Learning from Derived Oracles

Sharad Saxena
Department of Computer and Information Science
University of Massachusetts at Amherst
Amherst, MA 01003

COINS Technical Report 87-105
October 19, 1987

Contents

Abstract

A specification of a task to be performed is often not sufficient for the efficient execution of the task. In this report a method is presented by which an executable specification of the task, and a set of legal operators to perform the task, are used to automatically deduce the subgoals that help in the problem solving activity. Efficient procedures for achieving these deduced subgoals are then learned using inductive methods.

1 Introduction

One aspect of intelligent behavior is the ability to execute efficiently, instructions for a task given in general terms. These instructions form a specification for the task to be performed if they express the purpose of the desired task, without indicating an algorithm by which to perform the task. A specification of a computation may be unacceptable as a means of performing the computation if, either the computation is expressed in terms of procedures that are not executable by the computing agent, or if, the specification is expressed in terms of procedures that are very inefficient to evaluate on the computing agent. Therefore it becomes necessary to transform these specifications into procedures, that are computationally more efficient.

The approach taken here is to derive from the specification of the goal the specifications for the subgoals necessary for efficient execution of the task. For each subgoal efficient procedures to achieve the subgoal are then learned.

This report describes a project to apply this approach to two problems. The first problem is to learn efficient ways of playing the game of Tic-Tac-Toe from the definition of a "win" , and a set of legal operators that define the game. The subgoals necessary for a "win" are deduced as predicates over the board configurations, efficient classifiers for each subgoal are then learned by inductive means. The second problem is to synthesize efficient logic programs for a task whose executable specification is given along with a schema to achieve the goal. The procedures necessary for an efficient program are derived from the schema. For each procedure, a logic program is then induced.

The algorithms to induce efficient procedures for achieving the subgoals in both the domains require an oracle to answer queries about the domain being modeled by the induction algorithm. The derived subgoals, though inefficient to execute, by virtue of being executable, serve as oracles for these queries. The induced procedures are expressed in a language that makes them more efficient to evaluate than the corresponding oracles. As learning progresses, the induced procedures can answer queries about a larger portion of the domain and fewer queries are asked of the oracle.

2 The Learning Algorithm

The learning problem studied here is,

Given

1. A specification of the goal to be achieved.
2. The set of legal operators that can be applied to achieve the goal.

Find

Efficient procedures for achieving the goal.

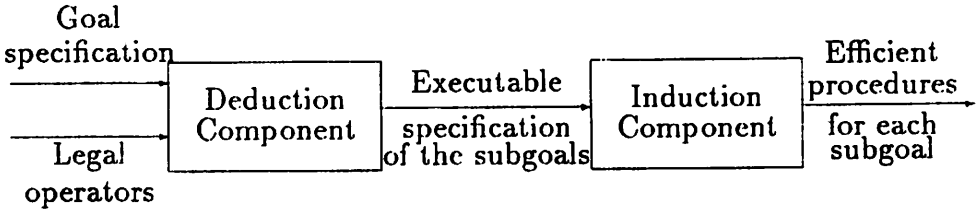

Figure 1: The learning method

The approach taken to solve the learning problem is illustrated in Figure 1.

The deduction component of the algorithm reduces the given goal specification and the legal operators into a set of subgoals necessary for problem solving. For each subgoal the induction component builds an approximation. In the Tic-Tac-Toe problem, the induction component is required to build an approximation for the subgoal as a disjunctive normal form propositional expression that has no negated variables. Such expressions are known as *monotone* DNF expressions.

For the problem of finding an efficient logic program from executable specification, the induction component is required to build an approximation that can be expressed as first order predicate calculus expressions, where the predicates are the procedures known to be required to achieve the subgoal. The procedures required to achieve a particular subgoal are known as a result of the deductive process.

The inductive component successively produces better approximations as it sees more examples. In addition each approximation is expressed in a language that makes the approximation more efficient to evaluate than the derived definition of the subgoal.

Figure 2 illustrates the events that occur when a subgoal is to be achieved. If the approximation is reliable enough, it is used for achieving the goal. Otherwise, the derived definition of the subgoal is used to achieve the subgoal. If Y is the result of achieving the subgoal X then the pair (X, Y) is given as the next fact to the induction algorithm. The induction algorithm uses this fact to produce a better approximation for the subgoal. This process continues until the approximation has the required reliability.

3 Learning Monotone DNF expressions

In [15] a method of integrating induction and deduction to learn to play the game of Tic-Tac-Toe efficiently– given a definition of the "win" predicate, and a set of legal operators to play the game– was described. The induction algorithm used in [15] was developed as a part of this project. In this section, first the method used in [15] to derive the subgoals to be achieved for achieving the goal is summarized. The induction algorithm used in [15] is then described in detail.

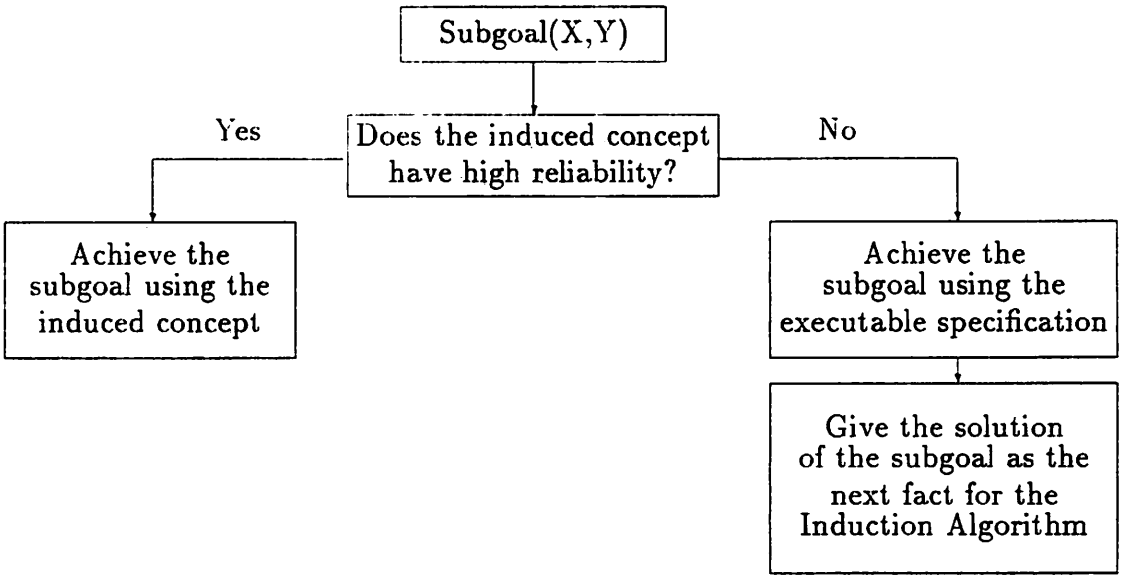

Figure 2: The learning algorithm

3.1 Deriving the Oracles

The definition of a "win" for *"x"* in the game of Tic-Tac-Toe, is expressed as a predicate over the board configurations b. This predicate, denoted by $win(\text{"}x\text{"}, b)$, says that any board configuration that has 3 *"x"*s in a row is a win for *"x"*. The learning system knows this predicate and the set of legal operators that can be used to play the game.

