Refinement Strategies for Inductive Learning Of Simple Prolog Programs

Marc Kirschenbaum
Mathematics & Computer Science Dept.
John Carroll Univesity
Cleveland, Ohio 44118
U.S.A.

Leon S. Sterling
Computer Engineering & Science Dept.
Case Western Reserve University
Cleveland, Ohio 44106
U.S.A.

Abstract

This paper extends Shapiro's Model Inference System for synthesizing logic programs from examples of input/output behavior. A new refinement operator for clause generation, based upon the decomposition of Prolog programs into *skeletons,* basic Prolog programs with a well-understood control flow, and *techniques,* standard Prolog programming practices is described. Shapiro's original system is introduced, skeletons and techniques are discussed, and simple examples are provided, to familiarize the reader with the necessary terminology. The Model Inference System equipped with this new refinement operator is compared and contrasted with the original version presented by Shapiro. The strengths and weaknesses of applying skeletons and techniques to synthesizing Prolog programs is discussed.

1 Introduction

Inductive learning of concepts, given a set of examples and counterexamples, has been given a lot of attention in the Artificial Intelligence community. This paper concerns a special case of inductive learning, synthesizing Prolog programs from examples of their input/output behavior. An incremental inductive inference algorithm was developed in [Shapiro, 1983] for synthesizing logic programs. Shapiro named his implementation the Model Inference System (MIS).

MIS has several components including: detection and removal of a false clause, detection of the inability to prove a goal known to be true, and the ability to find a new clause to justify the known truth of a goal. Anyone experimenting with MIS quickly discovers that some programs are easy to learn, others can be synthesized with difficulty, and others are beyond the scope of the system. The reason for the variation in performance can be traced to the refinement operator used to produce new clauses, and the search strategy employed to determine if the new clause correctly implies the examples. Hence the refinement operator, due to its influence upon the scope of the system, is the focal point for this paper.

This paper will describe MIS with one of its refinement operators and with a new refinement operator based on work on decomposing Prolog programs into skeletons, basic Prolog programs with a well-understood control flow, and techniques, standard Prolog programming practices. In contrast to Shapiro's refinement operator, which checks and adds new clauses one at a time, the new operator produces all refinements and then checks the clauses generated. In fact, every time MIS tries to learn a new clause the refinement operator goes through the same order of clause generation. By checking previously refuted clauses, the program refrains from repeating its mistakes. We will denote MIS equipped with our new operator as the Model Inference System with Skeletons and Techniques (MISST).

In MISST the refinement operator consists of two phases. The first phase matches the necessary data structures with a skeleton - an appropriate control flow for the program. This generation of a skeleton is accomplished by creating a template using only the input arguments from the program to be synthesized. Once the skeleton is created, the second phase enhances the skeleton by applying a technique to it. Each technique will generate a program to be checked for correctness. Examples of the types of programs which are hard or impossible to learn and those easy to learn will be given for each system. We will examine the implications of the results and cite areas for future research.

2 The Model Inference System

The Model Inference System is an implementation of an incremental inductive inference algorithm. Given a set of examples and counterexamples of a new concept, MIS produces a set of Horn clauses to represent the concept. Whenever the current set of clauses prove a counterexample true, the proof tree is used to determine the faulty clause in the set which is then removed. If there exists an example not explained by the current set of clauses, a new clause is generated using a refinement operator. A major assumption of the system is that an oracle exists which knows the truth or falsity of any particular ground instance of the concept to be learned.

The refinement operator determines the type of Prolog programs which can easily/not easily be learned through MIS. One refinement operator given in [Shapiro, 1983] was specialized for generating clauses for definite clause

grammars whereas a second operator was presented as a more general operator. Each operator was designed for synthesizing a different type of program. In this paper, we use the generalised refinement operator for all comparisons with MISST.

MIS needs to have access to certain knowledge to successfully synthesize a logic program. The following two types of knowledge are specific to the intended target program and are supplied by the user.

Declarations about the type and mode of each variable are used to determine how the variables will be instantiated.

Information about 'allowable' predicates guides clause creation. An added goal is related to the other goals in the clause through the instantiation of variables as specified by the type and mode declarations.

