \\ & |< & CTX \,\&\, [L = s] \;::>\; K \\ CTX \,\&\, [L = i] \;::>\; K & & \end{array}$$

In the above example, L is a *structured descriptor*. L represents some dimension of values that form a hierarchy; a, b, ... , i, and s can be mapped to some node in the hierarchy. The node corresponding to s is the lowest parent node whose descendants include nodes a, b, ... and i, in the hierarchy corresponding to L. For example, L might be a hierarchy of shapes, a might be a rectangle, b might be a square, c might be a triangle, and s might be a polygon:

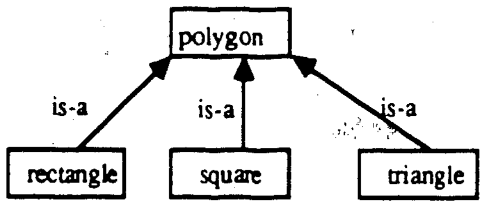

Figure 2.4. Type Hierarchy for Generalization

Michalski's constructive generalization rule is:

$$\begin{array}{lcl} CTX \,\&\, F_1 \;::>\; K & & \\ & |< & CTX \,\&\, F_2 \;::>\; K \\ F_1 \Rightarrow F_2 & & \end{array}$$

For example, "[has-seat(obj-1)] ::> chair and [has-seat(obj-1)] ⇒ [function = sitting]" can be generalized to "[function = sitting] ::> chair."

There is a subtle difference between climbing the generalization tree which Michalski classifies as selective generalization, and constructive generalization. In climbing the generalization tree, the descriptor or feature, shape, does not change, but rather is given a new value. In constructive generalization, an entirely new descriptor, "function," is introduced into the concept description. This relates to our earlier classification of learning situations according to whether the relevant feature, the relevant values, or possibly *neither* were present in the examples.

A limitation of Michalski's constructive generalization rule is that it is not complete. This is shown by the following example:

Initial Inductive Assertion :
Found-in(Obj-1, Place-1) & Living-Room(Place-1) ::> Chair(Obj-1)

Prior Knowledge:
Found-in(y, x) & Living-Room(x) ⇒ Comfortable(x)
Living-Room(x) ⇒ People-gathering-place(x)
People-gathering-place(x) & Found-in(x, y) & Comfortable(y)
⇒ Sit-on(People, y) & Function(y, Support, People)

Inductive Assertions Generated by Constructive Induction:
Comfortable(Obj-1) ::> Chair(Obj-1)
Found-in(Obj-1, Place-1) & People-gathering-place(Place-1)
::> Chair(Obj-1)

The following inductive assertion cannot be generated by the constructive generalization rule:

Found-in(Obj-1, Place-1) & People-Gathering-place(Place-1) &
Sit-on(People, Obj-1) & Function(Obj-1, Support, People)
::> Chair(Obj-1)

However, if we have a complete set of deductive rules of inference, we can infer logical consequences of the prior knowledge from which the constructive generalization rule will generate a somewhat better inductive assertion. For example, the prior knowledge from the previous example has as logical consequence:

Found-in(y, x) & Living-Room(x)
⇒ Found-in(y, x) & People-gathering-place(x)
& Sit-on(People, y) & Function(y, Support, People)

Using the above prior knowledge, the previous initial inductive assertion, and the constructive generalization rule, we can generate the inductive assertion:

People-Gathering-place(Place-1) & Sit-on(People, Obj-1) &
Function(Obj-1, Support, People) ::> Chair(Obj-1)

This is almost the desired inductive assertion, but note that there is no means of keeping Found-in(Obj-1, Place-1) in the inductive assertion, despite the fact that this information might be very desirable to the user. Further, if important facts were lost by application of constructive generalization before deduction was applied, the system would not be able to generate even this non-optimal assertion. An important point is the following: Michalski does not discuss in detail whether constructive induction is controlled or simply applied exhaustively to generate all possible descriptions.

2.5. Summary

These systems illustrate how systems can learn concepts from examples and by asking questions. ARCH demonstrates how knowledge in an example can be incorporated into a concept description as "forbidden" and "required" features, and how prior knowledge can be used to introduce new values into a concept description. LEX shows how a concept description can be represented as version space, and how the version space can constrain the learning process. MARVIN demonstrates how prior knowledge of the form of implication statements can be used to introduce new values and new features into a concept description, and how learned concepts can be used as prior knowledge on future learning tasks. Michalski presents a general framework and methodology for learning from examples, and defines the notion of constructive induction that is central to this research.

Of the four systems described, only MARVIN and INDUCE do constructive induction. The constructive induction mechanisms of MARVIN and INDUCE are not complete; they cannot generate concept descriptions that intuitively follow from the prior knowledge and presented examples. Further, both systems use constructive induction without considering specific goals of the learning task. Rather, constructive induction is used as an unconstrained generator of descriptions that are later evaluated by the system.

Chapter 3

Approach

3.1. Scope and Assumptions

The main goal of this research is to explore a means by which constructive induction could be guided and controlled during learning. As noted earlier, this is an open issue for "knowledge-intensive" learning and this work makes some preliminary steps towards understanding this issue in a carefully circumscribed learning context. This section presents the simplifying assumptions made for this work and describes where LAIR falls along the dimensions analyzed in section 1.3. The implications of these assumptions are then discussed in more detail.

LAIR learns conjunctive concept descriptions from both positive and negative examples. Furthermore, LAIR learns incrementally. The most important implication of incremental learning is that not all the positive and negative examples are available for inspection. For LAIR, this also translates into a limited memory for previously-seen examples. LAIR's goal is to learn a concept description that can be used to correctly classify the presented positive and negative examples. LAIR includes both generalization and specialization rules for transforming concept descriptions, which enable it to do constructive induction. It infers new knowledge on an as-needed basis to meet the information requirements of the learning process. LAIR is provided with the object and part correspondences among the examples. Each object or part in one example corresponds to no more than one object or part in another example. LAIR learns from examples of high *integrity*. High integrity examples are unambiguous and thus provide reliable guidance for learning (Barr & Feigenbaum, 1982). LAIR uses a subset of the first-order predicate calculus as a representation language. This subset includes lambda abstraction but does not include universal quantification or disjunction. LAIR uses prior knowledge about the domain in the form of production rules and frames, and extends its knowledge through the learning process.

The above assumptions restrict the both kinds of situations in which LAIR can learn, and simplify LAIR's learning task. First, because object and part correspondences are provided, LAIR does not need to match objects and parts between the concept description and the examples. This contrasts with one of the main accomplishments of Winston's ARCH program.

LAIR's restricted description language makes the learning task easier, at the expense of restricting the kinds of concepts LAIR can learn. For example, since LAIR does not include disjunction, LAIR cannot directly learn that an object is liftable if it is "seen to be lifted or it is light and graspable." To learn this kind of disjunctive concept, LAIR would have to learn each disjunct as a separate, conjunctive concept. LAIR also cannot directly learn the description of "tallest" defined as "taller than every other object." However, LAIR can learn an equivalent description of "tallest" by first learning the concept "not-tallest" defined as "something else is taller." Thus, the restricted language requires that LAIR sometimes learn intermediate concepts before learning the desired concept.

LAIR's goal is to recognize and classify patterns (examples) as being members of a concept. Specifically, LAIR tries to learn a concept description that it can prove true of each of the positive examples, and none of the negative examples, using the description of the examples and its prior knowledge as axioms. Thus, any two descriptions that classify examples the same way are equivalent to LAIR: their form or content does not matter. There could be many features true of all the positive examples and none of the negative examples, but not all of this information may be useful to LAIR for classifying objects. For example, if the feature "can be used to write" were not inferrable of all of the positive examples of "pen," then this feature would be dropped from the concept description, since it could not be used in the classification task. Note that "can be used to write" could be true of all the positive examples, but some deficiency in LAIR's knowledge about the world makes it unable to infer this feature for all of the positive examples. This issue is outside LAIR's scope. For LAIR, a description is important for what it can do for *classification*.

LAIR's classification goal also has implications for the order in which concepts must be learned. LAIR tries to infer features of a concept description using its prior knowledge. If the prior knowledge is incomplete in a way that prevents an important feature from being inferred, then LAIR might not be able to learn a correct concept description. Clearly, if LAIR is missing a concept that is critical to the *subsequent* learning of a new concept, then it may not be able to learn that subsequent concept. In addition, negative features could be incorrectly added to the concept description if LAIR lacked the knowledge to prove the features true of some example. While LAIR does not require that presentation of *examples* be ordered, LAIR does require that learning of *concepts* be partially ordered. This is the same point made earlier about the completeness of the knowledge base: a newly-learned concept becomes (possibly essential) "prior knowledge" for the next learning task. In general, this is an important theme that dominates much of machine learning today—what you can learn depends in part on what you already know. The issue is *how* that knowledge can be effectively applied to the learning task.

LAIR learns from examples of high integrity. Example integrity is determined by three factors: error, order of presentation, and presence of irrelevant information. Two kinds of error can occur: *measurement* error and *classification* error. Measurement error occurs when the description of an example is incorrect, e.g., if a 2.5" cylindrical object is incorrectly observed to be a 2.3" cylindrical object. Classification error occurs when a positive example is presented to the system as a negative example—a *false negative* example—or a negative example is presented to the system as a positive example—a *false positive* example. LAIR assumes that examples are free of error. In systems that classify their own examples by means of a critic module, it may be difficult to guarantee this. For example, LEX uses a critic to create examples from solution traces. LEX's critic is not infallible, so some negative examples may be presented as positive examples.

The second factor affecting example integrity is order of presentation. Depending on the learning algorithm, different sequences of example presentation may result in slower learning or even failure to learn. LAIR learns faster for some orders of presentation, but it is still able to learn correctly for any order of presentation, given that enough examples are presented.

The third factor affecting example integrity is presence of irrelevant information. For example, LAIR might be presented with the example of a suitcase as an instance of the concept "liftable." The description of the suitcase might include irrelevant information, such as its color, its manufacturer, and its style. LAIR can learn from examples that contain irrelevant information.

Incremental learning has implications for extending LAIR to find its own correspondences between example objects and parts. Given only a limited number of examples, it may be impossible to determine the correct correspondence between objects and parts. For example, LAIR might be presented "three people sitting on a bench; from left to right, a poet, a violinist, and a novelist." The second example might be "three people sitting on a bench; from left to right, a novelist, an oil-worker, and a doctor." If LAIR was trying to learn the concept "three people sitting on a bench; from left to right, a writer, a person, and another person," then the poet in the first example should be matched with the novelist in the second example. On the other hand, if LAIR was trying to learn th[illegible]ncept "a novelist sitting on a bench" then the novelist in the first example should be matched with the novelist in the second example. Clearly, the correct match depends on what concept is being learned.

LAIR's incremental learning interacts with its strategy of doing inference on an as-needed basis. This interaction may require that examples be presented more than once, or that additional examples be presented that are similar to previous examples. For example, LAIR does inference on an as-needed basis, but it may not know precisely what is needed when an example is presented. This can adversely affect learning in two ways. First, LAIR may specialize a concept description with a predicate that cannot be proven true of some past positive example that has been forgotten. LAIR cannot detect this error unless that old positive example is presented again or unless that predicate is unprovable for some new positive case as well. Second, LAIR may not draw all the needed inferences from a negative example because its current concept description may be specific enough to rule out the previously-seen negative example. Later, after that example is forgotten, the concept description may be generalized in response to a positive example. This can result in the concept description being too general to rule out the negative example. LAIR cannot detect this error unless that negative example, or one similar to it, is encountered again.

LAIR's assumptions and design have implications for backtracking in learning systems. A backtracking system backtracks to previous descriptions and reconsiders the examples that it has seen since the previous description was formed. LAIR has been designed to be an incremental learner with a limited memory for previously seen examples, and is not designed to support backtracking. Backtracking can be used to solve the problem of choosing an incorrect match, and to eliminate the requirement that examples be presented again, or further examples be presented that are similar to previously-seen examples. However, in most simple backtracking schemes, much of the information and work done since the backtracking point is discarded and must be repeated from the next backtracking point. Further, it is often difficult to extend simple backtracking schemes to intelligent backtracking schemes that can avoid this redundant work. LAIR's underlying assumption is that it is easier to extend its design to handle matching than it is to extend a simple backtracking approach to one that avoids this redundant work.

LAIR's present design allows it to learn only one concept at a time. Later sections on design and implementation present more detail on the kind of bookkeeping LAIR does during learning. The basic idea is that LAIR puts various markers on predicates in its knowledge base while it is learning. After LAIR has learned the correct concept description, the teacher may ask LAIR to add the learned concept to the knowledge base. If this is not requested, then LAIR will clear out all the markers when it is asked to learn a new concept. Thus, LAIR cannot learn multiple concepts simultaneously even when they do not build on one another. However, if concepts were independent, LAIR could be extended to learn multiple concepts, by tagging the markers with the name of the concept to which they correspond.

LAIR also assumes that only a finite number of new features can be introduced in the examples and that only a finite number of features are inferrable from the examples. This requirement is necessary for LAIR's concept description to converge to correctness, and is discussed further in section 3.8.

3.2. Overview

The general framework and approach taken in LAIR can be summarized as follows. First, climbing the generalization tree (which introduces new values for already-seen descriptors) and constructive generalization (which introduces new descriptors) are both regarded as cases of *constructive induction*. Second, constructive induction is achieved via inference and deduction. Third, constraints that are inherent in the learning task are used to limit the knowledge retrieval and deduction. An important result of the second and third points is that LAIR has no constructive induction rule among its concept transformation rules. The constructive induction rule is derived from inference rules and concept revision rules that use description constraints. This approach also uses an "information conservation" principle when generalizing a concept description. Information conservation attempts to retain knowledge about a descriptor that will be dropped from an incorrect concept description by checking to see whether some inference derivable from that descriptor may be relevant to the correct concept description. Finally, I have a relaxed definition of a "correct" concept description. In LAIR, a transformation of a concept description might make the concept description incorrect with respect to previously-seen (but forgotten) examples, but it is guaranteed to converge on the correct description with enough examples. Descriptions are always correct with respect to previously-seen and remembered examples.

3.3. The Knowledge Representation

The knowledge used by LAIR can be represented in a subset of first-order predicate calculus. Rules are used to represent prior knowledge, lambda expressions are used to represent concept descriptions, and propositions are used to represent example descriptions.

3.3.1. Descriptions

Descriptions are used to describe concepts and examples. A description is of the form

$\lambda x[P_{11}(x) \wedge \ldots \wedge P_{1n}(x) \wedge \neg P_{21}(x) \wedge \ldots \wedge \neg P_{2m}(x)]$

where each $P_{ij}(x)$ is of form $P^n(t_1, \ldots, t_n)$ where P^n is an n-adic predicate symbol chosen from some fixed, finite set of such symbols, one of the t_i is x, and the other t_i are constants or skolem functions of x. I will informally refer to the unnegated P_{ij} as Requireds (REQs) and the negated P_{ij} as NOTs.