A sequence of moves that result in a win for *"x"* are then analyzed. Dijkstra's "weakest precondition" method is invoked to determine the necessary conditions at each step in the sequence of moves leading to a win for *"x"*. Starting at the end of the sequence, the method regresses facts that must be true or false at each step in the sequence, as in program verification [3]. At each step, the conjunction of facts defines the necessary conditions that guarantee that the tail of the sequence will lead to a win for *"x"*. The conjunction of necessary conditions is used directly as the definition of a subgoal to be achieved.

A *cause* predicate helps to express the weakest preconditions. The cause predicate is defined as:

$$cause(b, P) \equiv \sim P(b) \bigwedge (\exists x) P(domove(x, b))$$

This says that, condition P can be *caused* in the board configuration b if and only if, P is not already true, and there exists a move such that after that move P will be true.

For example the subgoal of being in a state where the game is has been won by *"x"* is expressed as

$$tic0(y) \equiv win(x, y)$$

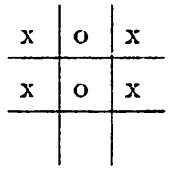

Figure 3: A Tic-Tac-Toe board

Similarly being in a state from which *"x"* cannot lose in 1 step is expressed as

$$tic1(y) \equiv \sim [\sim (win(y)) \bigwedge \text{more-moves(y)} \bigwedge cause(\overline{y}, tic0)]$$

here $\overline{y}$ denotes the board configuration where it is o's turn to make a move. The above expression says that *"x"* cannot lose in 1 step in the board configuration y, if it is not true that the board is not in a state where someone has already won, and there exists a move that will make it a win state for o. If such a move were to exist, o would select that move and win, therefore *"x"* would lose in 1 step.

Similar subgoal predicates for each move in the sequence of moves that led to a win for *"x"* are defined. Automatic deduction of these subgoals defines the facts that must be true for the goal to be true. The learning system can use these subgoal definitions to play the game. At any stage in the game the system should try to achieve one of the subgoals, in the order $tic0, tic1, \ldots$, and so on.

However notice that the cause predicate contains an existential quantifier. The cause predicate tests candidate moves to find the one that satisfies the predicate. The nested subgoal definitions therefore contain nested existential quantification. The result of evaluating such nested quantification is equivalent to doing a complete game tree search. The next section shows how this inefficiency can be removed.

3.2 Learning efficient recognizers for the oracles

In the previous section it was shown how subgoal predicates for achieving the goal can be derived for the game of Tic-Tac-Toe. This section explains how efficient recognizers for the subgoal predicates can be learned.

An induction algorithm is used to determine a fast recognizer for each subgoal predicate. The induction algorithm needs positive and negative examples of the concept being learned. The board configurations where the subgoal predicates evaluate to *true* serve as positive examples. The board configurations where the subgoal predicates evaluate to *false* serve as negative examples. A board configuration is represented by the contents of each square in the board. For example a board configuration like Figure 3 would be represented as

$$(x \text{ at } 0) \bigwedge (x \text{ at } 2) \bigwedge (x \text{ at } 3) \bigwedge (x \text{ at } 5) \bigwedge (o \text{ at } 1) \bigwedge$$
$$(o \text{ at } 4) \bigwedge (b \text{ at } 6) \bigwedge (b \text{ at } 7) \bigwedge (b \text{ at } 8).$$

The hypothesis space of concepts is represented by disjunctive normal form(DNF) ex-

pressions, with the propositional variables being the contents of each square. For example, a proposition that "x" is at the square 0 (upper left corner), would be represented by the variable (x at 0). A particular hypothesis is of the form

$$(l_{11} \bigwedge l_{12} \bigwedge \cdots l_{1j_1}) \bigvee \cdots \bigvee (l_{k1} \bigwedge l_{k2} \bigwedge \cdots l_{kj_k})$$

where

$$lm, n \in \{(x \text{ at } 1), (o \text{ at } 1), (b \text{ at } 1), \cdots, (x \text{ at } 9), (o \text{ at } 9), (b \text{ at } 9)\}$$

A property of this form of representation is that a hypothesis can be represented as a monotone DNF expression. If a particular literal is in the negated form, for example, say it is $\overline{x \text{ at } m}$, then this literal is logically equivalent to (o at $m \bigvee b$ at m). By replacing all negated literals by their logically equivalent forms an expression in DNF form can be converted into an expression in monotone DNF form.

3.2.1 An algorithm for learning monotone DNF expressions

L. G. Valiant [16][17] has presented an algorithm for learning concepts that can be described by monotone DNF expressions, using a *necessity oracle*. Given a vector of assignment of truth values to a subset of variables (the others being undetermined, that is, they could be either true or false), the *necessity oracle* for a concept will determine, whether the vector is a positive instance of the concept. If the vector is a positive instance, then so will be any vector obtained from it, by making some of the undetermined variables determined (that is, giving them the truth assignment of either True or False).

The algorithm works with positive examples. The algorithm successively drops various conjuncts to form generalizations, and asks the *necessity oracle*, whether making the dropped conjuncts undetermined results in a positive instance. If it does, then the conjunct is dropped, otherwise it is kept and the process is repeated for other conjuncts, until all conjuncts have been tried. The resulting vector of conjuncts is known as a *prime implicant* of the DNF expression and is added to the DNF expression being built for the concept. At the end, the algorithm has determined the literals that are essential in making this instance a positive example and removed the extraneous literals. Valiant's algorithm for learning monotone DNF expressions is reproduced in Figure 4, a more detailed discussion and analysis of the algorithm may be found in [16]

The algorithm uses the following notation, $A \Longrightarrow B$ means if A is true then B is true. A monomial m is a prime implicant of a DNF expression representing the concept F, if $m \Longrightarrow F$ and if $m' \not\Longrightarrow F$ for any m' obtained by deleting one literal from m. '$\star$', will mean that the corresponding literal is made undetermined. The *necessity oracle* called *ORACLE* will return T for a vector v, if the vector is a positive instance of the concept. t stands for the number variables in the example. The test $v \not\Longrightarrow g$ in Figure 4 amounts to asking whether none of the monomials of g are made true by the values determined to be true in v. Every time an example is produced such that $v \not\Longrightarrow g$, the inner loop of the algorithm will find a prime implicant m to add to g. Each m is different from any previously added (otherwise $v \Longrightarrow g$).

The algorithm is intialized with $g = FALSE$

$v \leftarrow NEXT - EXAMPLE$ {This should be a positive example.}

if $v \not\Longrightarrow g$ **then**

begin

for i=1 to t **do**

if P_i is determined in v **then**

begin

set $\tilde{v}$ equal to v but with $P_i = \star$

if $ORACLE(\tilde{v}) = T$ then $v = \tilde{v}$

end

set m equal to product of all literals q such that $v \Longrightarrow q$

$g = g \bigvee m$

end

Figure 4: Valiant's Algorithm for Learning Monotone DNF expressions

3.2.2 Valiant's method modified for learning efficient recognizers

One of the decisions to be made when training an efficient recognizer is; when does one stop training, and start using the efficient recognizer for performance tasks? This problem becomes especially poignant when multiple concepts are to be learned and the concepts may be defined in terms of each other. A wrong classification for a concept A not only results in degraded performance, but may also result in a noisy training instance for another concept B, that may be defined in terms of A. This makes it necessary to either have a noise immune induction algorithm, or to ensure that every time a classification is made using the fast recognizer it is reliable.