Other knowledge is included as part of the MIS database. For example, there is a list of instantiations for each type. A list variable can be instantiated to [] or [X|X.].

The general refinement operator performs in the following way:

1. Instantiate output variables to some input variables, removing those variables from the yet to instantiate output list.
2. Instantiate inputs in the head of the clause to one in the list of possibilities. In the case of lists, this could create two new input variables, one for the head of the list and one for the tail.
3. Instantiate outputs in the head of the clause to one in the list of possibilities. In the case of lists, this could create two new output variables, one for the head of the list and one for the tail.
4. Unify two input variables which are selected at random.
5. Add a goal which generates some output.
6. Add a goal for test purposes. The input variables for the new goal are selected from the input variables for the head.

MIS has specialized search strategies to determine if a clause generated by the refinement operator covers a particular goal. A clause $A \leftarrow B_1, B_2, \cdots, B_n$ *covers* a goal A if there is a substitution θ such that $A\theta = A'\theta$ and $B_i\theta$ are true for $1 \leq i \leq n$. The three given strategies described in [Shapiro, 1983] are called eager, lazy, and adaptive. The eager strategy will find a clause to cover the goal in question and, if necessary, query the user to determine the truth of the goals in the body. This is a powerful strategy in the sense that it will go beyond the current set of facts to synthesize a program. The obvious drawback is the numerous interactions required with the user. The lazy strategy behaves in an opposite manner, only using goals known to be true when checking the examples against the hypothesized clauses. The lazy strategy is less powerful than the eager one but does have the advantage of not requiring any assistance from the user. The adaptive strategy is a combination of the previous two. Like the lazy approach, the adaptive strategy will not query the user, but it will try to see if a goal is correct by not only checking the facts for that goal but also by checking the facts for the body of the clause. The strategy choice is important as to which kind of programs can be synthesized as can be seen with the well-known program *append,* given below, which can only be synthesized using the eager strategy.

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

The complete MIS algorithm is given in Figure 1. The repeat loop in Figure 1 is bounded by the allowed depth of the proof tree. The default depth used is 25.

The MIS Algorithm

Given a possibly empty set of Horn clauses (background information), goals to be called by the target concept p, false-solutions = [], and true.solutions = [].

repeat

  Read the next example or counterexample of p and add it to the corresponding list of true or false solutions,

  repeat

    If it is possible to derive a fact in false_solutions then find a false clause and remove it from the set of Horn clauses representing the hypothesis.

    If it is impossible to derive a fact in true solutions then generate the refinements until a previously untried clause covers this fact. Add this clause to the set of Horn clauses.

  until neither of the If tests is entered

  Output the current set of Horn clauses.

forever

Figure 1

3 Skeletons and Techniques

Wirth presented the idea of stepwise refinement as a methodology to be used during program development to produce clear, well structured programs [Wirth, 1971]. Currently, the collection of Prolog examples available from the literature is lacking in structure. We have developed the method of stepwise enhancement to provide this missing structure [Kirschenbaum and Sterling, 1990; Lakhotia and Sterling, 1990].

Stepwise enhancement delivers a structured and procedural approach to Prolog program development which can be described as follows: When a new problem is attempted, isolate the basic control flow needed to solve the problem and embody it in a *skeleton.* Once the skeleton has been determined, extra computations are included by applying appropriate programming methods, which we call *techniques,* to yield an *extension.* Separate extensions can be combined to produce the desired product. The extension(s) can then be regarded as another skeleton allowing us to repeat the process until the final program has been developed. The number of

refinements made during the top-down development of the program will determine how often the above process will be repeated.

Stepwise enhancement can be applied to manipulating recursive data structures, one of Prolog's strengths, since the various methods available for handling recursive data structures can be naturally partitioned into several skeletons based on a common control flow. These skeletons constitute the basic building blocks for program development. For example, if we need to process an entire list of elements, we might want to use the following skeleton:

```
traverse([]).
traverse([X|Xs]) :-
  traverse(Xs).
```

If we want to process a list until we find a particular element then the appropriate skeleton is:

```
search([X|Xs]).
search([X|Xs])
  search(Xs).
```

Two points need mentioning here. First, a slight modification in the base case produces a different skeleton. Second, the only purpose of a skeleton is to drive computations built upon it.