Examples are denoted by constants, e.g. Ex-1. These constants are interpreted as "situations" rather than any particular object in the example. Examples are described by applying a description to the constant, e.g.

λx[Pos(Arch, x) ∧ Block(f(x), x) ∧ Block(g(x), x) ∧ Ontop(f(x), g(x), x)](Ex-1)

meaning

"Ex-1 is a positive example of the concept Arch; in Ex-1, there are two blocks, one of which is on top of the other."

Equivalently, this can be written

Pos(Arch, Ex-1) ∧ Block(f(Ex-1), Ex-1) ∧ Block(g(Ex-1), Ex-1) ∧
Ontop(f(Ex-1), g(Ex-1), Ex-1)

One of the predicates in each example description must be Pos($Concept, x$). This predicate means "x is a positive example of $Concept$." If the example is positive, Pos($Concept, x$) occurs as a REQ, i.e. in the form Pos($Concept, x$). If the example is negative, it occurs as a NOT, i.e. in the form ¬Pos($Concept, x$). Pos($Concept, x$) is not allowed in concept descriptions, because it can be used to form a correct, but trivial, definition of the concept. Example descriptions are assumed to be complete with respect to the fixed set of predicates from which descriptions are constructed and the knowledge base. This will be discussed further in section 3.4., but for example descriptions the important point is that NOTs are not represented explicitly: they are simply omitted from the description.

3.3.2. The Knowledge Base

LAIR has a knowledge base that constitutes the prior knowledge it has before the beginning of the learning task. This knowledge is what LAIR accesses during constructive induction. Prior knowledge is represented in the form of implications. These are constrained to contain a single unnegated consequent, and one or more negated or unnegated antecedents, where antecedents and consequents may contain constants, universally quantified variables, or skolem functions. For example,

> ∀example∀boy[Boy(boy, example) ∧ Tall(boy, example) ∧ Handsome(boy, example) ∧ ¬Gauche(boy, example) ∧ Has(boy, eyes(boy), example) ∧ Eyes(eyes(boy), example) ∧ Blue(eyes(boy), example) ⇒ Likes(Susan, boy, example)]

has unnegated antecedents "Tall(boy, example)," "Has(boy, eyes(boy), example)," "Blue(eyes(boy), example)," and "Handsome(boy, example)," negated antecedent "¬Gauche(boy, example)," and consequent "Likes(Susan, boy, example)," where "example" and "boy" are universally quantified variables, "Susan" is a constant, and "eyes(boy)" is a skolem function of "boy." Note that every predicate must have a variable corresponding to examples, in this case, *example*. The use of the variable *example* to denote an example rather than the constant "Susan" is motivated by the potential need to use

"Susan" in several examples. For example, "Susan" might occur in both a positive and negative example of the concept "a girl who is with her lover," if in the first example she is with her lover and in the second example she is not with her lover. The general form of rules is therefore

$$\forall \text{example} \forall y_1 \ldots \forall y_n [P_{11}(\text{example}) \wedge \ldots \wedge P_{1n}(\text{example}) \wedge \neg P_{21}(\text{example}) \wedge \ldots \wedge \neg P_{2m}(\text{example}) \Rightarrow Q(\text{example})]$$

where each P_{ij}(example) and Q(example) is of form $P^n(t_1, \ldots ,t_n)$ where P^n is an n-adic predicate symbol, one of the t_i is example, and the other t_i are one of $y_1 \ldots y_n$ or are constants or skolem functions of example or $y_1 \ldots y_n$. I will informally refer to these implications as *rules*. Concepts that LAIR learns are translated into this form, corresponding to *Description* ⇒ *Concept*, and can be added to the knowledge base. They are then treated as prior knowledge for later learning tasks.

3.4. Deductive Inference

LAIR uses deductive inference to determine whether "constraints" are satisfied, to classify examples, and to extend example descriptions. LAIR uses the OPS4 (Forgy, 1979) production system and a proof task module to do these tasks. Recall that rules are of the following form:

$$\forall x \forall y_1 \ldots \forall y_n [P_{11}(x) \wedge \ldots \wedge P_{1n}(x) \wedge \neg P_{21}(x) \wedge \ldots \wedge \neg P_{2m}(x) \Rightarrow Q(x)]$$

The predicates $P_{ij}(x)$ (containing free variables corresponding to the universally quantified example, $y_1 \ldots y_n$) can be instantiated as follows. Let $\sigma = \{t_1/v_1, t_2/v_2, \ldots, t_n/v_n\}$. The pair t_i/v_i means that term t_i is substituted for variable v_i throughout. Each occurrence of a variable must have the same term substituted for it, and no variable can be replaced by a term containing that same variable. Let $P(x)\sigma$ denote the result of substituting those constants for those variables in a predicate $P(x)$. For example,

$$P(y_1, f(y_2), x)\{A/y_1, B/y_2, \text{Ex-1}/x,\} = P(A, f(B), \text{Ex-1})$$

LAIR has the following rules of inference. Suppose *Ex* is the example, *KB* is a set of rules, and *Ex-Description* = $\lambda x[P_{11}(x) \wedge \ldots \wedge P_{1n}(x) \wedge \neg P_{21}(x) \wedge \ldots \wedge \neg P_{2m}(x)]$ is the description of *Ex*.

1. $KB \wedge Ex\text{-}Description(Ex) \vdash P_{1j}(Ex)$
2. $KB \wedge Ex\text{-}Description(Ex) \vdash \neg P_{2j}(Ex)$
3. If $KB \wedge Ex\text{-}Description(Ex) \vdash P(Ex)$ and

   $KB \wedge Ex\text{-}Description(Ex) \vdash Q(Ex)$ then

   $KB \wedge Ex\text{-}Description(Ex) \vdash P(Ex) \wedge Q(Ex)$

   where $P(Ex) = P(x)\{Ex/x\}$, $Q(Ex) = Q(x)\{Ex/x\}$, where $P(x)$ and $Q(x)$ have the form of a REQ, a NOT, or a description.
4. If for some rule in *KB*

   $\forall x \forall y_1 \ldots \forall y_n[P_{11}(x) \wedge \ldots \wedge P_{1n}(x) \wedge \neg P_{21}(x) \wedge \ldots \wedge \neg P_{2m}(x) \Rightarrow Q(x)]$

   and some substitution σ, for each unnegated antecedent

   $KB \wedge Ex\text{-}Description(Ex) \vdash P_{1j}(x)\sigma$

   and for each negated antecedent

   $KB \wedge Ex\text{-}Description(Ex) \vdash \neg P_{2j}(x)\sigma$

   then

   $KB \wedge Ex\text{-}Description(Ex) \vdash Q(x)\sigma$
5. If $P(x)$ has the form of a REQ, and

   $KB \wedge Ex\text{-}Description(Ex) \nvdash P(Ex)$

   then

   $KB \wedge Ex\text{-}Description(Ex) \vdash \neg P(Ex)$

   This rule is based on the assumption mentioned in section 3.3.1. that the example descriptions are complete with respect to the fixed set of predicates from which descriptions are constructed and the knowledge base. The example descriptions are assumed to be consistent, so NOTs in example descriptions can be omitted, since they can be derived by this rule.

LAIR can use these rules *extend* example descriptions. Extending an example description is defined as adding new predicates as REQs or NOTs to the example description. For example, if

$Ex\text{-}Description = \lambda x[\mathrm{Pos}(\mathrm{Stable}, x) \wedge \mathrm{Bottom}(f(x), g(x), x) \wedge \text{No-Bumps}(g(x), x)]$
$KB = \{\forall x,y[\text{No-Bumps}(y, x) \Rightarrow \mathrm{Flat}(y, x)]\}$
$Q(\mathrm{Ex}) = \lambda x\mathrm{Flat}(g(x), x)](\text{Ex-3})$
$Ex = \text{Ex-3}$

then $KB \wedge Ex\text{-}Description(Ex) \vdash Q(Ex)$, so

$\lambda x[\mathrm{Pos}(\mathrm{Stable}, x) \wedge \mathrm{Bottom}(f(x), g(x), x) \wedge \text{No-Bumps}(g(x), x)$
$\wedge\ \mathrm{Flat}(g(x), x)](\text{Ex-3})$

is an extended description of Ex-3.

LAIR uses concept descriptions to classify examples as follows. Suppose *D* is a description of *Concept*, *Ex* is an example to be classified, *Ex-Description*(*Ex*) is the description of *Ex*, and *KB* is the knowledge base. If

$KB \wedge Ex\text{-}Description(Ex) \vdash D(Ex)$

then LAIR assumes that according to *D*, *Ex* is a positive instance, otherwise that *Ex* is a negative instance.

3.5. Correctness of Concept Descriptions

A concept description *D* of *Concept* is defined to be *correct* iff it can be used to correctly classify the positive and negative examples of the concept with respect to the knowledge base. Notice that the same description may be correct or incorrect depending on the contents of the knowledge base: if LAIR is lacking some piece of vital information, it may not be able to use the description to correctly classify the examples. More formally, *D* is correct iff for every positive example *Ex* of the concept,

$KB \wedge Ex\text{-}Description(Ex) \vdash D(Ex)$

and for every negative example *Ex* of the concept,

$KB \wedge Ex\text{-}Description(Ex) \nvdash D(Ex)$

where *Ex-Description*(*Ex*) is the description of *Ex*, and *KB* is the knowledge base. Almost by definition, however, an incremental learner has not seen all the positive examples of a concept. Further, an incremental learner may not remember all previously-seen examples, or may be lacking some important piece of knowledge necessary to determine correctness. Therefore, *relative correctness* is defined as follows. A description *D* is relatively correct at time *t* iff for every *remembered* positive example *Ex* of the concept,

$$KB \wedge \mathit{Ex\text{-}Description}(Ex) \vdash D(Ex)$$

and for every *remembered* negative example Ex of the concept,

$$KB \wedge \mathit{Ex\text{-}Description}(Ex) \nvdash D(Ex)$$

where *Ex-Description*(*Ex*) is the description of *Ex*, and *KB* is the knowledge base. LAIR remembers only one past positive example and the current example, but this definition holds true for any set of remembered examples.

3.6. Constraints on the Concept Description

LAIR keeps track of whether a predicate is provable for some positive example and whether it is unprovable for some positive example. These form important constraints on the concept description and on the concept revision rules. The important properties of constraints are described below:

Drop(*P*, *t*) — LAIR has inferred at or prior to time *t* that *P* is false of some positive example, and has stored this information with the predicate in the knowledge base. Informally, P is "dropped." Note that if **Drop**(*P*, *t*), then for some positive example *Ex* of the concept,

$$KB \wedge \mathit{Ex\text{-}Description}(Ex) \vdash \neg P(Ex)$$

Some(*P*, *t*) — LAIR has inferred at or prior to time *t* that *P* is true of some positive example, and has stored this information with the predicate in the knowledge base. Note that if **Some**(*P*, *t*), then for some positive example *Ex* of the concept,

$$KB \wedge \mathit{Ex\text{-}Description}(Ex) \vdash P(Ex)$$

One constraint on the concept description is that it cannot include a predicate *P* that is dropped. Otherwise, the concept description would be incorrect. This can be proved as follows. Suppose **Drop**(P, t). Then by the **Drop** axiom, for some positive example Ex, $KB \wedge Ex\text{-}Description(Ex) \vdash \neg P(Ex)$. Suppose the concept description, D, includes P. Then D is of form $\lambda x[P_1(x) \wedge \ldots \wedge P_n(x) \wedge P(x)]$. Clearly $KB \wedge Ex\text{-}Description(Ex) \nvdash D(Ex)$, so by definition, D is incorrect.

An analogous constraint on the concept description is that it cannot contain the negation of a predicate that has been found to be true of any positive example, i.e., a predicate P for which **Some**(P, t) is true. The proof is similar.

3.7. Inductive Inference

LAIR uses the following concept revision rules, described in terms of:

constraint—conditions that must be true for the rule to be used.

initial-description—initial description of the concept.

transformation—either "specializes to" or "generalizes to."

revised-description—revised description of the concept

postconditions—conditions assumed to hold after the rule is used.

In the following rules, "Curr(*Concept*, *Ex*, *t*)" denotes "*Ex* is the current example of *Concept* at time *t*," and "Past(*Concept*, *Ex*)" denotes "*Ex* is the past, positive example of *Concept* remembered by LAIR." The first presented example is assumed to be positive, and becomes the past example. Note that the D's that are concept descriptions, and the P's, and Q's that are added or dropped from concept descriptions, are assumed not to have the variable x occurring in them. If-then renditions of these rules are included for readability.

- The add-REQ rule—adds an unnegated predicate (a "REQuired") to the concept description

<u>constraint</u>:

$\neg$**Drop**(P, t) $\wedge$
Curr($Concept$, $Curr$, t) $\wedge$ Past($Concept$, $Past$, t) $\wedge$
$[[KB \wedge Curr\text{-}Description(Curr) \vdash P(Curr)]$
$\quad \vee \neg \text{Pos}(Concept, Curr)] \wedge$
$[KB \wedge Past\text{-}Description(Past) \vdash P(Past)]$

initial-description:	D
transformation:	specializes to
revised-description:	$\lambda x[D(x) \wedge P(x)]$
postconditions:	$\mathbf{Some}(P, t)$

If a predicate has never been **Drop**'d from the concept description
& it can be proven true of all the remembered positive examples
Then add the predicate to the concept description
and remember it is true of **Some** positive example

- The add-NOT rule—adds a negated predicate (a "NOT") to the concept description.

constraint:	$\neg\mathbf{Some}(P, t) \wedge$
	$\mathrm{Curr}(Concept, Curr, t) \wedge \mathrm{Past}(Concept, Past, t) \wedge$
	$[KB \wedge Curr\text{-}Description(Curr) \vdash P(Curr)] \wedge$
	$\mathrm{Neg}(Concept, Curr) \wedge$
	$[KB \wedge Past\text{-}Description(Past) \vdash \neg P(Past)]$
initial-description:	D
transformation:	specializes to
revised-description:	$\lambda x[D(x) \wedge \neg P(x)]$

If a predicate has not been proved of **Some** positive examples
& it can be proven true of the current negative example
& it cannot be proven true of the remembered past positive example
Then add its negation to the concept description

- The drop-REQ rule—drops a REQ from the concept description.

constraint:	$\mathrm{Curr}(Concept, Curr, t) \wedge \mathrm{Past}(Concept, Past, t) \wedge$
	$[[KB \wedge Curr\text{-}Description(Curr) \vdash \neg P(Curr)]$
	$\wedge \mathrm{Pos}(Concept, Curr)] \vee$
	$[KB \wedge Past\text{-}Description(Past) \vdash \neg P(Past)]$
initial-description:	$\lambda x[D(x) \wedge P(x)]$
transformation:	generalizes to
revised-description:	D
postconditions:	$\mathbf{Drop}(P, t)$

If a predicate in the concept description cannot be proven true of the remembered positive examples
Then drop it from the concept description
& remember it has been **Drop**'d

- The drop-NOT rule—drops a NOT from the concept description.

constraint:	$\mathrm{Curr}(Concept, Curr, t) \wedge \mathrm{Past}(Concept, Past, t) \wedge$
	$[[KB \wedge Curr\text{-}Description(Curr) \vdash (Curr)]$
	$\wedge \mathrm{Pos}(Concept, Curr)] \vee$
	$[KB \wedge Past\text{-}Description(Past) \vdash P(Past)]$
initial-description:	$\lambda x[D(x) \wedge \neg P(x)]$
transformation:	generalizes to
revised-description:	D
postconditions:	$\mathrm{Some}(P, t)$

If	a negated predicate P in the concept description can be proven true of any remembered positive example
Then	drop $\neg P$ from the concept description
&	remember P is true of **Some** positive example

Notice LAIR does not have a constructive induction rule! This rule would be written in our framework as follows:

- The constructive induction rule—applies domain knowledge to generalize a concept description.

constraint:	$\forall x[P(x) \Rightarrow Q(x)]$
initial-description:	$\lambda x[P_1(x) \wedge \ldots \wedge P_n(x) \wedge P(x)]$
transformation:	generalizes to
revised-description	$\lambda x[P_1(x) \wedge \ldots \wedge P_n(x) \wedge Q(x)]$

If	the concept description includes a predicate P
&	the knowledge base contains a rule P implies Q
Then	drop P from the concept description
&	add Q to the concept description

The concept revision rules given earlier are essentially equivalent to the constructive induction rule. Their relationship will be discussed further in section 3.9.