Valiant's algorithm has a property that it has only single sided errors. If a classifier built by using Valiant's algorithm classifies a particular configuration as a positive instance, *then it is a positive instance.* However if it classifies a particular configuration as a negative instance then it could be a positive instance. This single sided error property can be used to check whether a classification is reliable, by accepting only those classifications that are without error.

However if only error free classifications are accepted, then the negative instances can never be reliably classified by the fast classifier. This can be rectified by learning classifiers for both the concept and for the negation of the concept.

The subgoal predicates derived for the various concepts are used as the *necessity oracles*

	Classification: Positive	Classification: Negative
Fast-positive	Reliable	Unreliable
Fast-negative	Unreliable	Reliable

Figure 5: Reliability of classification

for the concepts. These predicates can also be used as *necessity oracles* for the negation of the concept. Therefore two classifiers for each concept are built, one for the concept(using positive examples), and one for the negation of the concept(using negative examples).

When required to classify a configuration, the configuration is classified using both the classifiers, if any one of the classifiers returns a reliable classification it is used immediately. If an unreliable classification is returned, the slow definition of the concept is used for classification purposes and the appropriate fast classifier is trained. This is summarized in fig. 5

3.3 Summary of the inductive element

During the course of a game, it is required to determine whether a particular board configuration belongs to a concept. The following sequence of events occur when this happens:

1. The fast classifier for the concept is used to classify the configuration.
2. If the classifier in 1 calls the configuration a positive instance, it is a reliable classification, the classification is returned.
3. If the classification in 1 is not reliable, fast classifier for the negation of the concept is used to classify the configuration.
4. If the classifier in 3 calls the configuration a negative instance, it is a reliable classification, the classification is returned.
5. If the classification is not reliable, the derived subgoal is used to classify the configuration. If the classification is a positive instance fast classifier for the concept is trained, otherwise fast classifier for the negation of the concept is trained.

As a result of the learning architecture and the ease of determining when the classifications returned by the induced classifiers are reliable, the following desirable properties are observed during learning.

1. No noisy training instances are generated.

2. Fast recognizers are usable for performance tasks even when they have not completely learned the concept.

3. No mechanism for switching over from derived definitions to induced classifiers is needed.

For the *necessity oracle* to determine whether a vector of literals is a positive instance it has to test, using the slow definition, whether all possible assignments to the undetermined variables still keep the instance positive. This makes the time required to train the fast classifier large. As a result if a concept is defined in terms of other concepts, the classifiers for which are not yet fully trained, then it may take a long time to train these high level classifiers. However as the lower level criteria get trained, the higher level criteria can get trained, not only because the lower level criteria are fast, but also because the lower level criteria no longer require any training.

4 Learning Logic Programs

This section illustrates how an efficient logic program to sort a list of numbers can be learned given an executable specification of the sorting task, and the set of operators to solve the task. This problem can be summarized as

Given

1. An executable specification for the program to be synthesized.

2. A program schema that specifies the abstract algorithm to be used to synthesize the program

Find An efficient program for the specification.

In the work reported here deductive and inductive approaches to program synthesis have been combined to derive the program. The program schema and the specification of the program are used to automatically deduce the subgoals necessary for an efficient program. Efficient procedures for achieving each subgoal are then inferred using inductive methods. The induction algorithm used here is Shapiro's *Model Inference System* [12],[13].

Shapiro's *Model Inference System* infers a model **M** for a given first order language **L**. Facts are presented as sentences of **L** that are either true or false in **M**. A brief description of the *Model Inference System* is included here. Complete details of this system can be found in [12][13].

The set of most general sentences in the language **L**, that are no longer than some parameter d, and that have not been refuted by the facts known so far, is maintained as a hypothesis of the model. If the sentences in the conjecture are ever discovered to imply a fact known to be false, then an error detecting algorithm, which Shapiro calls the *contradiction backtracking algorithm*, is invoked. The *contradiction backtracking algorithm* finds

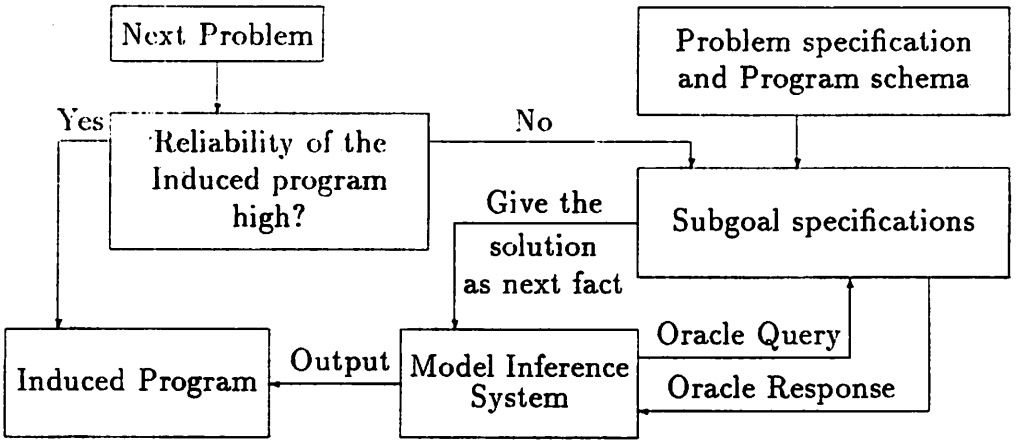

Figure 6: Interaction between various components used for deriving logic Programs

at least 1 false sentence in the conjecture. If the sentences forming the current conjecture are discovered not to imply some positive fact, then the parameter d is incremented and all sentences of size $\leq d$ that have not been refuted are added to the conjecture.

An oracle for the model **M** is needed by the *contradiction backtracking algorithm* to isolate false sentences from the conjecture. An oracle is also needed to determine whether a particular sentence in the language is refuted. These oracles need to answer the following types of questions about a particular sentence in the language.

1. Is the given ground sentence of **L** true or false in the model **M** ? [1]
2. What are the permissible set of values for the variables in the given sentence of **L** to make it true in **M** ? It is assumed that this set is finite.

The deduced subgoals serve as oracles for the concepts represented by the subgoals. The queries asked by the induction algorithm can be answered automatically by the derived oracles. The oracles try to achieve the appropriate subgoal by executing their specification. If the subgoal can be achieved then the sentence is true. This also gives the permissible values for the variables.

In addition to being an oracle, the *Model Inference System* also requires the user to name the predicates needed for the program being synthesized. In the approach taken here each derived subgoal is a predicate needed for the program being synthesized. Therefore the user does not need to know a-priori the predicates needed for the program. The deduction component finds the specifications for the predicates needed. These specifications are given a name by the user.