In contrast to the characterization of skeletons in terms of control flow, techniques should be conceived in terms of the specific goal to be accomplished. For example, the appropriate method for counting the number of elements of a list, the number of nodes in a tree, or the number of goal reductions or depth in a proof tree, has been to increment an argument and then carry it as a context. Standard Prolog programming practices, or techniques, like this one have the common feature that they build upon an already existing program. The technique *calculate* can be applied to *traverse* to produce a program to find the length of a list. There are different variants of *calculate,* one for each type of arithmetic operation that could be added to the skeleton as an extra goal.

Intuitively, one can think of skeletons as the mechanism which controls the program whereas techniques determine what is done with the data. Skeletons, therefore, make explicit the control flow that the program is expected to follow. A consequence of this is our restriction of induction to Prolog programs rather than logic programs more generally since logic programs are nondeterministic and therefore display no predetermined control flow. The separation of control flow from technique is the main idea for providing extra structure for program synthesis. For more information about skeletons and techniques see [Kirschenbaum and Sterling, 1990].

4 MISST's Clause generation

In our prototype MISST system, we have restricted our attention to list data structures. The skeletons are generated by choosing one or more list arguments to recurse upon and by determining all the reasonable base cases. We have made the same assumption as MIS that there exists an all knowing oracle to answer the queries posed by the system. We also assume the presence of information detailing allowable predicates for clause creation.

An interesting aspect of skeleton generation occurs when the target program uses other predicates for testing and updating arguments. Predicates like $\leq$ and $>$ require the comparison of two variables which, in all generality, may be any of the variables in the head of the clause or even worse it may be output variables from some other goal in the body of the clause. For example, here are a few of the reasonable possibilities for a skeleton predicate with one list, two element variables and which uses < and >.

```
sk([X|Xs],Y,Z) :-
  X ≤ Y,
  sk(Xs,Y,Z).

sk([X|Xs],Y,Z) :-
  X ≤ Z,
  sk(Xs,Y,Z).

sk([X|Xs],Y,Z) :-
  Y ≤ Z,
  sk(Xs,Y,Z).

sk([X|Xs],Y,Z) :-
  X > Y,
  sk(Xs,Y,Z).
```

Without using some knowledge we will generate too many possibilities. The following quick estimate clearly shows why. Let

- N = the arity of the skeleton predicate
- p_i = number of arguments in the predicate *pi*
- M = number of possible predicates to be used in the skeleton
- $S = \Sigma_{i=1}^{M} P_i$.

Then, in the special case where each of the M predicates is included in the skeleton once, and all the arguments come from the head of the clause, we will generate N^s skeletons. Therefore the total number of skeletons generated is at least $0(N^S)$. An example of a poorly generated skeleton would be

```
sk([X|Xs],Y,Z) :-
  X ≥ X,
  X > X,
  sk(Xs,Y,Z).
```

Therefore, we have added the following knowledge to MISST.

1. Mode of variables - unify input variables for the extra predicates only with the input variables for the head of the clause or with an output variable from another predicate.
2. Type of variables - only unify variables of the same type.
3. Incompatible predicates - a predicate *exclusive (P red l,Pred2, Type)* is used to indicate if two predicates can only be used together under certain restrictions. For example, < and $\geq$ cannot be used in the same clause if they have the same arguments in the same order. Another situation we would like to avoid would be having both $X < Y$ and $Y > X$ in the same clause. For binary predicates this can be accomplished by having a predicate *opposite(Predl,Pred2),* which will tell if the two predicates produce the same results if the arguments are

switched. A third category checks for symmetry. We do not want to produce one clause having $X \neq Y$ and another clause which is identical except it contains $Y \neq X$.

4. Last recursive call - every recursive skeletal clause has the recursive predicate as the last goal.

This information is used both in the creation of the clauses and for pruning the redundant clauses. In the sense that this is knowledge about the program, it could be called meta knowledge. Related terms in machine learning are background knowledge and preference criteria. The effect of the knowledge is to bias what programs are learned. The skeleton creation phase can be thought of as a description language to determine which concepts are describable. This use of a restricted hypothesis space is one kind of bias description found in [Utgoff, 1986].