3.8. Relative Completeness, Preservation of Relative Correctness, and Convergence

Three important properties of these rules as implemented in LAIR are "relative completeness," "preservation of relative correctness," and "convergence." Intuitively, these mean that the rules do not make the description incorrect with respect to the remembered examples, that the rules can always find a correct description if it is possible, and that the concept description eventually becomes correct. This section defines these properties formally, and proves, given certain important assumptions, that they are true of LAIR.

A description revision rule R *preserves relative correctness* iff given description D, if D is relatively correct then R revises D to a description D' that is also relatively correct.

The add-REQ rule can be shown to preserve relative correctness as follows. Suppose D is a description that is relatively correct at time t, *Past* is the past example, *Curr* is the current positive example, and the add-REQ rule revises D to $D' = \lambda x[D(x) \wedge P(x)]$. Since D is relatively correct at time t,

$KB \wedge \textit{Past-Description}(\textit{Past}) \vdash D(\textit{Past})$ and

$KB \wedge \textit{Curr-Description}(\textit{Curr}) \vdash D(\textit{Curr})$

Since add-REQ was applied at time t, its constraint must be true, so

$KB \wedge \textit{Past-Description}(\textit{Past}) \vdash P(\textit{Past})$ and

$KB \wedge \textit{Curr-Description}(\textit{Curr}) \vdash P(\textit{Curr})$

Therefore,

$KB \wedge \textit{Past-Description}(\textit{Past}) \vdash D(\textit{Past}) \wedge P(\textit{Past})$ and

$KB \wedge \textit{Curr-Description}(\textit{Curr}) \vdash D(\textit{Curr}) \wedge P(\textit{Curr})$

Since LAIR remembers only *Past* and *Curr*, for every seen and remembered positive example of the concept,

$KB \wedge \textit{Ex-Description}(\textit{Ex}) \vdash D(\text{Ex}) \wedge P(\text{Ex})$

Therefore, D' is relatively correct at time t.

The proof of preservation of relative correctness for the add-NOT rule is similar. The drop-REQ and drop-NOT rules trivially preserve correctness because they are constrained to apply in situations where the initial description is incorrect.

Drop rules have another important property: they never drop a predicate that is in a correct concept description. This can be proven as follows. Suppose $D = \lambda x[P_{11}(x) \wedge \ldots \wedge P_{1n}(x) \wedge \neg P_{21}(x) \wedge \ldots \wedge \neg P_{2m}(x)]$ is a correct concept description. If P_{1j} could be drop-REQ'd, then at some time t, its constraint **Drop**(P_{1j}, t) would be true. Therefore, for some positive example Ex, $KB \wedge Ex\text{-}Description(Ex) \vdash \neg P(Ex)$. Therefore $KB \wedge Ex\text{-}Description(Ex) \nvdash D(Ex)$, so D is incorrect, a contradiction. Therefore, P_{1j} could not be dropped. The proof that the P_{2j} cannot be drop-NOT'd is similar.

Completeness is defined relative to LAIR's ability to use the concept description to classify examples. Suppose there exists a correct description that be proven true of all the positive examples and false of all the negative examples. Then a set of concept ~~revision~~ rules is *relatively complete* iff a correct description D' can be derived from every intermediate description D derivable from any set of positive and negative examples, using the rules. Note correctness is an equivalence relation over concept descriptions, and that LAIR is not necessarily able to find any particular correct description, but only some correct description given one exists. Further, relative completeness does not mean that LAIR will find a correct concept description, but rather that LAIR could find a correct description from any point in the learning process given the required examples are presented.

Relative completeness will be proven as follows. First, I will prove a correct concept description can be derived from the null description using the concept revision rules and the deductive rules. Second, the first result will be used as the basis case to prove a correct concept description is derivable from any intermediate description. LAIR is assumed to have a helpful teacher who presents the required examples.

Suppose D' is a description that is correct, and D' can be used to correctly classify instances of a concept. Then D' is of form

$$\lambda x[P_{11}(x) \wedge \ldots \wedge P_{1n}(x) \wedge \neg P_{21}(x) \wedge \ldots \wedge \neg P_{2m}(x)]$$

and for every positive example Ex of $Concept$,

$$KB \wedge Ex\text{-}Description(Ex) \vdash D'(Ex)$$

where *Ex-Description*(*Ex*) is the description of *Ex* presented to the system. Therefore, for every P_{1j}(x) and every positive example *Ex*,

$$\overline{KB} \wedge Ex\text{-}Description(Ex) \vdash P_{1j}(Ex)$$

Recall LAIR remembers a past positive example *Past* and a current example *Curr*. The helpful teacher is requested for a positive example, so that *Curr* is positive. Then by the above,

$$\text{Curr}(Concept, Curr, t) \wedge \text{Past}(Concept, Past, t) \wedge$$
$$\text{Pos}(Concept, Curr) \wedge [KB \wedge Curr\text{-}Description(Curr) \vdash P_{1j}(Curr)] \wedge$$
$$KB \wedge Past\text{-}Description(Past) \vdash P_{1j}(Past)$$

Also, since D' is correct, **Drop**(P_{1j},t) cannot be true for any time t. Thus the constraint of add-REQ is satisfied, and P_{1j} can be added to the concept description. This process can be repeated for each P_{1j} to add them all to the concept description.

I will now show that each of the P_{2j}(x) can be added to the concept description. Each of the P_{2j}(x) is assumed to be necessary to rule out some negative example. Otherwise, they could be dropped from the concept description without losing correctness. Therefore, for some negative example *Ex*,

$$KB \wedge Ex\text{-}Description(Ex) \vdash P_{2j}(Ex)$$

If *Ex* is made the current example, then

$$\text{Curr}(Concept, Ex, t) \wedge \text{Neg}(Concept, Ex) \wedge [KB \wedge Curr\text{-}Description(Ex) \vdash P_{2j}(Ex)]$$

Also, since D' is correct, P_{2j} cannot be provable of any positive example, so if *Past* is the past example, then

$$\text{Past}(Concept, Past, t) \wedge \neg\textbf{Some}(P, t) \wedge [KB \wedge Past\text{-}Description(Past) \vdash \neg P(Past)]$$

Thus the constraint of add-NOT is satisfied, and $\neg P_{2j}$ can be added to the concept description. This process can be repeated for each P_{2j} to add them all to the concept description as NOTs.

Therefore, a correct concept description D' can be derived by some sequence of concept revision rules starting from scratch—the null description. I will now show that D' (or another correct concept description) can be derived from any intermediate concept description D_n, by induction on number k of applications of concept revision rules to derive the intermediate description, D_n, from the null description. By the previous result, the basis case is true, where $n = 0$. Assume if D_n has been derived by n applications of concept revision rules then D' can be derived from D_n. I will show that D' can be derived from any description D_{n+1} derived in $n + 1$ applications of concept revision rules. Clearly, D_{n+1} can be derived in one step from some description D_n that is derived in n steps. There are four rules that can apply to D_n to derive D_{n+1}: add-REQ, add-NOT, drop-REQ, or drop-NOT. Suppose D_{n+1} is derived by add-REQ from D_n, so $D_{n+1} = \lambda x[D(x) \wedge P(x)]$. If P is in D', then P does not need to be added later, so the step in the derivation of D' from D_n where P was added can be omitted. If P is not in D' and P can be dropped then D' can be derived from D_{n+1} by the same sequence used to derive D' from D_n with this extra step. If P cannot be dropped, then the constraint of drop-REQ must be false, so

$$\text{Past}(\mathit{Concept}, \mathit{Past}, t) \wedge [\mathit{KB} \wedge \mathit{Past\text{-}Description}(\mathit{Past}) \vdash P(\mathit{Past})]$$

must be true, and for every positive example that could become the current positive example, i.e. every positive example Ex of the concept,

$$\mathit{KB} \wedge \mathit{Ex\text{-}Description}(\mathit{Ex}) \vdash P(\mathit{Ex})$$

If the same derivation of D' from D_n is applied to D_{n+1}, the final concept description is D' with one extra predicate, P:

$$\lambda x[D'(x) \wedge P(x)]$$

This concept description can also be shown correct as follows. Since D' is correct, for every positive example Ex

$$\mathit{KB} \wedge \mathit{Ex\text{-}Description}(\mathit{Ex}) \vdash D(\mathit{Ex})$$

and for every negative example Ex,

$KB \wedge \textit{Ex-Description}(Ex) \nvdash D(Ex)$

Therefore, for every positive example *Ex*

$KB \wedge \textit{Ex-Description}(Ex) \vdash D(Ex) \wedge P(Ex)$

and for every negative example *Ex*,

$KB \wedge \textit{Ex-Description}(Ex) \nvdash [D(Ex) \wedge P(Ex)]$

so this concept description is also correct. A similar argument shows that application of add-NOT transforms D_n into a D_{n+1} from which D' or another correct concept description can be derived.

I will now show that if D_{n+1} is derived from D_n by applying drop-REQ or drop-NOT then a correct concept description can be derived from D_{n+1}. Assume that D_{n+1} is derived from D_n by applying drop-REQ. Then the constraint of drop-REQ must be true:

$\text{Curr}(Concept, Curr, t) \wedge \text{Past}(Concept, Past, t) \wedge$

$[[[KB \wedge \textit{Curr-Description}(Curr) \vdash \neg P(\text{Curr})] \wedge \text{Pos}(Concept, Curr)] \vee$

$[KB \wedge \textit{Past-Description}(Past) \vdash \neg P(Past)]]$

Therefore, for some positive example,

$KB \wedge \textit{Ex-Description}(Ex) \vdash \neg P(Ex)$

But then *P* cannot be in *D'*, or *D'* would not be correct. Therefore, *P* is not in *D'*, so *D'* can be derived from D_{n+1} by applying the same sequence of concept revisions that was applied to D_n. A similar argument shows that application of drop-NOT transforms D_n into a D_{n+1} from which *D'* or another correct concept description can be derived. Therefore, by the induction theorem, *D'* or another correct concept description can be derived from any intermediate description D_n, for all *n*. The most important consequence is that the system does not need to backtrack to a previous point in the learning process, since it can derive a correct concept description from any intermediate description. Of course, LAIR might have to be presented with an example it has already seen (or a similar example), but it will not have to discard the work it has done since last seeing that example.

LAIR's concept description converges to correctness under a number of assumptions. First, the number, *m*, of inferrable features and new features are introduced in examples, is assumed to be finite. This implies that the number of examples is finite, since only a finite number of examples can be created from a finite number of features in our representation. Second, the examples are assumed to be presented in the following way. Suppose the examples are $Ex_1 \ldots Ex_n$. The teacher presents each of examples $Ex_1 \ldots Ex_n$ once. If no change occurred in the concept description, then the teacher stops, otherwise repeats this process. The order of presentation can be changed between iterations, as long as each example is presented once, and the number of times an example is presented could be greater than one, provided it is less than some fixed bound *k*. If no change in the concept description occurs for one of these iterations, then the concept description is correct. LAIR's learning procedures (described later) restore relative correctness of the concept description after each example is presented. Therefore, if no change occurs in the concept description for some iteration, then the concept description is correct. Since a dropped feature cannot be added to a concept description, there are only 4*m* changes that can be made to the concept description. If the concept description was not correct, then at least one change must have been made during each iteration. Therefore, after 4*m* iterations, the concept description will be correct.

The assumption that only a finite number of new features is introduced in the examples is required for the following reason. Suppose that we had an infinite number of features, and some infinite subset, $F_1, F_2, \ldots$, was inferrable of the remembered, positive examples. If the *i*-th positive example were $F_1 \wedge F_2 \wedge \ldots \wedge \neg F_{i-1} \wedge F_i$, then LAIR's concept description would never converge, since it would drop F_{i-1} and add F_i for each example.

3.9. Derivation of Constructive Induction

The deductive rules and relaxed concept revision rules derive the constructive induction rule, providing the initial description is relatively correct. Since LAIR's rules preserve relatively correctness, each intermediate concept description is true of the past positive example. The add-REQ rule is relaxed by removing the **Drop** constraint, and the drop-REQ rule is relaxed by removing all its constraints.

Assume $\lambda x[P_1(x) \wedge \ldots \wedge P_n(x) \wedge P(x)]$ is the initial description and the constraint of the constructive induction rule is a rule in the knowledge base: $\forall x[P(x) \Rightarrow Q(x)]$. Then by relative correctness, if *Past* is the past example, then

$$KB \wedge \textit{Past-Description}(\textit{Past}) \vdash \lambda x[P_1(x) \wedge \ldots \wedge P_n(x) \wedge P(x)](\textit{Past})$$

Therefore, $KB \wedge \textit{Past-Description}(\textit{Past}) \vdash P(\textit{Past})$, and so *Past-Description* can be extended to *Past-Description* $\wedge$ *P*. Since $\forall x[P(x) \Rightarrow Q(x)]$ is in *KB*,

$$KB \wedge \textit{Past-Description}(\textit{Past}) \wedge P(\textit{Past}) \vdash Q(\textit{Past})$$

Similarly, for the positive current example *Curr*,

$$KB \wedge \textit{Curr-Description}(\textit{Curr}) \wedge P(\textit{Curr}) \vdash Q(\textit{Curr})$$

Therefore the relaxed constraint of add-REQ is satisfied, and $\lambda x[P_1(x) \wedge \ldots \wedge P_n(x) \wedge P(x)]$ can be specialized to $\lambda x[P_1(x) \wedge \ldots \wedge P_n(x) \wedge P(x) \wedge P(x) \wedge Q(x)]$. This specialized concept description can be generalized by the relaxed drop-REQ rule to $\lambda x[P_1(x) \wedge \ldots \wedge P_n(x) \wedge Q(x)]$, which is just the concept description produced by the constructive induction rule. Therefore, LAIR's relaxed rules derive the constructive induction rule.