The architecture of the learning process and flow of data between various components is illustrated in Figure 6.

[1] A ground sentence is one that has no variables.

4.1 Deriving the Oracles

This section illustrates a method to automatically derive the specifications for the subgoals to be achieved for synthesizing a program to sort a list of numbers. The specification of a sorted list and a program schema specifying the abstract algorithm to do the sorting are given to the system. The system deduces the subgoals for doing Insertion sort by first deducing that a procedure that will insert an element into a sorted list and return a sorted list is needed. The system then uses the same program schema– which was used to derive the subgoals needed to synthesize a program for sorting a list– to synthesize a program for inserting an element into a sorted list such that the resulting list is also sorted. This attempt at using the schema to generate the subgoals does not give any new subgoals.

The facts known to the system during the deduction process are represented in PROLOG notation. In this notation a program is a finite set of *definite clauses*, that are universally quantified logical sentences of the form

$$A : -B_1, B_2, \ldots, B_k. \quad k \geq 0$$

where the A and the B's are logical atoms, also called goals. Such a sentence is read as "A is implied by the conjunction of the B's", and is interpreted procedurally as ,"To satisfy A, satisfy the goals $B_1, B_2, \ldots, B_k$.

Consider the following specification for a program to sort a list of integers in this notation.

```
sorted(X,Y) :- permutation(X,Y), ordered(Y).
```

The definition of *sorted* says that, Y is the sorted list X, if Y is a permutation of X and Y is ordered. A list is ordered if it is a increasing sequence.[2]

Also consider the program schema in Figure 7 to do the task of sorting. This schema says that if X is a primitive list then the solution can be found directly. Otherwise break X into head and the rest of the list. Solve the rest of the list and compose the solution. Notice that this schema is an instance of the divide and conquer strategy for algorithm design. Different ways of splitting the list will lead to different sorting algorithms. For example if the schema says, split the list into two parts, solve each part and compose the solution, then merge sort is got.

To deduce the procedures needed to synthesize Insertion Sort the facts known to be true after each step in the program schema given in Figure 7 are collected.

The first step in the schema is the test whether a list is *primitive*. The system knows that a list is *primitive* if it contains only 1 element. If now the 'No' branch from the test is taken, the next step is the *decompose* operator. The decompose operator breaks the list into the head and the rest of the list. Therefore the fact known to be true after the decompose operator is:

$$construct(E, X1, X).$$

[2]The predicates permutation(X,Y), and ordered(Y) are known to the system.

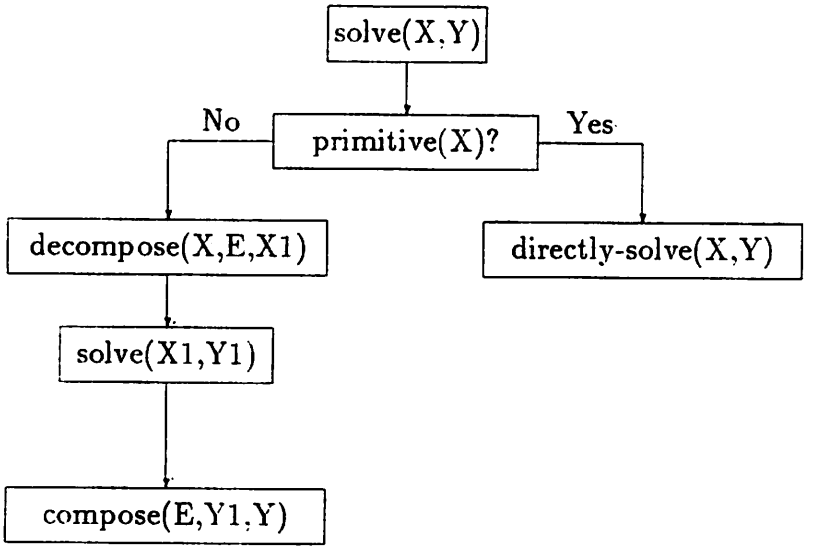

Figure 7: Program schema used for learning to sort

where *construct* is a list construction procedure that takes an element E and adds it as the head of the list $X1$ to give the list X. The next procedure in the schema is *solve*$(X1, Y1)$. This is the problem being solved, therefore its specification is known. The procedure *solve* takes the list X1 and produces the list Y1, therefore the facts known to be true at the end of the procedure solve are

$$construct(E, Y1, X), permutation(X1, Y1), ordered(Y1).$$

The next procedure in the schema is *compose*$(E, Y1, Y)$. The specification of any predicate gives the relationship between the variables of the predicate. One of the facts known to be true before *compose* is called is, *permutation*$(X1, Y1)$. This predicate has as its input parameter a variable $X1$ that is not an input parameter of *compose*. $X1$ is also not an output produced by the facts known to be true before *permutation* is called. As a result *permutation*$(X1, Y1)$ does not constrain either the input variables of *compose* or the outputs produced before. Therefore the predicate *permutation*$(X1, Y1)$ is dropped and the input specification of compose becomes:

$$construct(E, Y1, X), ordered(Y1).$$

After the *compose* operation has been done, the result should be a list of numbers that is sorted. Therefore after the *compose* operation it should be the case that

$$permutation(X, Y), ordered(Y).$$

Combining the input and the output specification of *compose* gives the definition of *compose* operator as:

$$construct(E, Y1, X), ordered(Y1),$$
$$permutation(X, Y), ordered(Y).$$

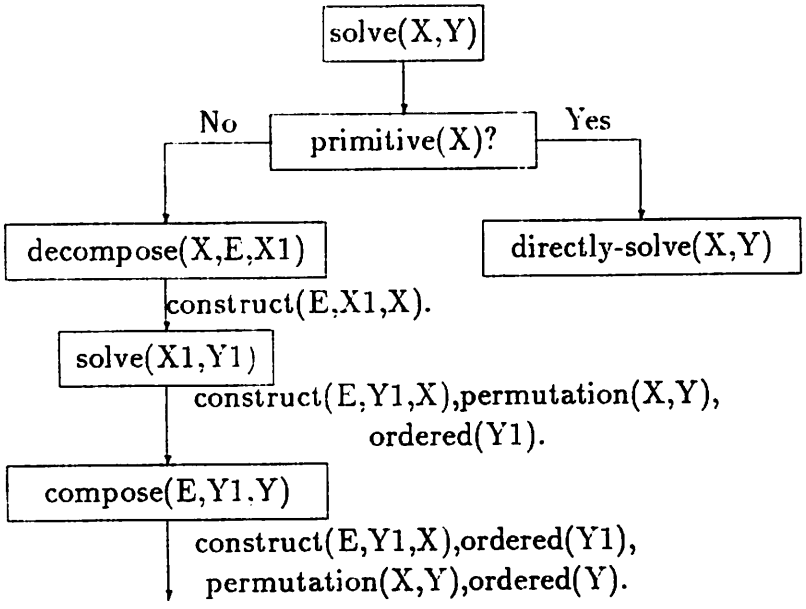

Figure 8: Decomposition of sorted(X,Y)

This new definition becomes a subgoal to be achieved. This process is illustrated in Figure 8. For the 'Yes' branch of the the *primitive*(X) predicate symbolic evaluation is used to find if any new subgoals are defined there. When a list X is primitive it contains a single element. Symbolic evaluation of *permutation*($[X], Y$),*ordered*(Y), gives *sorted*($[X], [X]$). Therefore *directly-solve* does not change the input. The decomposition of the specification of *sorted* gives us only 1 new procedure

$$compose(E, Y1, Y) \ :- \ construct(E, Y1, X), ordered(Y1), \\ permutation(X, Y), ordered(Y).$$

Call this procedure $insert(E, X, Y)$. Next this specification is analyzed using the same schema as in Figure 7, appropriately changed to reflect the new inputs. Decomposition of the *insert* operation using the schema is illustrated in Figure 9. The facts known to be true after each operation are listed after that operation. Notice that, when the specification for the procedure $compose(E1, Y1, Y)$ is being derived, the known facts true at that point are,

$$construct(E1, X1, X), construct(E, X1, Z1), ordered(X1), \\ permutation(Z1, Y1), ordered(Y1)$$

Of these facts the only fact whose input variables are a subset of the input variables of *compose* and the output variables in the facts known to be true before that fact is $ordered(Y1)$. The input specification of *compose* is therefore $ordered(Y1)$. The output specification of *compose* is the set of facts that must be true after the schema has been used to solve the *insert* problem, therefore the output specification of *compose* is:

$$construct(E, X, Z), ordered(X), permutation(Z, Y), ordered(Y)$$

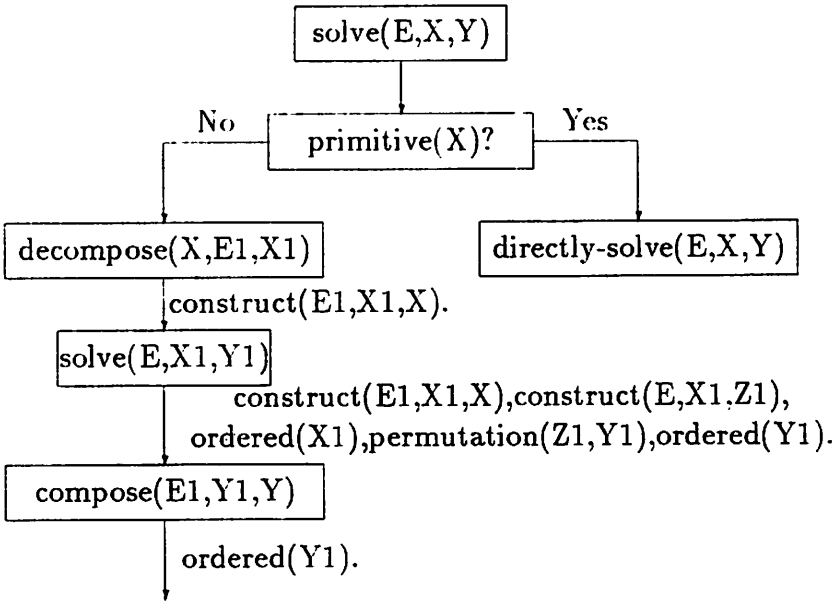

Figure 9: Decomposition of the insert procedure

None of these facts have their input variables as the subset of the input variables of *compose* and the outputs of the facts collected so far (namely $ordered(Y1)$). The specification of compose then becomes $ordered(Y1)$. This is a known goal. Therefore no new procedures are added by the 'No' branch of the schema. Symbolic evaluation of the *directly-solve* procedure yields:

$$Solve(E,[E1]) \equiv E =< E1,[E,E1]$$

The Directly-solve procedure gives the $=<$ operator as a subgoal. This is a subgoal known to the system, therefore it is not decomposed further.

4.2 Learning efficient recognizers for the oracles

From the program schema the subgoals that have to achieved, to achieve the goal, were derived. The *decompose* operator is a known function in PROLOG, therefore it is not mentioned explicitly as a procedure called for achieving a goal. For the sorting problem (call it *Sorted*), the analysis of previous section showed that:

```
Sorted calls Sorted, Insert.
```

For the *Insert* procedure no new subgoals were derived, except $=<$ and *ordered*, which are operations known to the system. Therefore

```
Insert calls Insert,=<,ordered.
```

To induce efficient programs for these procedures Shapiro's *Model Inference System* is used. The specifications for the various subgoals serve as oracles for the concept they

represent. The oracle queries that are generated by the induction algorithm are answered automatically by executing these specifications.

For the problem of sorting a list of numbers a session with the Learning system is presented in the Appendix.

4.3 Summary of the inductive element

The following events take place when a particular query about the model being inferred is asked.

- If the reliability of the inferred model is high then the query is answered using the inferred model.
- If the reliability of the inferred model is not high, then the query is answered using the derived subgoals.
    1. The query and its solution are presented to the *Model Inference System* as the next fact.
    2. If the *Model Inference System* generates any oracle queries, then these queries are answered using the derived oracles.
    3. The *Model Inference System* returns the updated conjecture of the model

5 Reliability Issues

The induced programs try to approximate the functions computed by the oracles. If the approximations are computationally more efficient, then it is desirable to use them instead of the specifications. However if it is required that the same function be computed by both the appoximation and the oracle, an algorithmic way of determining whether the approximation should be used or not is needed. The approximation is perfect if it gives the same result as the oracle everywhere where the oracle computation converges. However in general, given only a finite amount of information, it is not possible to know when the approximation is perfect. In absence of an algorithmic way of determining whether an approximation is perfect, it can be ensured that if the computation by the approximation converges it gives the same result as the oracle.

In this section first the fact that there exists no algorithmic way of determining that the approximation is perfect is proved. Then the weaker condition mentioned above is formally stated. A method for ensuring that this condition is met while learning monotone DNF expressions is illustrated, and a way of ensuring that this condition is met while learning logic programs is suggested.

Let $\varphi(x)$ be the function computed by the oracle.[3] Let $\psi(x)$ be the function that approximates $\varphi(x)$.The approximation would be perfect if it satisfies the following requirement.

Completeness Requirement

$$(\forall x \in dom(\varphi))[\varphi(x) = y \Longrightarrow \phi(x) = y].$$

That is, for all elements of the domain for which the oracle computation converges, the appoximation converges and gives the same answer.

There can exist no algorithmic way to determine when the completeness requirement has been met. Because, if $dom(\varphi)$ is infinite then there exist functions that cannot be approximated by any finite amount of information. This fact is proved using the diagonalization principle.

Induction tries to identify an efficient function that has the same input-output characteristic as the oracles. Therefore the identification paradigm of learning [1] is used. This paradigm is summarized here.