There are clearly descriptions that LAIR's unrelaxed rules cannot produce. However, there are good reasons for not generating these descriptions. If the **Drop** constraint on add-REQ was violated, then add-REQ would generate an incorrect description. If the constraint on drop-REQ was violated, then drop-REQ would drop a feature that was true of all remembered positive examples, losing specificity of the concept description. Since LAIR's concept revision rules are relatively complete, inability to generate these descriptions does not seem like a serious shortcoming. In contrast, the undecomposed constructive induction rule is not relatively complete, as shown by the examples in sections 2.3 and 2.4.

3.10. Summary

This section has outlined the general approach and framework taken in LAIR. LAIR learns concept descriptions in a conjunctive description language. For incremental learning systems, concept descriptions can be characterized by relative correctness with respect to the remembered examples and the time at which the description is formed. Constructive induction is based on inference and deduction; it is controlled by constraints that limit the knowledge retrieval and deduction. LAIR has no "constructive induction" rule among its concept transformation rules; constructive induction is derived from inference rules and concept revision rules that use description constraints. LAIR's rules are relatively complete, so LAIR can always find a correct, usable example (if one exists). LAIR's design as an incremental concept learner with finite memory for previous examples precludes backtracking or maintaining multiple concept descriptions. In LAIR, a transformation of a concept description might make the concept description incorrect with respect to previously-seen (but forgotten) examples, but it is guaranteed to converge on the correct description with enough examples. Descriptions are always correct with respect to previously-seen and remembered examples.

LAIR makes a number of simplifying assumptions that limit the situations to which it is applicable. Object correspondences are provided for it. This avoids many of the problems that typically call for solved by backtracking techniques. Thus, if LAIR were extended to do matching, it might have to backtrack because it chose an incorrect match, even if it would not have to backtrack because it chose the wrong concept revision rule. Second, LAIR may have to see examples (or similar examples) more than once to correct an incorrect specialization decision, or to correct a failure to specialize. However, LAIR does not backtrack through all the changes it has made to the concept description (in the sense of undoing them) since last seeing that example. This feature is due to LAIR's concept transformation rules plus the assumptions made about (a) the integrity of the examples (positive cases are never presented as negative ones, or vice versa), and (b) the fact that only a finite number of new features are being introduced in the examples.

Chapter 4

LAIR's Knowledge Base

4.1. Introduction

LAIR's knowledge base consists of frames that represent knowledge about examples, concept descriptions, constraints on concept descriptions, and prior or learned knowledge about the domain. LAIR uses a production system to do deductive inference and to perform the various learning tasks.

LAIR's hybrid knowledge representation scheme is motivated by the importance of matching the representation scheme with the processes that will use the knowledge. Several kinds of knowledge are compatible with frame-based representations: knowledge about predicates such as constraints, instantiations, related predicates, related implications, and concept descriptions; prior knowledge in the form of implication statements; meta-knowledge about prior knowledge such as whether it has been activated for inference. Storing, modifying, and retrieving information are easily handled within a frame-based representation. On the other hand, much of the knowledge about tasks performed in learning are best represented as IF-THEN rules: actions to be taken to classify examples, to revise concept descriptions, and to prove propositions. Production system languages offer useful control structures for organizing tasks. Some languages like OPS4 provide pattern matching algorithms that can be exploited for a task like this.

Frames represent information in the following form (Winston & Horn, 1981):

```
(frame
     (slot$_1$ (facet$_{11}$ (value$_{111}$)
                        .
                     (value$_{11n}$))
                .
           (facet$_{1p}$. .  )
         .
     (slot$_r$  ))
```

Frames in our system are essentially record structures with composite slot-facet fields. Values can be either primitive values or other frames. The important idea is that all of the information about a frame can be quickly accessed, including related frames, if they are stored as values on a frame.

Production systems consist of three basic components: a set of IF-THEN rules, a data base often called working memory (WM), and an interpreter for the rules (Davis & King, 1977). An IF-THEN rule consists of a condition and an action; working memory consists of a set of symbol strings. The interpreter matches the conditions of the rules against the data base according to some prespecified scheme, and executes the actions of some subset of the rules that have matched. Often, the IF-THEN rules correspond to implication statements, working memory consists of some set of explicitly true propositions, and the interpreter asserts the consequent of the IF-THEN rules if the antecedent is satisfied by working memory. In this way the invocation of rules can be viewed as a chained sequence of *modus ponens* actions. The production system, written in OPS4 (Forgy, 1979), guides the system between its various tasks, and does deductive retrieval of information.

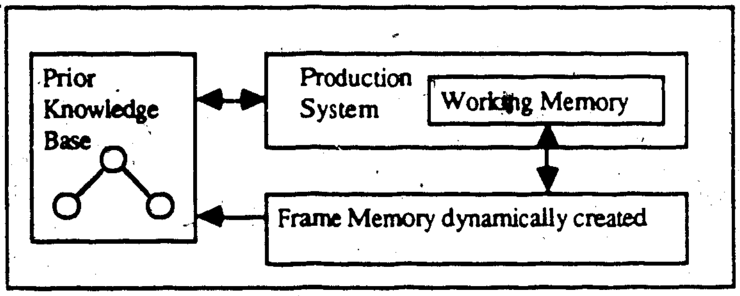

Figure 4.1 System Architecture

Figure 4.1 presents a schematic view of how knowledge is represented in the different formalisms and how they interact. Frames serve as a long term memory in which prior knowledge and meta-knowledge about the system can be represented. Frames can be dynamically created as a function of learning in a domain; frames representing learned concepts may be moved permanently to long term memory. The production rules can create, modify, and delete frames by means of actions that are performed when a rule fires.

Actions can invoke quite specialized methods for searching spaces of generalizations, updating constraints, creating predicate frames, and deriving instances of a predicate. Derivation of predicate instances, in turn, involves the creation of rules from knowledge stored on frames, and activating these rules on working memory.

4.2. Working Memory

Working memory is a set of propositions that represents either control information about the task or descriptive information about an example. The control information is used to communicate information between various tasks, and to do standard programming operations such as sequencing, iteration, and conditionals. Example descriptions, which are conjunctions of propositions, are represented in working memory as sets of propositions each of which is assumed to be true. The syntax of propositions in working memory is different from the standard logical syntax: the form is

(example (Predicate-name arg_1 arg_2 ... arg_n*))*

where *example* is a constant symbol denoting an example, *Predicate-name* is a predicate symbol denoting an $n+1$ predicate, and arg_1 arg_2 ... arg_n *example* are arguments of the predicate. For example, the proposition

Loves(John, Mary, example1)

(meaning "John loves Mary is true of example1") would be represented in working memory as

(example1 (Loves John Mary))

The conjunctive description

Loves(John, Mary, example1) $\wedge$ Tall(John, example1)

would be represented in working memory as

(example1 (Loves John Mary)), (example1 (Tall John))

This representation is motivated by the need to easily collect all of the propositions true of a particular example, and to easily distinguish descriptive information from control information.

Control information, such as "the current task is unification," is represented as a working memory proposition (task-is-classification). The system is able to distinguish between this kind of information and the descriptive information described previously on the basis of syntactic differences. However, control information can sometimes involve descriptive information, such as "the current subtask is to conserve the information in the predicate λxLoves(John, Mary, x)," which is represented as the proposition (unify subtask (Loves John Mary)).

4.3. Frame Memory

The two main kinds of frames are frames representing predicates and frames representing rules.

4.3.1. Frames Representing Predicates

Knowledge is organized around predicates over examples. Frames corresponding to these entities are created during learning as instantiations of more general predicate frames stored in the prior knowledge base. There are two types of predicate frames: *most-general-predicate* frames and *less-general-predicate* frames. A most-general-predicate frame corresponds to a predicate expression that has more than one argument, e.g., λx,y,z[body(x, y, z)]. A less-general-predicate frame corresponds to a predicate expression that has only one argument, e.g. λz[body(a(z), b(a(z)), z)]. Since less-general-predicates are the "building blocks" of concept descriptions, constraints on concept descriptions are stored on these frames. The most important slots on a less-general-predicate frame are:

DEFN (definition)
Lambda expression defining the predicate; e.g., $\lambda z[body(a(z), b(a(z)), z)]$.

MOST-GEN-PRED (most general predicate)
The frame representing the most general predicate corresponding to the predicate. The most general predicate is obtained by replacing all constants and skolem functions in a predicate definition by variables; for example, the most general predicate of $\lambda z[body(a(z), b(a(z)), z)]$ is $\lambda x,y,z[body(x, y, z)]$.

PROPOSITIONS
Propositions that are instances of the predicate.

SOME
T iff the predicate is true of some positive example.

REQ
T iff the predicate is an unnegated predicate in the concept description.

NOT
T iff the predicate is a negated predicate in the concept description.

DROPPED
T iff the predicate is not derivable for some positive example.

REQ-SPACE
T iff the predicate has been considered as a possible REQ in a concept description during the current subtask.

NOT-SPACE
T iff the predicate has been considered as a possible NOT in a concept description during the current subtask.

A typical less-general-predicate frame might be:

Body-001

DEFN	$\lambda x[body(a(x), b(a(x)), x)]$
PROPOSITIONS	body(a(Ex-1), b(a(Ex-1)), Ex-1), body(a(Ex-2), b(a(Ex-2)), Ex-2)
REQ	T
SOME	T
NOT	NIL
DROPPED	NIL
MOST-GEN-PRED	Body-000
REQ-SPACE	T
NOT-SPACE	NIL

The second kind of predicate frame, the most-general-predicate frame, is used to organize knowledge about relationships between predicates and rules. Less-general-predic[illegible] inherit inference rules from most-general-predicates. Most-general-predicates can inherit propositions from less-general-predicates. The most important slots of a most-general-predicate frame are:

DEFN (definition)
Lambda expression defining the predicate; e.g., $\lambda x,y,z[body(x, y, z)]$.

LESS-GEN-PREDS (less general predicates)
Inverse of the MOST-GEN-PRED slot on less-general-predicate frames.

CONSEQUENT-OF
Rules in which the predicate, or one of its less general predicates, is a consequent.

ANTECEDENT-OF
Rules in which the predicate, or one of its less general predicates, is an unnegated antecedent.

NEG-ANTECEDENT-OF
Rules in which the predicate, or one of its less general predicates, is a negated antecedent.

A typical most-general-predicate frame might be:

Body-000

DEFN	$\lambda x,y,z[body(x, y, z)]$
LESS-GEN-PREDS	Body-001
CONSEQUENT-OF	NIL
ANTECEDENT-OF	hot-0094, open-vessel-0100
NEG-ANTECEDENT-OF	NIL

4.3.2. Rule Frames

In the above frame, hot-0094 and open-vessel-0100 refer to rule frames. Rule frames represent knowledge corresponding to logical implications. The most important slots on a rule frame are:

CONSEQUENTS
Consequents of the implication.

ANTECEDENTS
Unnegated antecedents of the implication.

NEG-ANTECEDENTS
Negated antecedents of the implication.

A typical rule frame might be:

graspable-0104	
CONSEQUENTS	graspable(y, z)
ANTECEDENTS	cyl(y, z), small(y, z), light(y, z), body(x, y, z)
NEG-ANTECEDENTS	hot(y, z)

This rule corresponds to the implication statement

$$\forall x,y,z[\mathrm{cyl}(y, z) \wedge \mathrm{small}(y, z) \wedge \mathrm{light}(y, z) \wedge \mathrm{body}(x, y, z) \wedge \neg\mathrm{hot}(y, z) \Rightarrow \mathrm{graspable}(x, z)]$$

"If something is light with a small, cylindrical body that is not hot, then it's graspable."

4.4 Summary

LAIR's knowledge base is a hybrid scheme that consists of frames, working memory elements, and productions. Knowledge migrates to the production system as needed; learned knowledge is stored in the frame base. The hybrid approach is motivated by the need to match the knowledge to be represented with the processes that use the knowledge.

Chapter 5

How LAIR Learns

5.1. Design

LAIR consists of the following task modules:

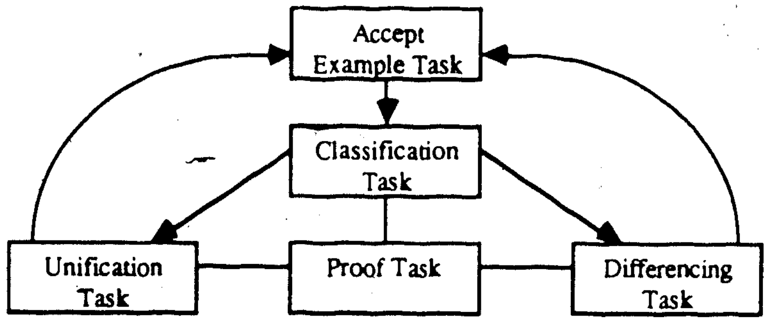

Figure 5.1. The System Modules

1. **Accept Example Task** — Clears out the previous example from memory and accepts the description of the next example from the teacher. This involves updating working memory and frame memory with the example description, and adding features of the example to the concept description if they do not violate relaxed constraints on the add-REQ or add-NOT rules.

2. **Classification Task** — Determines the relative correctness of the concept description with respect to the remembered examples (the past positive example and the current example) by calling the Proof Task to prove the description true of the examples. If the description is not relatively correct, then this task tries to identify the causes of failure. If the current example is positive, the description is generalized to restore relative correctness by dropping REQ's, then the Unification Task is called to conserve the lost information. If the current example is negative, the Differencing Task is called to restore relative correctness.

3. **Unification Task** — Specializes the concept description to conserve information lost during the Classification Task by add-REQ'ing new features that can be proven

of all the remembered, positive examples. This involves inference, which is handled by the Proof Task.

4. **Differencing Task** — Specializes the concept description when it incorrectly classifies a negative example as a positive case, to restore relative correctness. This involves add-REQ'ing or add-NOT'ing new features that correctly rule out the current negative example as a positive instance of the concept.
5. **Proof Task** — This task tries to prove a predicate true of the examples remembered by the system. It is used by the Classification Task, Unification Task, and Differencing Task.

5.2. Learning the Concept of a Cup

To describe how LAIR learns, we will explain learning a concept description for "cup."[1] Assume LAIR has just learned a number of rules relating to the concepts "hot," "stable," "open-vessel," "graspable," and "liftable." LAIR remembers only one previous example, which must be positive, plus the current example. Initially, the concept description is λex[], interpreted as "anything."

The Accept Example Task attempts to add information from the examples to the concept description that do not violate the constraints on the description. The first example is "cha-cup +," so LAIR forms the following description of the past positive example:

Pos(cup, Ex-1) ∧ bottom(a(Ex-1), c(a(Ex-1)), Ex-1) ∧ flat(c(a(Ex-1)), Ex-1) ∧ concavity(a(Ex-1), e(a(Ex-1)), Ex-1) ∧ cylinder(b(a(Ex-1)), Ex-1) ∧ small(b(a(Ex-1)), Ex-1) ∧ body(a(Ex-1), b(a(Ex-1)), Ex-1) ∧ was-lifted(a(Ex-1), Ex-1) ∧ upwards(e(a(Ex-1)), Ex-1)

"A positive example of a cup is something with a flat bottom, an upwards-pointing concavity, a small, cylindrical body, that was lifted."

[1]This learning task is inspired by the ANALOGY program (Winston, Binford, Katz & Lowry, 1983) that learns the physical description of a cup from a functional definition, examples, and precedents.

Since initially there are no constraints on the concept description, all of the features can be added to the concept description.

λex[bottom(a(ex), c(a(ex)), ex) ∧ flat(c(a(ex)), ex) ∧ concavity(e(a(ex)), ex) ∧ cylinder(b(a(ex)), ex) ∧ small(b(a(ex)), ex) ∧ body(a(ex), b(a(ex)), ex) ∧ was-lifted(a(ex), ex) ∧ upwards(e(a(ex)), ex)]

The second example, "typical-cup +," is accepted, resulting in the following description of the current example:

Pos(cup, Ex-2) ∧ bottom(a(Ex-2), c(a(Ex-2)), Ex-2) ∧ flat(c(a(Ex-2)), Ex-2) ∧ concavity(e(a(Ex-2)), Ex-2) ∧ cylinder(b(a(Ex-2)), Ex-2) ∧ small(b(a(Ex-2)), Ex-2) ∧ body(a(Ex-2), b(a(Ex-2)), Ex-2) ∧ upwards(e(a(Ex-2)), Ex-2) ∧ handle(a(Ex-2), Ex-2) ∧ light(a(Ex-2), Ex-2)

"A positive example of a cup is something with a flat bottom, an upwards-pointing concavity, a small, cylindrical body, that is light and has a handle."

The Accept Example Task revises the concept description to include the features handle(a(ex), ex) and light(a(ex), ex). Note that these features are added although they are not true of all of the remembered positive examples. The Accept Example Task does not check this constraint, because it will be checked by the Classification Task. Since all of the features in the concept description must be checked for relative correctness, it is simpler to check this constraint during classification.

The Classification Task now determines relative correctness by trying to prove the concept description true of the remembered positive examples. The Classification Task finds that the concept description is not relatively correct, and restores relative correctness by dropping was-lifted(a(ex), ex), handle(a(ex), ex), and light(a(ex), ex). Although the concept description is relatively correct, in some sense information has been "lost."

The Unification Task attempts to conserve the lost information by specializing the revised concept description with inferences it can make from the dropped REQ's. Essentially, this means "Does this descriptor, which I've decided to drop, imply something else that is provable of both the current example and the past positive example(s)?" Information conservation is important in LAIR for several reasons. First, when the current example is positive, then LAIR is essentially learning from positive-only examples. Systems that learn from positive-only examples must have methods to avoid over-generalization. Information conservation avoids over-generalization by compensating for the loss of information in generalization by a specialization step. Further, specialization is focused only on descriptors that can be inferred from the dropped REQ's.

The Unification Task searches for inferrable descriptors by looking at *rule-generalizations* of the dropped REQ's. A predicate *P* is a rule-generalization of a predicate *Q* if (1) *Q* has a MOST-GEN-PRED that is ANTECEDENT-OF a rule that has *P* as a CONSEQUENT, or (2) *P* is a rule-generalization of another predicate that is a rule-generalization of *Q*. The relationship between *P* and *Q*, for case (1), is shown in figure 5.2.

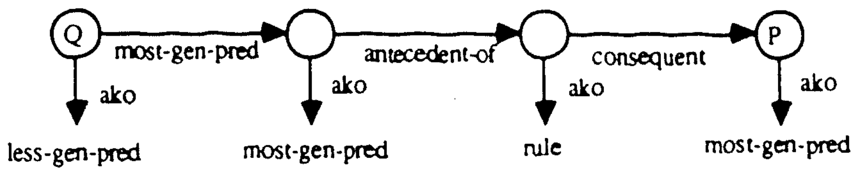

Figure 5.2. Rule Generalizations

The Unification Task creates a subtask for each dropped REQ by initializing a "REQ-boundary" to the dropped REQ. Each subtask is an heuristic search of the space of descriptors to find a less general predicate of a rule-generalization of the dropped REQ that can be add-REQ'd to the concept description. The subtask chooses a REQ-boundary element *P* and tries to add-REQ it to the concept description. This involves inference: *P* must be proven true of *Past* and *Curr* because they are both positive, remembered examples. If a less general predicate of *P* can be add-REQ'd, then the subtask terminates successfully. Otherwise, *P* is deleted from the REQ-boundary and *P*'s rule-generalizations are added to the REQ-boundary if *P* (or a less general predicate than *P*) satisfies the following constraint:

$\exists$ex[Remembered(ex, t) $\wedge$ Pos($Concept$, ex) $\wedge$ P(ex) $\wedge$ $\neg$REQ(P, t)]
"P is true of a remembered, positive example of the concept, and is not currently a REQ."

If the REQ-boundary is empty, then the subtask terminates unsuccessfully, otherwise a new REQ-boundary element is chosen and the process repeated.

For the "typical-cup +" example, the first REQ-boundary is initialized to {was-lifted(a(ex), ex)}. Since was-lifted(a(ex), ex) was dropped, it cannot be add-REQ'd, and is removed from the REQ-boundary. However, was-lifted(a(ex), ex) has a rule-generalization, liftable(y, ex), derived from a rule learned earlier:

If an object was lifted
Then that object is liftable

Since was-lifted(y, ex) satisfies the REQ-boundary constraint, its rule-generalization can be added to the REQ-boundary, yielding {liftable(y, ex)}. LAIR calls the Proof Task to try and prove instances of liftable(y, ex). The proposition liftable(a(Ex-1), Ex-1) is proven by using the above inference rule, and liftable(a(Ex-2), Ex-2) is proven by accessing, and activating, a number of inference rules that were learned earlier:

If something is graspable
& it's light
Then it's liftable

If something has a handle
Then it's graspable

Therefore, the add-REQ rule adds liftable(a(ex), ex) to the concept description. The Unification Task continues by creating REQ-boundaries for handle(a(ex), ex) and light(a(ex), ex), but no information can be conserved for these cases. The resulting, relatively correct concept description is:

λex[liftable(a(ex), ex) $\wedge$ bottom(a(ex), c(a(ex)), ex) $\wedge$ flat(c(a(ex)), ex) $\wedge$ concavity(a(ex), e(a(ex)), ex) $\wedge$ cylinder(b(a(ex)), ex) $\wedge$ small(b(a(ex)), ex) $\wedge$ body(a(ex), b(a(ex)), ex) $\wedge$ upwards(e(a(ex)), ex)]

LAIR requires one more example, "balanced-cup +," that results in the following description of the current example:

> Pos(cup, Ex-3) $\wedge$ balanced(a(Ex-3), Ex-3) $\wedge$ contents(a(Ex-3), d(a(Ex-3)), Ex-3) $\wedge$ handle(a(Ex-3), Ex-3) $\wedge$ light(a(Ex-3), Ex-3)
>
> "A positive example of a cup is something that is balanced, contains something, has a handle, and is light."

LAIR accepts this example, revises the concept description, and conserves information as for the "typical-cup +" example, resulting in the desired concept description:

> λex[liftable(a(ex), ex) $\wedge$ stable(a(ex), ex) $\wedge$ open-vessel(a(ex), ex) $\wedge$ body(a(ex), b(a(ex)), ex)]
>
> "Something that is liftable, stable, an open-vessel, and has a body."

There are a number of things to note about learning on the "balanced-cup +" example. First, the example differed from the concept description in a number of different ways, in contrast to the "typical-cup +" example that differed only in "how it was liftable." Therefore, LAIR was able to drop more irrelevant features and add more relevant features. Using information conservation, LAIR learns faster when positive examples show the typical variance in the concept, i.e., when "far-hits" are presented. Second, the predicate body(a(ex), b(a(ex)), ex) is not eliminated from the concept description. Although this predicate is not necessary for relative correctness, LAIR's goal is just to find a correct description. Demanding "minimally" correct descriptions is beyond its scope.

How does LAIR learn concept descriptions including negated predicates? Negated predicates in a concept description are essentially features of negative examples that are useful in ruling out those negative examples as instances of the concept. Negated predicates can be added by two tasks: the Accept Example Task, and the Differencing Task. As with positive examples, the Accept Example Task attempts to incorporate information from negative examples that do not violate constraints on the concept description. If a predicate in an example description is not true of some positive example, then that predicate is added to the concept description. A more difficult situation arises when the concept description incorrectly classifies a negative example as an instance of the concept. The following example is taken from learning of a rule for inferring "graspable." At this point, several examples have been presented, and the current concept description is:

> $\lambda x[\text{body}(a(ex), b(a(ex)), ex) \wedge \text{cylinder}(b(a(ex)), ex) \wedge \text{small}(b(a(ex)), ex)]$
>
> "Something with a small, cylindrical body."

The past positive example is "insulated-object +,"whose description is:

> Pos(graspable, Ex-4) $\wedge$ insulated(a(Ex-4), Ex-4) $\wedge$ body(a(Ex-4), b(a(Ex-4)), Ex-4) $\wedge$ small(b(a(Ex-4)), Ex-4) $\wedge$ cylinder(b(a(Ex-4)), Ex-4) $\wedge$ contents(a(Ex-4), d(a(Ex-4)), Ex-4) $\wedge$ hot(d(a(Ex-4)), Ex-4)
>
> "A positive example of 'graspable' is something insulated with a small, cylindrical body, and hot contents."

The current negative example is "uninsulated-object –," whose description is:

> $\neg$Pos(graspable, Ex-5) $\wedge$ body(a(Ex-5), b(a(Ex-5)), Ex-5) $\wedge$ small(b(a(Ex-5)), Ex-5) $\wedge$ cylinder(b(a(Ex-5)), Ex-5) $\wedge$ contents(a(Ex-5), d(a(Ex-5)), Ex-5) $\wedge$ hot(d(a(Ex-5)), Ex-5)
>
> "A negative example of 'graspable' is something with a small, cylindrical body, and hot contents."

The Classification Task finds that the concept description is too general to correctly classify "uninsul-obj –." The Differencing Task now attempts to specialize the concept description by creating an "add-REQ subtask" (with a REQ-boundary) and an "add-NOT subtask" (with a NOT-boundary). *Neg-rule-generalizations* are used by this task. A predicate *P* is a neg-rule-generalization of a predicate *Q* iff *Q* has a MOST-GEN-PRED that is a NEG-ANTECEDENT-OF a rule that has *P* as a CONSEQUENT. The relationship between *P* and *Q* is shown in figure 5.3.

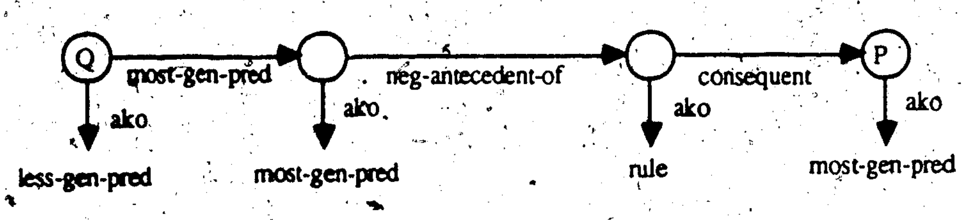

Figure 5.3. Negative Rule Generalizations

The add-REQ subtask initializes the REQ-boundary to all predicates true of the past positive example but not of the current negative example. Next, the add-REQ subtask chooses a REQ-boundary element Q and tries to add-REQ it to the concept description. If the attempt is successful, then the Differencing Task terminates. Otherwise, Q is deleted from the REQ-boundary. If Q satisfies

$Q(Past)\ \neg Q(Curr) \wedge \neg \mathrm{REQ}(Q, t)$

"Q is true of the *Past* example, false of the *Curr* example, and is not currently a REQ'd."

then its rule-generalizations are added to the REQ-boundary, and its neg-rule-generalizations are added to the NOT-boundary.

The add-NOT subtask initializes the NOT-boundary to all predicates true of *Curr* but not of *Past*. Next the add-NOT subtask chooses a NOT-boundary element Q and tries to add-NOT it to the concept description. If the attempt is successful, then the Differencing Task terminates. Otherwise, Q is deleted from the NOT-boundary. If Q satisîfies

$Q(Curr) \wedge \neg Q(Past) \wedge \neg \mathrm{NOT}(Q, t)$

"Q is true of the *Curr* example, false of the *Past* example, and is not currently a NOT."

then its rule-generalizations are added to the NOT-boundary, and its neg-rule-generalizations are added to the REQ-boundary.

The Differencing Task alternates between these two subtasks, since each adds new elements to the other's boundary. If both boundaries are empty, then the Differencing Task terminates unsuccessfully. Success by either the add-REQ subtask or the add-NOT subtask produces a relatively correct description.

For the "uninsulated-object –" example, the REQ-boundary is initialized to {insulated(a(ex), ex)}, and the NOT-boundary is initialized to { }. During earlier learning, insulated(a(ex), ex) was dropped, so it cannot be add-REQ'd. Since insulated(a(ex), ex) has no rule-generalizations, the REQ-boundary becomes empty. However, insulated(a(ex), ex) has a neg-rule-generalization, hot(b(a(ex)), ex), arising from the rule learned earlier:

```
If    something has contents
   & it has a body
   & the contents are hot
   & it's not insulated
Then its body is hot
```

The add-NOT subtask successfully adds ¬hot(b(a(ex)), ex) to the concept description, and the Differencing Task terminates with the following concept description:

$\lambda ex[\text{body}(a(ex), b(a(ex)), ex) \wedge \text{small}(b(a(ex)), ex) \wedge \text{cylinder}(b(a(ex)), ex) \wedge \neg\text{hot}(b(a(ex)), ex)]$

"Something with a small, cylindrical body that isn't hot."

This concept description is relatively correct, as required. LAIR learns the domain theory for the "cup" domain, consisting of 1 rule for "hot" and "cup," 2 rules each for "stable," "open-vessel," graspable," and "liftable." Learning requires 26 examples and takes about 570 seconds of CPU time.

5.3. Learning the Concept of an Arch

5.3.1 Overview

The "arch" concept was one of the first learned by machine learning systems (Winston, 1975), so provides a basis for comparison for LAIR. The system is given successive examples of arches and non-arches, and must induce a description of an arch. LAIR does constructive induction by treating Winston's generalization tree as prior knowledge of the system, and using this knowledge to infer features that are implicit in the example descriptions. The sequence of examples used by Winston's ARCH program, and LAIR, is shown below.