An inductive inference machine is an algorithmic device or a Turing machine that works as follows:
First the machine is put into some initial state with its tape memory completely blank. From there it proceeds algorithmically except that from time to time the device requests an input or produces an output. Each time it requests an input an external agency feeds the machine a pair of natural numbers (x,y) or a '*', and then returns control to the machine. Typically the input is printed in some designated area of the tape memory, or on some auxiliary tape in such a manner that the machine may scan and make use of it. The outputs produced by the machine are all natural numbers and are to be interpreted as an index of a computable function in an effective enumeration[4] of all computable functions. It is assumed that this effective enumeration is known to the inductive inference machine.

Let φ be a computable function. $\overline{f}$ is an enumeration of φ if $\overline{f} = (a_0, a_1, ...)$ is an infinite sequence in which a_i is either a pair $[x, f(x)]$ or a '*', and further more $[x, f(x)]$ appears at least once in $\overline{f}$ for every $x \in dom(\varphi)$

$M[\overline{f}] \downarrow i$ denotes that M with input $\overline{f} = (a_0, a_1, ...)$ converges in limit to i, if whenever $(a_0, a_1, ...)$ are fed in this order to M, there eventually comes a time when M produces a i and never again produces a different number.

M is said to identify φ if for every enumeration $\overline{f}$ of φ there exists a i such that $M[\overline{f}] \downarrow i$ and ϕ_i is an extension of φ, that is $\phi_i(x) = \varphi(x)$ for all $x \in dom(\varphi)$. Here ϕ_i stands for the ith computable function in the effective enumeration of computable functions.

Given the above model it is extended to finite sequences in the following manner. Let f^* be a finite prefix of $\overline{f}$. $M[f^*] = j$, denotes that if j is the last number output by M on

[3]Without loss of generality consider only functions of one variable. By use of pairing functions any function of more than 1 variable can be converted into a function of 1 variable.

[4]An effective enumeration is one in which there is a computable procedure, such that given the index of a function in the enumeration, it gives the function.

being given the input $f^{\cdot}$, and say that ϕ_j is the current conjecture. In particular $M[\lambda] = 0$, where 0 is the index of the computable function that diverges everywhere and λ is prefix consisting of no elements.

Now consider the function defined as follows

$$\varphi(x) = \begin{cases} \phi_j(x) + 1 & \text{where } M[f^{\cdot}] = j, \text{ if } \phi_j(x) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

where $f^{\cdot}$ is the finite prefix seen by M so far.

Clearly φ is computable. However, there cannot exist an inductive inference machine M that will approximate φ, given any finite prefix of $\overline{f}$. Intuitively φ is the function, that for a given input finds out what the result of the computation , by the current conjecture, would be for a particular value, and gives a different answer.

Even though the *completeness requirement* cannot be met we can try to meet a weaker requirement.

Correctness requirement

$$(\forall x \in dom(\varphi))[\phi(x) = y \Longrightarrow \varphi(x) = y].$$

That is, if the approximation converges at a point in the domain, then the answer is correct, otherwise the approximation does not converge. Trivially the *correctness requirement* can be met by letting ϕ be all the known values, and making ϕ diverge everywhere else.

The *correctness requirement* suggests a strategy to be used for determining the the appropriate moments for using the induced approximation. If at each stage an approximation that satisfies the correctness criteria exists, then for a given instance of the query, if the query can be answered in a specified resource bound, the result is correct. If however, the computation does not converge in the specified resource bound, then the oracle definitions can be used to answer the query. The result obtained from the oracle computation can be used to provide the next fact for the induction algorithm.

5.1 Reliable Learning of Monotone DNF expressions

Valiant's algorithm for learning Monotone DNF expressions has a property of *single sided errors.* If at any stage of learning the induced classifier, classifies an instance as a member of the concept, then it is correct. However if it classifies an instance as not being a member of the concept, the instance could be a member of the concept.

Classifiers for both the concept and its complement can be learned, this process has been explained in section 3.2. If either of them return a value that classifies the instance as a member of their concept, then that classification is reliable. Otherwise the oracle is used to get the classification of the instance, this classification serves as a training instance for the inductive classifier.

It should be noted that for a finite number of propositional variables, the set of instances

is finite, therefore the completeness requirement can also, in theory, be met. The easiest way to do this would be to check for each member of the domain, whether the induced classifier returns the same answer as the oracle. However for large domains this is not practical.

5.2 Reliable Learning of Logic Programs

Let S be a specification for a logic program P and let R be a query. P is said to conform to the specification S, with respect to query R, if and only if,

$$(\forall \xi)[(P \vdash R(\xi)) \Longrightarrow (S \vdash R(\xi))].$$

where ξ is the $n-$tuple of arguments of the query R.

In principle, the correctness of any ξ computed by a particular execution of P can be investigated by trying to deduce $R(\xi)$ from S, denoted by $S \vdash R(\xi)$. More generally, it would be desirable to prove the partial correctness of P to be able to deal with all possible computed solutions.

This can be done by exploiting the transitivity of logical implication. Suppose that it can be shown that every procedure in P is logically implied by S. If $S \vdash P$ holds, then for any $n-$tuple ξ, if we have $P \vdash R(\xi)$, then $S \vdash R(\xi)$ holds by transitivity of $\vdash$. That is, any logic program is partially correct if its procedures are logically implied by its specification. The expression for partial correctness criterion therefore simplifies to $S \vdash P$.

A way to incorporate this correctness requirement in the *Model Inference System* would be to mark as 'false' all sentences of size $\leq d-$ where d is the current value of the complexity parameter– that are not logically implied by the specification of the oracle. However the question, whether a sentence is logically implied by a set of axioms, is in general, only semi-decidable. A bound is therefore needed on the number of steps allowed before it is concluded that a sentence is not logically implied by the specification. Further work is needed on finding ways to determine this bound. In the system that has been implemented reliability is measured by recording the number of queries that were answered correctly in past.

6 Related Work

The process of learning efficient procedures for the derived concept can be viewed as the *operationalization* of the derived concepts.

Operationality is defined in [5] as:

Given

- A concept description

- A performance system that makes use of the description.
- Performance objectives specifying the type and extent of system improvement desired.

Then

The concept description is considered *operational* if it satisfies the following requirements:

1. **Usability:** The description must be usable by the performance system.
2. **Utility:** When the description is used by the performance system, the system's performance must improve in accordance with the specified objectives.

In the work reported here each derived oracle is the concept description. The learning architecture presented in Figure 2 is the performance system. The performance objectives are specified as learning the concepts represented by the oracles, in a specific language. For the game of Tic-Tac-Toe, this language is the set of monotone DNF expressions, where the propositional variables are the contents of the various board positions. For logic programs, the language is the set of first order predicate calculus sentences over the predicate names defined by the oracles. Using the above terminology, the oracle definitions are *useful*, but have low utility, and therefore they are not operational. The aim of the learner is to get at an equivalent definition that is both useful and has high utility.

In [5], Keller has approached the problem of operationalization by using a set of transformation operators. The application of transformation operators results in the search of the concept description space. A control strategy, that favors those transformations that have improved the system performance in past, is used to reduce the search.