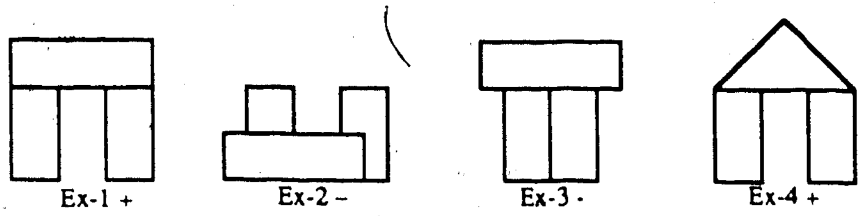

Figure 5.4 Arch Examples

In contrast to ARCH, which matched concept descriptions against examples, LAIR is told how objects and parts correspond. This information is given by using the same skolem function name for the corresponding part or object across examples.

5.3.2 Trace

LAIR starts with prior knowledge corresponding to Winston's generalization tree. For this trace, we will present example descriptions, prior knowledge, concept descriptions, and predicates in the logical syntax. The generalization tree is represented as the following implications:

∀ex∀y[Shape(y, Square, ex) ⇒ Shape(y, 4-sided, ex)]
∀ex∀y[Shape(y, Rectangle, ex) ⇒ Shape(y, 4-sided, ex)]
∀ex∀y[Shape(y, 4-sided, ex) ⇒ Shape(y, Polygon, ex)]
∀ex∀y[Shape(y, Wedge, ex) ⇒ Shape(y, Polygon, ex)]
∀ex∀y[Shape(y, Oval, ex) ⇒ Shape(y, Polygon, ex)]

The first example is:

Pos(arch, Ex-1) ∧ Shape(a(Ex-1), Square, Ex-1) ∧ Shape(b(Ex-1), Square, Ex-1) ∧ Shape(c(Ex-1), Square, Ex-1) ∧ OnTop(a(Ex-1), b(Ex-1), Ex-1) ∧ Ontop(a(Ex-1), b(Ex-1), Ex-1) ∧ Abuts(a(Ex-1), b(Ex-1), Ex-1) ∧ Abuts(a(Ex-1), c(Ex-1), Ex-1)

LAIR forms the initial concept description:

λx[Shape(a(x), Square, x) ∧ Shape(b(x), Square, x) ∧ Shape(c(x), Square, x) ∧ OnTop(a(x), b(x), x) ∧ Ontop(a(x), b(x), x) ∧ Abuts(a(x), b(x), x) ∧ Abuts(a(x), c(x), x)

The second example is a negative example of an arch:

¬Pos(arch, Ex-2) ∧ Shape(a(Ex-2), Square, Ex-2) ∧ Shape(b(Ex-2), Square, Ex-2) ∧ Shape(c(Ex-2), Square, Ex-2)

This example is correctly classified as a non-instance of the arch concept because several REQ's in the concept description, OnTop(a(x), b(x), x), Ontop(a(x), b(x), x), Abuts(a(x), b(x), x), and Abuts(a(x), c(x), x), are not satisfied by the current example. The third example is a negative example of an arch because blocks a(x) and b(x) are touching:

¬Pos(arch, Ex-3) ∧ Shape(a(Ex-3), Square, Ex-3) ∧ Shape(b(Ex-3), Square, Ex-3) ∧ Shape(c(Ex-3), Square, Ex-3) ∧ OnTop(a(Ex-3), b(Ex-3), Ex-3) ∧ Ontop(a(Ex-3), b(Ex-3), Ex-3) ∧ Abuts(a(Ex-3), b(Ex-3), Ex-3) ∧ Abuts(b(Ex-3), c(Ex-3), Ex-3) ∧ Abuts(a(Ex-3), c(Ex-3), Ex-3)

LAIR modifies its description to include ¬Abuts(b(x), c(x), x) during the Accept Example Task because this does not violate any constraints on the concept description. The revised concept description is:

λx[Shape(a(x), Square, x) ∧ Shape(b(x), Square, x) ∧ Shape(c(x), Square, x) ∧ OnTop(a(x), b(x), x) ∧ Ontop(a(x), b(x), x) ∧ Abuts(a(x), b(x), x) ∧ Abuts(a(x), c(x), x) ∧ ¬Abuts(b(x), c(x), x)]

This modified description correctly classifies the current example because Abuts(b(Ex-3), c(Ex-3), Ex-3) can be proven. The fourth example is not correctly classified because Square(a(Ex-4), Ex-4) cannot be proven.

Pos(arch, Ex-4) ∧ Shape(a(Ex-4), Wedge, Ex-4) ∧ Shape(b(Ex-4), Square, Ex-4) ∧ Shape(c(Ex-4), Square, Ex-4) ∧ OnTop(a(Ex-4), b, Ex-4) ∧ Ontop(a(Ex-4), b(Ex-4), Ex-4) ∧ Abuts(a(Ex-4), b(Ex-4), Ex-4) ∧ Abuts(a(Ex-4), c(Ex-4), Ex-4)

LAIR drops Shape(a(x), Square, x) from the concept description, and remembers this as a constraint on future concept descriptions. LAIR searches for information to conserve, looking at Shape(a(x), 4-sided, x), which is Drop'd, and Shape(a(x), Polygon, x), which is add-REQ'd. The final concept description is:

λx[Shape(a(x), Polygon, x) ∧ Shape(b(x), Square, x) ∧ Shape(c(x), Square, x) ∧ OnTop(a(x), b(x), x) ∧ Ontop(a(x), b(x), x) ∧ Abuts(a(x), b(x), x) ∧ Abuts(a(x), c(x)(x), x) ∧ ¬Abuts(b(x), c(x), x)]

The user can request that this be converted into a rule for recognizing instances of an arch, by specifying a name of a predicate for the concept, and arguments of the predicate. For example, if the user specifies that the name of the predicate is to be Arch, and the arguments are to be variables corresponding to skolem functions a, b, and c, the following implication would be added to the system's knowledge:

> ∀x[illegible]ape(y, Polygon, x) ∧ Shape(z, Square, x) ∧ Shape(w, Square, x)
> [illegible]Top(y, z, x) ∧ Ontop(y, z, x) ∧ Abuts(y, z, x) ∧ Abuts(y, w, x)
> ∧ ¬Abuts(z, w, x)] ⇒ Arch(y, z, w, x)]]

Note that the skolem functions a, b, c are changed to variables y, z, w. The frame memory is modified to include the frames and slots necessary to represent this rule and to support its use in future learning, so that Arch(y, z, w, x) can be instantiated to be a feature of future examples.

5.3.3 Discussion

LAIR is able to correctly learn the description of an arch from the examples presented to ARCH. In comparing LAIR to ARCH, it is important to realize that LAIR cannot match up descriptions for itself, which was the heart of Winston's program and one reason why ARCH could deal only with near-misses. However, there are several other important differences between LAIR and ARCH:

1. ARCH requires near-misses even if the matching is done for ARCH. In contrast, LAIR does not require near-misses. ARCH requires near-misses because, in addition to the matching problem, ARCH will not know which feature to make a NOT if there are more than one. If ARCH makes an error in choosing which feature to make a NOT, then ARCH will either not learn correctly or will have to backtrack to the point at which it chose the feature, and choose some other feature. LAIR is able to undo its NOTs with the drop-NOT rule, so LAIR can revise just that one error, rather than

revising all of the learning that occurred between the time the error was made and the time the error was discovered.

2. LAIR represents generalization hierarchies as implications, which can also represent non-hierarchical knowledge. ARCH's knowledge representation is restricted to hierarchies or total orders.

5.4. Extending LAIR to Learn Analogies

5.4.1 Overview

This example shows how LAIR can learn by analogy. We can view analogy as transfer of knowledge from a "base domain," such as the solar system, to a "target domain," such as the Rutherford model of the atom. For example, we say "the atom is like the solar system." This analogy problem has been studied by Gentner (1983) and Burstein (1986).

5.4.2. Trace

LAIR starts with knowledge about the objects in the base domain and target domain, represented as the following implications. This knowledge is previously learned knowledge that has been taught to LAIR at some earlier time, and plays an important role in learning the analogy. LAIR's prior knowledge is shown below:

Knowledge about the Sun:

$\forall ex \forall y[\mathrm{Sun}(y, ex) \Rightarrow \mathrm{Yellow}(y, ex)]$
$\forall ex \forall y[\mathrm{Sun}(y, ex) \Rightarrow \mathrm{Hot}(y, ex)]$
$\forall ex \forall y[\mathrm{Sun}(y, ex) \Rightarrow \mathrm{Massive}(y, ex)]$
$\forall ex \forall y[\mathrm{Sun}(y, ex) \Rightarrow \mathrm{Phys\text{-}Obj}(y, ex)]$

Knowledge about a Planet:

$\forall ex \forall y[\mathrm{Planet}(y, ex) \Rightarrow \mathrm{Phys\text{-}Obj}(y, ex)]$

Knowledge about the Nucleus:

$\forall ex \forall y[\mathrm{Nucleus}(y, ex) \Rightarrow \mathrm{Charge}(y, \mathrm{Pos}, ex)]$
$\forall ex \forall y[\mathrm{Nucleus}(y, ex) \Rightarrow \mathrm{Phys\text{-}Obj}(y, ex)]$

Knowledge about the Electron:

$\forall ex \forall y[\text{Electron}(y, ex) \Rightarrow \text{Charge}(y, \text{Neg}, ex)]$

$\forall ex \forall y[\text{Electron}(y, ex) \Rightarrow \text{Phys-Obj}(y, ex)]$

Knowledge about Physics:

$\forall ex \forall x \forall y[\text{Phys-Obj}(y, ex) \wedge \text{Phys-Obj}(y, ex) \wedge \text{Massive}(x, ex) \Rightarrow \text{Grav-Attract}(x, y, ex)]$

$\forall ex \forall x \forall y[\text{Charge}(x, \text{Pos}, ex) \wedge \text{Charge}(y, \text{Neg}, ex) \Rightarrow \text{Elect-Attract}(x, y, ex)]$

$\forall ex \forall x \forall y[\text{Elect-Attract}(x, y, ex) \Rightarrow \text{Attracts}(x, y, ex)]$

$\forall ex \forall x \forall y[\text{Grav-Attract}(x, y, ex) \Rightarrow \text{Attracts}(x, y, ex)]$

$\forall ex \forall x \forall y[\text{Attracts}(x, y, ex) \Rightarrow \text{More-Massive}(x, y, ex)]$

$\forall ex \forall x \forall y[\text{Attracts}(x, y, ex) \Rightarrow \text{Revolves-Around}(x, y, ex)]$

To "solve" the analogy between the solar system and the atom, we give LAIR a description of the solar system and a description of the atom. Its task is to find a concept they are an instance of. It will find commonalities that could be argued to be the same as the mapping problem. So, first LAIR is presented with the description of the solar system:

$\text{Pos}(\text{Ex-1}) \wedge \text{Sun}(A(\text{Ex-1}), \text{Ex-1}) \wedge \text{Planet}(B(\text{Ex-1}), \text{Ex-1})$

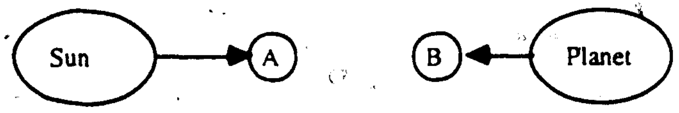

Figure 5.5. The Solar System

LAIR forms the following analogical concept:

$\lambda x[\text{Sun}(A(x), x) \wedge \text{Planet}(B(x), x)]$

Next, LAIR is presented with the description of the atom:

$\text{Pos}(\text{Ex-2}) \wedge \text{Nucleus}(A(\text{Ex-2}), \text{Ex-2}) \wedge \text{Electron}(B(\text{Ex-2}), \text{Ex-2})$

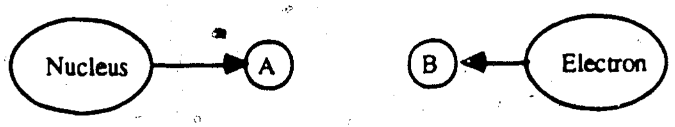

Figure 5.6. The Atom

LAIR first modifies the analogy description to include the predicates Nucleus(A(x), x) and Electron(B(x), x), since these predicates have not been found to be false of the target or base domains. The modified description,

λx[Sun(A(x), x) $\wedge$ Planet(B(x), x) $\wedge$ Nucleus(A(x), x) $\wedge$ Electron(B(x), x)]

is now tested against the target and the base domains. None of the predicates are implicitly true of both, so *all* of them are dropped from the analogy description.

LAIR conserves the information lost in dropping Sun(A(x), x), Planet(B(x), x), Planet(B(x), x), Electron(B(x), x), and Nucleus(A(x), x) by adding Attracts(A(x), B(x), x), Phys-Obj(B(x), x), More-Massive-Than(A(x), B(x), x), and Phys-Obj(A(x), x), respectively. LAIR's specialized analogy description is:

λx[More-Massive-Than(A(x), B(x), x) $\wedge$ Phys-Obj(A(x), x) $\wedge$ Phys-Obj(B(x), x)
$\wedge$ Attracts(A(x), B(x), x)]

However, although this description is relatively correct, and conserves information for each of the predicates that were dropped from the previous analogy description, it is lacking a predicate that we would like. LAIR is told that this is not a correct description, and goes on to add-REQ Revolves-Around(A(x), B(x), x). This produces the desired analogy description:

λx[More-Massive-Than(A(x), B(x), x) $\wedge$ Phys-Obj(A(x), x) $\wedge$ Phys-Obj(B(x), x)
$\wedge$ Attracts(A(x), B(x), x) $\wedge$ Revolves-Around(A(x), B(x), x)]

The Unification Task involves looking at many features that could not be added to the analogy description. LAIR represents this knowledge explicitly. To clarify LAIR's knowledge about the base and target domains, as opposed to the analogy, their descriptions are shown below in diagrammatic form. Note that these descriptions are much more specific than the original descriptions given to LAIR (figures 5.5 and 5.6).

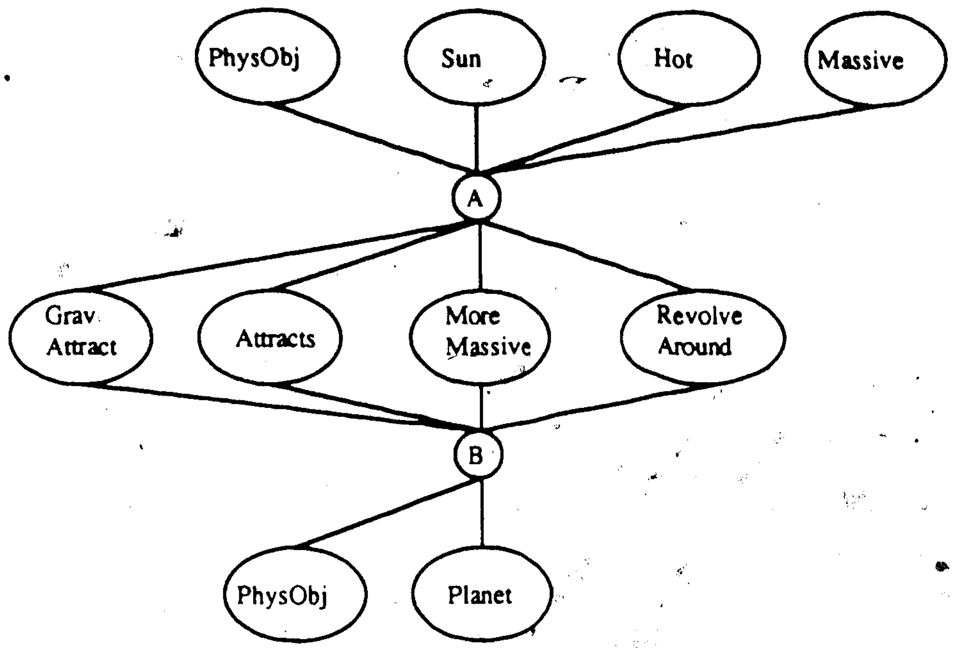

Figure 5.7. Knowledge about the Solar System

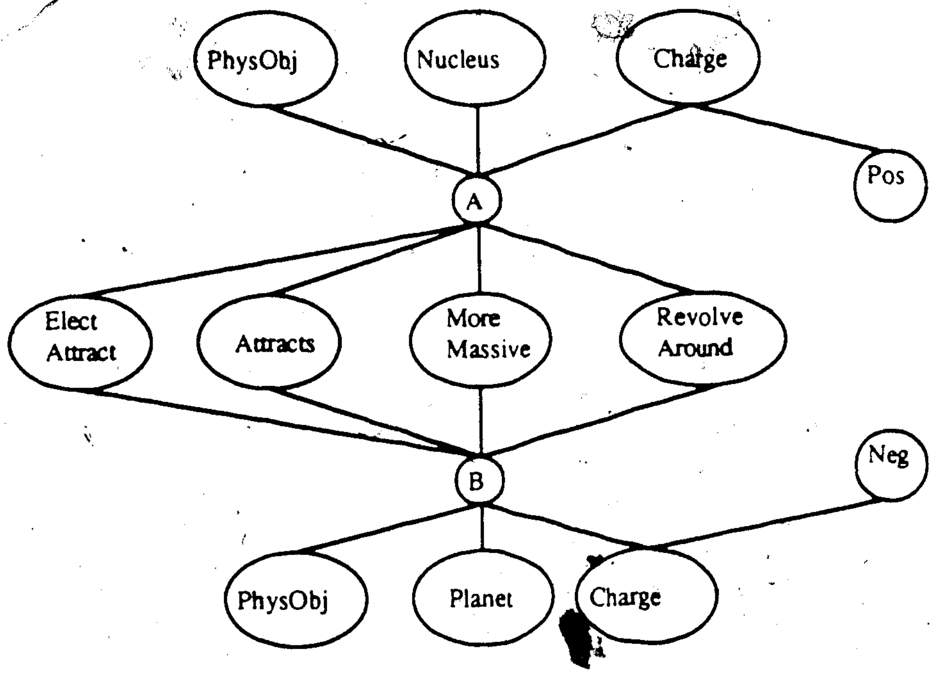

Figure 5.8. Knowledge about the Atom

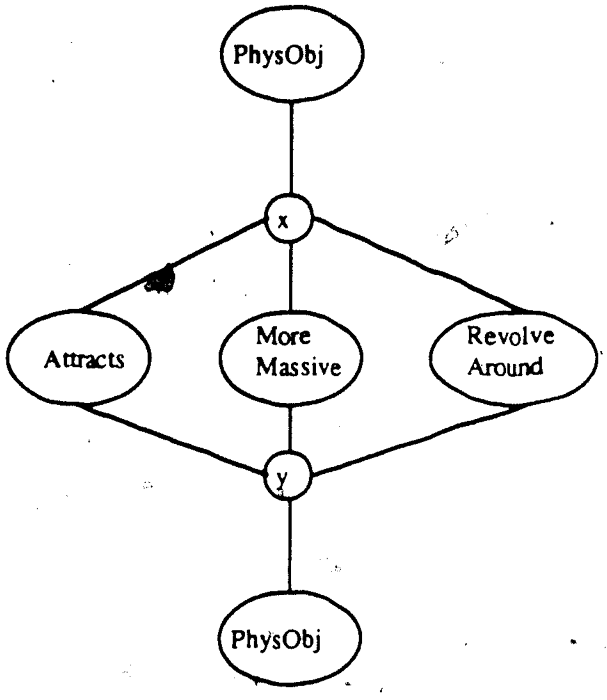

Figure 5.9. The Analogical Concept

5.4.3. Discussion

LAIR is able to correctly learn the analogy between the solar system and the atom by presenting the descriptions of them as positive examples of the analogy. Its performance brings up a very important point about this learning approach—deduction *stops* as soon as it is done conserving information and has an analogy description that covers the base and target domains. We will discuss this point, and compare the learning approach to Gentner's structure mapping theory.

Gentner's central claims are:

> Analogy is characterized by the mapping of relations between objects, rather than attributes of objects, from base to target; and, further that the particular relations mapped are those that are dominated by higher-order relations that belong to the mapping (the *systematicity* claim). These rules have the desirable property that they depend only on syntactic properties of the knowledge representation, and not on the specific content of the domain. Further, this theoretical framework allows us to state the differences between analogies and literal similarity statements, abstractions and other kinds of comparisons (Gentner, 1983).

It is worthwhile to see how Gentner's analysis would apply to our representation of the central force system analogy. Although there are no higher-order relations explicitly represented, if there were to be higher-order relations, or systems, clearly a gravitational system and a electrical attraction system would be reasonable candidates. The definition of attribute and relation must be modified since predicates that correspond to attributes in Gentner's theory have two arguments in our system, because of the need to represent example arguments. Therefore, we define attributes to be binary predicates, and relations to be ternary predicates. Gentner's Systematicity Principle correctly predicts that the attribute Phys-Obj(x, y) should be mapped over, since it participates in a system of relations: the gravitational attraction system. Our system also correctly predicts that the attribute Phys-Obj(x, y) should be mapped over, because it is provable of all the positive examples of a central force system. Gentner's syntactic theory and Systematicity Principle predict that the relationGrav-Attract(x, y, z) should be mapped over, as it is both a relation and one that is part of an interconnecting set of relations describing a system, namely a gravitational system. However, Grav-Attract(x, y, z) is not applicable in the target domain; our system in fact does not include this predicate in the central force system description. Thus, in this limited domain and for this limited example, our system finds a more accurate set of mappings.

In contrast to Gentner's approach that is based on syntax, and avoids domain-dependent information, our system relies on domain-dependent information to do the mapping. However, each of the pieces of prior knowledge seem reasonable for someone capable of understanding the analogy. Thus, the inclusion of this knowledge seems reasonable for a system that tries to understand analogies [illegible] human world. On the other hand, for systems that simply wish to take advantage of analogical mapping techniques without considerations of psychological plausibility, this is a disadvantage since representing domain knowledge is usually a labor-intensive and tedious task.

Gentner relies on predicates that ar[illegible]s. However, we cannot know whether a predicate will be represented as an object feature (e.g. massive(obj)) or as a relationship between an object and a value (e.g. [massiveness(obj) = 200]). In contrast, LAIR relies on the meaning of a predicate: what examples it is true of, and its relationship to other predicates via implications.

There are a number of disadvantages of our approach to analogical mapping. The first disadvantage is that it has no sense of what correspondences should be made and when all the necessary mappings are done. The fact that we need to prompt it for "Revolves-Around" is important. As a learning system, its fundamental goal is to learn a correct concept description from both positive and negative examples. Analogical mapping problems deal with learning from positive-only examples, so different constraints on the adequacy of a "concept description" or "analogical mapping" need to be formulated. Without a sound criteria for "when is a mapping complete?" a system cannot be expected to always find a complete mapping. Definition of such criteria for analogical mapping remains an open research problem. A disadvantage of LAIR, but not of our approach, is that "solar system" is defined as a "something that has a sun and a planet." LAIR could be extended to handle this definition if the skolem functions A(x) and B(x) were introduced using a rule, e.g.

$$\forall ex,x[\text{solar-system}(x, ex) \Rightarrow [\text{sun}(a(x), ex) \wedge \text{planet}(b(x), ex)]]$$

eliminating the need for the information about a sun and a planet to be explicit in the example. However, then the teacher could not tell LAIR the correspondences between skolem functions, because this information is given by presenting examples with the same skolem function name for corresponding parts. Since the system would generate the skolem function names, the teacher would not be able to ensure that the same names are generated. In principle, LAIR could be modified to accept this information explicitly rather than implicitly, so that it would be able to handle this case, but this is beyond the scope of the current implementation.

Another aspect of our system that needs clarification is that mapping is defined not as the process of *trying* to prove a feature in a base domain true of the target domain, but as a *successful* attempt to prove a feature of a base domain true of the target domain. A disadvantage of this approach is that analogies are frequently used to convey new information about the target. For example, a common analogy used to teach novice programmers is "variables are like boxes" (Burstein, 1986). Here, some transfer of features is intended, but not others. The system must infer what feature *types* are valid ones to map. If a learner lacks sufficient prior knowledge to successfully prove a feature of the base domain true of the target domain, LAIR would predict no transfer would occur. In practice, transfer does occur, something that cannot be accounted for by LAIR.

LAIR does not account for the fact that a learner may not even *try* to prove many features of the base domain true of the target domain. This may relate more to the way inference is done in LAIR rather than LAIR's applicability to analogical mapping problems. First, LAIR does deductive inference. Humans are notoriously bad at deduction! Deduction is one reasoning approach that we use to build intelligent systems, but may not have any corresponding process in the human machinery. Second, LAIR does deduction without regard to interrelationships between features other than those explicitly represented as implications. In practice, there may be relationships between features such as "if feature F_1 is not true of x, then feature F_2 cannot possibly be true of x; so don't even try to prove F_2 of x". If our learning system used such a deductive apparatus, then it *might* make correct predictions for such cases. The important point is inference is important in analogical mapping, and that systems that do not accurately model human inference may have difficulty in accurately modelling analogical mapping in humans.

In summary, our model of understanding analogies has several advantage over Gentner's theory. First, it makes a more accurate prediction of what is mapped for this formulation of the central force system problem. Second, it does not rely heavily on whether knowledge is represented as a relation or as a feature. Our model has several disadvantages. First, our model has little sense of what correspondences should be made and when the "mappings" are done. Second, our model cannot account for how information is conveyed from a base domain to a target domain in which the learner has little or no prior knowledge. Last, our model does not model human inference accurately, so it does not model intermediate steps of analogical mapping accurately. Nevertheless, there appear to be relationships between concept learning and analogical mapping that deserve further attention, and areas of common interest.

5.5. Summary

LAIR is able to learn concepts in a number of different domains. The traces showed how constraints guide constructive induction, how prior knowledge can influence or enable learning, and how information conservation can be a useful heuristic for learning concept descriptions. LAIR can be extended to learn analogical concepts, but further work is required if LAIR is to model human analogical learning. LAIR has some advantages over ARCH, but does not address the matching problem, for which ARCH's solution is a substantial contribution to machine learning.

Chapter 6

Conclusions and Implications

6.1 Introduction

This section summarizes conclusions about how to guide constructive induction, discusses relationships of this work to analogical mapping and explanation-based learning, and proposes some directions for future research.

6.2 Conclusions about Guiding Constructive Induction

LAIR demonstrates how constructive induction can be controlled by (1) reducing it to simpler operations, (2) constraining the simpler operations to preserve relative correctness, (3) doing deductive inference on an as-needed basis to meet specific information requirements of learning sub-tasks, and (4) constraining the search space by subtask-dependent constraints. In addition, LAIR's rules are relatively complete, in the sense that a correct, usable concept description can be derived, if one exists, without backtracking or maintaining multiple concept descriptions.

Reducing constructive induction to the operations of deductive inference, add-REQ, add-NOT, drop-REQ, and drop-NOT has several advantages. First, the reduced operations are simpler, and can be more tightly constrained. Second, the reduced operations are relatively complete and preserve relative correctness. This allows LAIR to learn a correct concept description without backtracking or maintaining multiple concept descriptions even if far-misses are presented or examples are presented in an arbitrary order.

Doing deductive inference on an as-needed basis is important for several reasons. First, deductive inference is expensive. By doing deduction on an as-needed basis, LAIR avoids having to infer many propositions that may be irrelevant to the task at hand. LAIR

Constructive induction can also be controlled by subtask-dependent constraints. For example, the REQ-boundary and the NOT-boundary are potential candidates for constructive induction, so smaller boundaries reduces the amount of constructive induction to be done. LAIR uses logical constraints on the expansion of these boundaries that are dependent on the current subtask. For example, the Differencing Task not only constrains candidates for constructive induction to be plausible new features of a concept description, but also to be plausible new features that correctly rule out the current negative example.

6.2. Relationship to Analogical Mapping

LAIR's performance on the "central force system" concept demonstrates that concept learning systems can solve some analogical mapping problems. However, the goal of LAIR, to form a relatively correct concept description, is too general for analogical mapping. Since analogical mapping also addresses the fundamentally different problem of conveying information from one domain to another, additional inference methods need to be defined if LAIR is to solve analogical mapping problems satisfactorily .

6.4. Relation to Explanation-Based Learning

An important and recent method of learning is explanation-based learning (Mitchell, Keller, & Kedar-Cabelli, 1986). Explanation-based learning uses a domain theory (what I refer to as prior knowledge) from which it is possible to explain *why* an example is an instance of a concept. This section will discuss the Explanation Based Generalizer (EBG) of Mitchell, Keller, and Kedar-Cabelli.

Mitchell et al. define the explanation-based concept learning problem with different terminology from that used in this work. I will present Mitchell et al.'s terminology, noting correspondences to our terminology where appropriate. A *concept* is defined as a predicate over some universe of instances, and each *instance* in this universe is described by a collection of ground literals that represent its features and their values. A concept is therefore equivalent to a concept in our system, and an instance is equivalent to an example in our system. A *concept definition* describes the necessary and sufficient conditions for being an example of the concept, while a *sufficient concept definition* describes sufficient conditions for being an example of the concept. A concept definition is therefore equivalent to a correct concept description in our system, and a sufficient concept definition would be a concept description that described no negative examples (although it might not describe all positive examples) in our system. A *generalization* of an example is a concept definition which describes a set containing that example, and corresponds to a concept description that describes that example in our system. An *explanation* of how an instance is an example of a concept is a proof that the example satisfies the concept definition. An *explanation structure* is the proof tree, modified by replacing each instantiated rule by the associated general rule.

The explanation-based generalization problem can be stated as follows:

Given:

- _Goal Concept:_ A concept definition describing the concept to be learned. (It is assumed that this concept definition fails to satisfy the Operationality Criterion.)
- _Training Example:_ An example of the goal concept.
- _Domain Theory:_ A set of rules and facts to be used in explaining how the training example is an example of the goal concept.
- _Operationality Criterion:_ A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed.