The work reported here presents an architecture to integrate deductive and inductive approaches to learning. The architecture uses deductive methods to automatically formulate learning subproblems. For each learning subproblem inductive methods are applied to arrive at efficient procedures for solving that subproblem. Lebowitz[7] and Pazzani, et al.[10] have approached the problem of integrating inductive and deductive approaches to learning differently. They use inductive methods to propose theories. Deductive methods are used to explain new facts based on these theories. These explanations are then generalized to cover similar situations and are stored for future use.

The section on learning logic programs relates to the work on program synthesis. Two major approaches to program synthesis have been:

1. Deriving a program for a given specification by either using theorem proving techniques or by direct application of transformations and rewriting rules, to the specification. Manna and Waldinger's work [8],[9], and Burstall and Darlington's work [2] are representative of these approaches.

2. Inferring a program from examples of its input output behavior. [13] contains a chapter on Program synthesis that is representative of this approach.

The problem with a purely deductive approach to program synthesis is that all the facts about the domain, that may be needed to synthesize a given program, may not be known a-priori. Similarly all the rewrite rules necessary for a particular synthesis to go through may not be known before the synthesis starts. This work presents a way to integrate the deductive and inductive approaches to program synthesis. The places where the deductive methods fail, inductive methods may be useful. For example, in the insertion sort problem illustrated here, it was found that the problem of inserting an element in its proper position in a sorted list, could not be solved by the schema given. However here a program to do this was synthesized using inductive methods. Though the synthesized program is guaranteed to produced correct results only for the observed instances, *model inference system* produces programs that generalize to give correct results for instances that have not been observed.

Smith in [14], has described a system for synthesizing a program from a problem specification and a program schema– which he calls a *design strategy*. The approach taken here– of decomposing a specification subgoals required for achieving the goal– is similar to the approach in [14]. In [14], the decomposition process continues until the specification of a subgoal matches the specification of a known operator. The known operator then produces a program segment in the target language. In the work reported here, inductive methods are used when it is found that further decomposition of the problem cannot be done with the current knowledge, and that the subgoal specification does not match any known operator.

Hogger in [4], describes a nonmechanical method to derive logic programs from specifications. The logic program is developed as a sequence of goals $(G_1, \ldots, G_f)$ where G_f is the synthesized logic program. Each goal G_{r+1} is derived from from its predecessor G_r, by the application of two types of inference rules, which are called *goal simplification* and *goal substitution*. This method is also a purely deductive procedure and requires a complete theory to be present for a program to be successfully synthesized.

Prieditis and Mostow [11] have developed a PROLOG meta-interpreter called PROLEARN for adapting to a particular execution environment. The meta-interpreter monitors a particular execution sequence and generalizes the execution by converting constants to variables and simplifying the resulting expressions by using partial evaluation. The generalized execution is a new rule which is used to solve future problem instances.

7 Conclusions

This work presents a method to integrate the deductive and inductive approaches to machine learning. Deduction can be used to derive useful concepts from a known theory. However if the theory is intractable or incomplete inductive methods may be

required. For the game of Tic-Tac-Toe, boolean expressions for the various subgoals– with the propositional variables being the contents of various board positions– can be derived directly from the subgoal concepts. The expansion and simplification of the various terms of the subgoal concepts will achieve this. But the simplification of boolean expressions is known to be intractable. By using inductive methods approximations for these concepts can be learned. While synthesizing insertion sort it was found that the given schema was not sufficient to synthesize a program for "insert" operation. The theory here was not complete to derive all the required facts. Here too inductive methods made it possible to synthesize the program.

The work reported here is only a preliminary investigation of ways of integrating deductive and inductive methods. There are a number of issues that have to be addressed if this approach is to be a general way of transforming high level specifications for a task into efficient procedures for doing that task. Some of the issues are:

1. What is the level of detail at which the operators must be specified ? The two extremes are giving all the subgoals, leaving nothing to be deduced by the learner, and giving nothing, leaving it for the learner to find the necessary operators.

2. What are the general data structures and algorithms that can be used to derive subgoals from a specification?

3. Developing induction algorithms that guarantee the partial correctness of the induced procedures.

As an extension of this work, general algorithms for deriving the specifications for the subgoals from the specification of the goal are being investigated.

8 Acknowledgments

This work was done under the guidance of Professor Paul E. Utgoff. The idea of combining Deduction and Induction into a single integrated system was his. I cannot isolate a single idea in this work that did not either come directly from him, or that did not arise out of a discussion with him. His meticulous reading of the earlier drafts of this report and suggestions for improvement have significantly improved this presentation. Any shortcomings that still remain in this work or in the presentation are because of my inability to rise to his standards. I would like to thank him for his time, support and encouragement.

Professor David A. Mix Barrington read an earlier draft of this report. This report has improved considerably because of his suggestions and comments.

I would like to thank Margret P. Connell and Harpreet Sawhney for always being interested in knowing what I was trying to do and for patiently listening to my explanations. The discussions with them always helped me to clarify the issues to myself.

I would also like to thank Peter S. Heitman for helping me with PROLOG.

There is a lot that I have to thank my friends Kartik Venkataraman, Sandeep K. Gupta, V. Krishnan and S. Raghuram for. It is because of them that my stay in Amherst continues to be very pleasant. They have always been very patient with me. Their willingness to listen to me and put things in proper perspective whenever I have had "things" to discuss has helped me a lot.

It is not possible for me to thank with words, those to whom I owe everything. Without the constant love, support and sacrifice of my parents and my sister, I could not have even begun.

9 References

[1] Blum,L. and Blum,M. *Towards a Mathematical Theory of Inductive Inference*, Information and Control, 28, 125-155, 1975.

[2] Burstall,R.M., and Darlington, J. *A transformation system for developing recursive programs* Journal of the ACM 24,1,(Jan 1977), 44-67,1977.

[3] Dijkstra, E.W. *A Discipline of Programming*, Prentice Hall,1976.

[4] Hogger, C. J. *Derivation of Logic Programs*, Journal of the ACM, vol 28, No 2, April 1981, pages 372-392, 1981.

[5] Keller, R.M. *The Role of Explicit Contextual knowledge in Learning Concepts to Improve Performance*, Technical Report ML-TR-7, Department of Computer Science, Rutgers University, 1987.

[6] Keller, R.M. *Defining Operationality for Explanation- Based Learning*, Proceedings AAAI-87,pages 482-486, 1987.

[7] Lebowitz, M. *Not the Path to Perdition: The Utility of Similarity-Based Learning*, Proceedings AAAI-86, pages 533-537, 1986.

[8] Manna, Z. and Waldinger, R. *A Deductive Approach to Program Synthesis*, ACM Transactions on Programming Languages and Systems, vol 2, No 1, pages 90-121, 1980.

[9] Manna, Z. and Waldinger, R. *The Deductive Synthesis of Imperative LISP Programs*, Proceedings AAAI-87, pages 155-160, 1987.

[10] Pazzani, M. , Dyer M. and Flowers M. *The Role of Prior Causal Theories in Generalization*, Proceedings AAAI-86, pages 545-550, 1986.

[11] Prieditis A.E. and Mostow J. *PROLEARN: Towards a PROLOG that learns*, Proceedings AAAI-87, 1987.

[12] Shapiro, E.Y. *Inductive Inference of Theories from Facts*, Technical Report 192, Yale University, Department of Computer Science,1981.