Determine:

- A generalization of the training example that is a sufficient concept definiton for the goal concept and that satisfies the operationality criterion.

Mitchell et al. make the important point that without the notion of operationality, the input goal concept definition could always be a correct output concept definition and there would be nothing to learn! Mitchell et al. state that

> The operationality criterion imposes a requirement that learned concept definitions must be not only correct, but also be *in a usable form* before learning is complete.

However, in principle, an operationality criterion might be used to impose other requirements on a goal concept definition. For example, one might require that the goal concept definition be understandable to the user, or that the goal concept definition be of a form that is particularly efficient for the system to use. The operationality requirement also presupposes that the system that is to use the output correct concept definition lacks the deductive apparatus of the EBG system: otherwise it would be able to use the the goal concept definition and the deductive apparatus to correctly classify all of the examples.

The EBG method is defined as follows:

1. _Explain:_ Construct an explanation in terms of the *domain theory* that proves how the *training example* satisfies the *goal concept* definition.
   - This explanation must be constructed so that each branch of the explanation structure terminates in an expression that satisfies the operationality criterion.

2. _Generalize:_ Determine a set of sufficient conditions under which the explanation structure holds, stated in terms that satisfy the *operationality criterion*.
   - This is accomplished by regressing the goal concept through the explanation structure. The conjunction of the resulting regressed expressions constitutes the desired concept definition.

An important point about explanation-based systems is that the goal concept definition must be input to the system. Similarity-based methods using constructive induction are capable of learning goal concept definitions. For example, the description of "cup" learned by LAIR is the goal concept of Winston's explanation-based ANALOGY program (Winston, Binford, Katz & Lowry, 1983). In an explanation-based system, the domain knowledge must be strong enough to deduce that the goal concept definition is true of the example. Therefore, each conjunct in the goal concept definition is an implicit feature of the example that can be added to a concept description by means of constructive induction. Mitchell et al. suggest:

> Further research is needed to extend the EBG method to generalization tasks in which the domain théory is not sufficient to deductively infer the desired concept ... Thus, a major research issue for explanation-based generalization is to develop methods that utilize imperfect domain theories to guide generalization, as well as methods for improving imperfect theories as learning proceeds ... While EBG infers concept definitions deductively from a single example, similarity-based methods infer concept definitions inductively from a

number of training examples. It seems clearly desirable to develop combined methods that would utilize both a domain theory and multiple training examples to infer concept definitions. This kind of combined approach to generalization will probably be essential in domains where only imperfect theories are available.

LAIR provides a partial solution to this problem, since learned concept can become prior knowledge (domain theory in Mitchell et al.'s terminology) for future learning tasks. For example, the description of "graspable" learned by LAIR in the "cup" domain is transformed into a rule, and is added to the domain theory.

Mitchell et al. state "explanation-based methods such as EBG overcome the main limitation of similarity-based methods: their inability to produce justified generalizations." However, the inability of similarity-based methods to produce justified generalizations is not inherent in the method: LAIR's generalizations are clearly justified since they only drop features that cannot occur in a correct concept description. LAIR, like EBG systems, can also justify their classification of examples by outputting a trace of the proof that a concept description is true of an example.

Another important difference between explanation-based generalization and LAIR is the centrality of examples to the learning process. Mitchell et al. note that "one wonders why training examples are required at all [in EBG systems] ... In principle, they are not ... Training examples provide a means of focusing the learner on formulating only concept descriptions that are relevant to the environment in which it operates." On the other hand, examples in LAIR convey important information. This difference is partially due to the fact that LAIR is acquiring new knowledge, i.e. a definition of the concept, whereas according to Mitchell et al., the explanation-based generalizer is "not acquiring truly 'new' knowledge, but only enabling the learner to reformulate, operationalize, or deduce what the learner already knows implicitly." However, as Mitchell et al. mention, "this statement is somewhat misleading ... this kind of learning is non-trivial."

Another way to view the difference between the two systems is as a competence/performance distinction. LAIR is able to learn knowledge that allows it to be more competent, i.e. to recognize instances of concepts that it has not previously known, whereas the EBG system is learning performance knowledge, i.e. how to efficiently recognize instances of a concept it already knows. Performance is important, and may even affect competence if a system has limited resources:

The explanation-based and similarity-based systems using constructive induction perform complementary functions. The similarity-based, constructive induction systems are useful for accretion of knowledge. The explanation-based systems are useful for tuning and restructuring this knowledge into more useful forms (Rumelhart & Norman, 1978; Anderson, 1986). The complementary nature of these forms of learning suggest that combined systems exploiting the best features of each method might be an interesting area for future research.

6.5. Directions for Future Research

There are many ways in which LAIR could usefully be extended. This section will discuss the following areas: deduction, alternate concept description forms, multiple concept learning in the same domain, and combination with explanation-based learning.

One area of future research is improvement and extension of LAIR's deduction strategy. LAIR could be extended to do hierarchical deduction: doing higher level, skeletal proofs before the full proof. LAIR's inference could also be extended to statistical and probabilistic inference. LAIR assumes that no noise is present in the examples. In systems that extract their own positive and negative examples from observation, noise often occurs. Statistical inference would be useful in dealing with these kinds of problems.

A second area of future research is extension of LAIR to learn disjunctive concept descriptions. LAIR assumes that it is learning a conjunctive concept. One way in which LAIR could be modified to learn disjunctive concepts is to have LAIR infer plausible groupings of positive examples into sets that could each be described by a different disjunct of a concept description. Many conjunctive concept learning systems have been extended to learn disjuncts by partitioning examples according to the time they were presented. Logically, however, there is no reason to assume that this is a good partitioning. Partitioning examples by using knowledge about the example descriptions might be a better way to learn disjunctive concept descriptions.

A third area of future research is multiple concept learning in the same domain. LAIR is currently able to learn concepts in a domain, and to use learned concepts as prior knowledge for future learning tasks. LAIR could be extended to use knowledge of how it learned previous concepts on future concept learning tasks by becoming aware of what is "relevant" or "important" in a domain.

A fourth area of future research is combination of the similarity-based, constructive induction learning method of our [illegible] with explanation-based methods. As mentioned, these methods seem complementary. Future research could be done on how these systems could be combined into an integrated, effective system.

Bibliography

Anderson, J. R. (1983). Cognitive and psychological computation with neural models. *IEEE Transactions on Systems, Man, and Cybernetics*, **13**, 799-815.

Anderson, J. R. (1986). Knowledge compilation: the general learning mechanism. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol II. Tioga: Palo Alto.

Angluin, D. (1978). Inductive inference of formal languages from positive data, *Information and Control*, **45**, 117-135.

Barr, A. & Feigenbaum, E. A. (Eds.) (1982). *The Handbook of Artificial Intelligence*, Vol. III, Morgan Kaufman: Los Altos.

Berwick, R. C. (1983). Learning from positive-only examples: the subset principle and three case studies. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol I. Tioga: Palo Alto.

Bobrow, D. G. & Winograd, T. (1977). An overview of KRL, a knowledge representation language. *Cognitive Science*, **1**, 3-46.

Bundy, A., Silver, B., & Plummer, D. (1985). An analytical comparison of some rule-learning programs. *Artificial Intelligence*, **27**, 137-181.

Burstein, M. H. (1986). Concept formation by incremental analogical reasoning and debugging. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol II. Tioga: Palo Alto.

Buchanan, B. G. & Feigenbaum, E. A. (1978). DENDRAL and META-DENDRAL: their applications dimension, *Artificial Intelligence*, **11**, 5-24.

Carbonell, J. G. (1983). Learning by analogy: formulating and generalizing plans from past experience. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol I. Tioga: Palo Alto.

Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (1983). An overview of machine learning. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol I. Tioga: Palo Alto.

Carbonell, J. G. (1986). Analogy in problem solving. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol II. Tioga: Palo Alto.

Davis, R. & King, J. (1977). An overview of production systems. *Machine Intelligence*, **8**, Elcock & Michie (Eds.), Wiley: New York.

Dietterich, T. G. & Michalski, R. S. (1983). A comparative review of selected methods for learning structural descriptions. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol I. Tioga: Palo Alto.

Feldman, J. A. (1981). A connectionist model of visual memory. In G. E. Hinton & J. R. Anderson (Eds.), Parallel Models of Associative memory (pp 44-81). Erlbaum; Hillsdale, NJ.

Forgy, C. L. (1979). The OPS4 Reference Manual. Technical Report, Department of Computer Science, Carnegie-Mellon University.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, **19**, 17-37.

Gentner, D. (1983). Structure mapping: A theoretical framework for analogy, *Cognitive Science*, 7, 155-70.

Hayes-Roth, F., & McDermott, J. (1976). Knowledge acquisition from structural descriptions. Department of Computer Science Working Paper, Carnegie-Mellon University, Pittsburgh.

Hayes-Roth, F. (1977). The role of partial and best matches in knowledge systems. Department of Computer Science Working Paper, Carnegie-Mellon University, Pittsburgh.

Hayes-Roth, F. & McDermott, J. (1978). An interference matching technique for inducing abstractions, *Communications of the ACM*, **21**, 401-411.

Hinton, G. E., Sejnowski, T. J., & Ackley, D. H. (1984). *Boltzmann machines: Constraint satisfaction networks that learn* (Tech. Rep. No. CMU-CS-84-119). Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Langley, P. (1986) A general theory of discrimination learning. To appear as a chapter in D. Klahr, P. Langley, & R. Neches (Eds.), *Production System Models of Learning and Development*. Cambridge, Mass.: MIT Press.

Lenat, D. B. (1977). Automated theory formation in mathematics, *Fifth International Joint Conference on Artificial Intelligence*, 833-844.

Lenat, D. B. (1982). The nature of heuristics, *Artificial Intelligence*, **19**, 189-249.

Lenat, D. B. (1983a). Theory formation by heuristic search: the nature of heuristics II: background and examples, *Artificial Intelligence*, **21**, 31-59.

Lenat, D. B. (1983b). EURISKO: a program that learns new heuristics and domain concepts: the nature of heuristics II: program design and results, *Artificial Intelligence*, **21**, 61-98.

Minsky, M. & Papert, S., (1969). *Perceptrons*, MIT Press: Cambridge, Mass.

Minsky, M. (1975). A framework for representing knowledge. In P. Winston (Ed.), *The Psychology of Computer Vision*. McGraw Hill: New York.

Michalski, R. S. (1973). AQVAL/1 - computer implementation of a variable valued logic system VL1 and examples of its application to pattern recognition, *Proceedings of the First international Joint Conference on Pattern Recognition*, 3-17.

Michalski, R. S. & Larson, J. B. (1978). Selection of most representative training examples and incremental generation of VL1 hypotheses: the underlying methodology and description of programs ESEL and AQ11, Report 867, University of Illinois.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol I. Tioga: Palo Alto.

Michalski, R. S., Amarel, S., Lenat, D. B., Michie, D., & Winston, P. H. (1986). Machine learning: challenges of the eighties. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol II. Tioga: Palo Alto.

Michalski, R. S., & Stepp, R. (1983). Learning from observation: conceptual clustering. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol I. Tioga: Palo Alto.

Michalski, R. S. (1986). Understanding the nature of learning: issues and research directions. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol II. Tioga: Palo Alto.

Michalski, R. S. & Winston, P. H. (1986). Variable precision logic, *Artificial Intelligence*, **29**, 121-146.

Mitchell, T. M. (1977). Version spaces: a candidate elimination approach to rule learning, *Fifth International Joint Conference on Artificial Intelligence*, 305-310.

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

Mitchell, T. M., Utgoff, P. E., & Banerji, R. B. (1983). Learning by experimentation: acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol I. Tioga: Palo Alto.

Mitchell, T. M., Keller, R. & Kedar-Cavelli, S. (1986). Explanation-based generalization: a unifying view, *Machine Learning* **1**, No. 1, January.

Morgan, C. G. (1972). *Inductive Resolution*, M. Sc. Thesis, University of Alberta.

Mostow, D. J. (1983). Machine transformation of advice into a heuristic search procedure. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol I. Tioga: Palo Alto.

Nilsson, N. J., (1965). *Learning Machines*, McGraw-Hill: New York.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*, Morgan Kaufman: Los Altos, California.

Reiter, R. (1978). On reasoning by default. *TINLAP* 2, 1978.

Reiter, R. (1980). A logic for default reasoning, *Artificial Intelligence*, **13**.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organiation in the Brain, *Psychological Review*, **65**, 386-407.

Rumelhart, D. E., & Norman, D. A. (1978). Accretion, tuning, and restructuring: three modes of learning. In J. W. Cotton & R. Klatzky, (Eds.), *Semantic Factors in Cognition*. Erlbaum Associates: Hillsdale, N. J.

Sammut, C., & Banerji, R. B. (1986). Learning concepts by asking questions. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol II. Tioga: Palo Alto.

Schubert, L. K. (1976). Extending the expressive power of semantic networks, *Artificial Intelligence*, 7, 163-198.

Simon, H. A. (1983). Why should machines learn? In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol I. Tioga: Palo Alto.

Stepp, R. E., & Michalski, R. S. (1986). In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol II. Tioga: Palo Alto.

Thomason, R. H. (1970). *Symbolic Logic: An Introduction*, Collier-Macmillan Limited: London.

Uhr, L. & Vossler, C. (1963). A pattern-recognition program that generates, evaluates and adjusts its own operators. In *Computers and Thought*, Feigenbaum, E. A. & Feldman, J. (Eds.), McGraw-Hill: New York, 251-268.

VanLehn, K. (1983). Human procedural skill acquisition: theory, model and psychological validation, *Proc. AAAI-83*, 420-423.

Vere, S. A. (1975). Induction of concepts in the predicate calculus, *Fourth International Joint Conference on Artificial Intelligence*, 281-287.

Vere, S. A. (1977). Induction of relational productions in the presence of background information, *Fifth International Joint Conference on Artificial Intelligence*, 349-355.

Vere, S. A. (1978). Inductive learning of relational productions. In D. A. Waterman & Hayes-Roth, F. (Eds.), *Pattern-Directed Inference Systems*, Academic Press: New York.

Vere, S. A. (1980). Multilevel conterfactuals for generalizations of relational concepts and productions, *Artificial Intelligence*, 14, 138-164.

Winston, P. H. (1975). Learning structural descriptions from examples. In Winston, P. H. (Ed.), *The Psychology of Computer Vision*, McGraw Hill: New York.

Winston, P. H. & Horn, B. (1981). *Lisp*, Addisson Wesley: Reading, Mass.

Winston, P. H., Binford, T. O., Katz, B. & Lowry, M. (1983). Learning physical descriptions from functional definitions, examples, and precedents. *Proc. AAAI-83*, 433-439.

Winston, P. H. (1986). Learning by augmenting rules and accumulating censors. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol II. Tioga: Palo Alto.

Yovits, M. C., Jacobi, G. T., & Goldstein, G. D. (Eds.) (1962). *Self-organizing Systems*, Spartan Books: Washington, D. C.