[13] Shapiro, E.Y. *Algorithmic Program Debugging*, MIT Press, 1982.

[14] Smith, D.R. *Top Down Synthesis of Divide and Conquer Algorithms*, Artificial Intelligence 27(1985) 43-96. 1985.

[15] Utgoff, P.E. and Saxena, S. *Learning via Induction and Deduction*, Unpublished Report, COINS Department, University of Massachusetts, 1987.

[16] Valiant, L.G. *A Theory of the Learnable*, Communications of the ACM, vol 27, no 11,1984,pp 1134-1142.

[17] Valiant,L.G. *Deductive Learning*, Philosophical Transactions of the Royal Society of London,series A, no 312, 1984,pp 441-446.

Appendix

In this appendix we present a session with the system that infers a logic program for Insertion sort. User input is indicated in italics.

User first inputs the specification for the program and the schema to be used to infer the program.

subgoals([(sorted(X,Y) : − permutation(X,Y),ordered(Y))]).

[For the goal]

(sorted(X,Y): −permutation(X,Y),ordered(Y))

[Give the Schema as a list of Clauses,follow by the Schema specification]

[the format is [[schema],(schemaspec)]]

[[(sorted(X,Y): −primitive(X),directly(X,Y)),
(sorted(X,Y): −decompose(X,Z1,U1),sorted(U1,W1),compose(Z1,W1,Y))],
(sorted(X,Y): −permutation(X,Y),ordered(Y))]

The system then asks the user to give the types of the variables, + is used to indicates that the corresponding variable is an input variable and - to indicate that the corresponding variable is an output variable.[ ] indicate that the corresponding variable is a list.

[Declare,sorted(X,Y)]

sorted(+[x],-[x]).

[Declare,compose(X,Y,Z)]

compose(+x,+[x],-[x]).

From the above information the system derives the specification for *insert* and asks the user to give a name to the specification.

[the following spec is called]

[construct(X,Y,Z),ordered(Y),permutation(Z,W),ordered(W)]

insert(X,Y,Z)

The procedures required by **induced_sorted** have now been determined.

[Procedures called by ,induced_sorted(X,Y), are ,induced_sorted(Z,U), and ,induced_insert(V,W,X1)]

The user is asked to specify the schema to be used for synthesizing a program for the *insert* operation.

[For the goal]

[construct(X,Y,Z),ordered(Y),permutation(Z,W),ordered(W)]

[Give the Schema as a list of Clauses,follow by the Schema specification]

[the format is [[schema],(schemaspec)]]

[[(insert(X,Y,Z): −primitive(Y),directly(X,Y,Z)),
(insert(X,Y,Z): −decompose(Y,W1,X2),insert(X,X2,U2).compose(W1,U2,Z))],
(insert(X,Y,Z): −construct(X,Y,W),ordered(Y),permutation(W,Z),ordered(Z))]

The user is then queried about the type of the variables of *insert*.

[Declare,insert(X,Y,Z)]

insert(+x,+[x],-[x]).

From the information about the schema to be used and the derived specification of the *insert* operation, the procedures required by the **induced-insert** operation have been determined.

[Procedures called by ,induced_insert(X,Y,Z), are ,induced_insert(U,V,W),
and ,X1=<Y1,ordered(Z1)]

The analysis of the problem specification and the program schema to derive the subgoals required for achieving the goal is now complete. The **Model Inference System** is now used to get efficient programs for these derived subgoals. The subgoal definitions will be used to answer the oracle queries generated by the **Model Inference System.**

The first sorting problem is submitted to the system.

execute(induced_sorted([2,3,1],X))

[checking facts]
[Error: missing solution,induced_sorted([2,3,1],[1,2,3]),diagnosing...]

The induction algorithm asks a query :
induced_sorted([3,1],X) ?

The specification of sorted is used to get the answer:
[Executing specification to get the answer]
[The answer is ,induced_sorted([3,1],[1,3]),true]

The induction algorithm next asks the query:
induced_insert(2,[3,1],X) ?

The specification of insert is executed to get the answer:
[Executing specification to get the answer]
[The answer is ,induced_insert(2,[3,1],X),false]

Similar queries about **sorted** and **insert** are asked until the **Model Inference System** finds a clause that is not false and can be used to solve the given problem.

[Found clause: ,(induced_sorted([X|Y],Z): −induced_sorted(Y,V),induced_insert(X,V,Z))]
[listing of,induced_sorted, :]

(induced_sorted([X|Y],Z): −induced_sorted(Y,V),induced_insert(X,V,Z)).

The known facts are then checked to see whether the program induced so far can indeed solve the problem.

[checking facts]
[Error: missing solution,induced_sorted([2,3,1],[1,2,3]),diagnosing...]

[Found clause: ,(induced_sorted([],[]): −true)]
[listing of induced_sorted]

(induced_sorted([X|Y],Z): −induced_sorted(Y,V),induced_insert(X,V,Z)).
(induced_sorted([],[]): −true).

At this stage the system has synthesized the program to do the sorting. But the insert operation has not yet been synthesized. The various queries about the insert operation that were asked and answered by the oracle are the facts that are now checked.

[checking facts]
[Error: missing solution,induced_sorted([2,3,1],[1,2,3]),diagnosing...]

[Found clause: ,(induced_insert(X,[],[X]): −true)]
[listing of induced_insert]

(induced_insert(X,[],[X]): −true).

This process of checking facts continues until a set of clauses that can solve the given problem is found. These clauses are:

[listing of induced_insert]
(induced_insert(X,[],[X]): −true).
(induced_insert(X,[Y],[Y,X]): −true).
(induced_insert(X,[Y|Z],[Y,X|Z]): −Y=<X).
(induced_insert(X,[Y],[X,Y]): −X=<Y).

[checking facts]
[no error found]

The next sorting problem is now given to the system.
execute(induced_sorted([3,2,1],X)).

[Induced Program is incorrect]
[The correct answer is]
induced_sorted([3,2,1],[1,2,3])

The system finds the correct answer and checks to find the cause of error. It then searches for a clause that will solve the problem. Each time the system is given a problem the system checks the facts, if the answer is wrong it finds the wrong clause, removes it from the program, and then searches for the clause that will sove the problem. After few more problem instances the system finds the correct program for insert.

[listing of,induced_insert, :]

(induced_insert(X,[],[X]): −true).
(induced_insert(X,[Y],[X,Y]): −X=<Y).
(induced_insert(X,[Y|Z],[Y|V]): −induced_insert(X,Z,V),Y=<X).
(induced_insert(X,[Y,Z|U],[X,Y,Z|U]): −X=<Y,Y=<Z).

[checking facts]
[no error found]

The synthesis is now complete. Notice that the last clause for **induced_insert** has an unnecessary test Y=<Z. The list into which an element is to be inserted is always sorted, therefore Y=< is always true. This fact is generated because in the synthesis process, before the procedure for *sorted* was completely synthesized, a query **induced_insert(2,[3,1],X)** is generated. The fact that this query cannot be true for any X is recorded. The test Y=<Z is present to account for this fact.