The only technique we have implemented so far is *collect.* This technique adds one output variable to the sought after predicate. Depending upon the other goals, the current head of the list is added or not to the output variable. This technique is similar to *calculate* mentioned earlier in that it adds an output variable, and depending upon the other goals in the clause, the value of the head of the list is combined with other values to be put into the output variable. An example of applying a collect technique to the skeleton search is to return the list remaining after the element has been found. In this case, the head of the input list is never added to the output list. This generates the program search_with_remainder(Xs,Ys).

```
search_with_remainder([X|Xs],Xs).
search_with_remainder([X|Xs],Ys) :-
  search_with_remainder(Xs,Ys).
```

5 Comparisons between MIS and MISST

MISST is given a fact for the target Prolog program and creates all possible skeletons using the knowledge mentioned above. These in turn are given to an enhancement module to produce all possible extensions to those clauses. The component of MIS which removes incorrect clauses when counterexamples are supplied is applied to the generated clauses to produce the final program. Thus, the skeleton creation and the various techniques in the enhancement module determine the possible programs to be learned.

MISST will synthesize programs having only lists and elements as variables and either has no output variables or the output variable is used for some type of collection. This last restriction can be overcome by including more techniques but it isn't clear at this point how slow the system will become as more techniques are added. Some of the programs used for comparison include: *prefix* (3.13), *suffix* (3.13), *append* (3.15), *sublist* (3.14), *member* (3.12), *nonmember* (7.5), *select* (3.19), and *subset* (7.7). The number in parenthesis after each of the above predicate names refers to the program number used in The Art of Prolog [Sterling and Shapiro, 1986]. We also used predicates *union, difference,* and *intersection* in our comparison. The code for *union* is

```
union([ ],Ys,Ys).
union([X|Xs],Ys,[X|Zs]) :-
  nonmember(X,Ys),
  union(Xs,Ys,Zs).
union([X|Xs],Ys,Zs) :-
  member(X,Ys),
  union(Xs,Ys,Zs).
```

The code for *difference* and *intersection* is similar to the code for *union.* In each case, the third argument exemplifies a collect technique.

Unfortunately, since MISST is based upon MIS, it also cannot remove redundant clauses. The program learned for *prefix* given below demonstrates this unwanted behavior.

```
prefix(X,X).
prefix([],[]).
prefix([],X).
prefix([X|Xs],[X|Ys]) :-
  prefix(Xs,Ys).
```

Removing redundant clauses is, in general, too computationally expensive to be practically implemented. One possible way of avoiding redundant clauses is to take a correct set of hypothesized clauses and form a new set of clauses by selecting a base clause plus all the recursive clauses and check if the examples can be proven by this new set of clauses. If not, repeat the above with a different base case. Of course, in general, a program might require two or more base cases and it is also possible that some of the recursive clauses will be redundant.

Prefix, suffix, and *append* were easy for both systems to learn. Not surprisingly, *member, select* and *append* were very easy for MIS to learn since that refinement operator gives a high priority to expanding and unifying elements on a list. MISST found *member, select* and *append* to be nontrivial programs to learn due to the large number of clauses created by the refinement operator. Once these clauses have been generated, it is necessary to feed the system counterexamples to weed out the inappropriate clauses. *Subset* was difficult for MISST to learn without *select* being learned first because the first program produced for *select* in conjunction with the one produced for *subset* will have an infinite loop in it. MISST will recognize the non-terminating condition but, due to the large default depth bound and the numerous clauses to be tested, this is a very slow process. *Union* has the same problems if *nonmember* and *member* are not synthesized first. It took a lot of memory to complete the synthesis of *union* even when its called goals are synthesized first. The code generated for *subset* is

```
subset([],[]).
subset([],X).
subset(X,X).
subset([X|Xs],Ys) :-
  select(X,Ys,Ys1),
  subset(Xs,Ys1).
subset([X|Xs],[X|Ys]) :-
  subset(Xs,Ys).
subset(Xs,[Y|Ys]) :-
```

```
    select(Y,Xs,Xsl),
    subset(Xsl,Ys).
subset(X,[Y|Ys]) :-
    subset(X,Ys).
```

The expected Prolog program is comprised of the second and fourth clause above. The reason why *select* was easy for MIS to learn, caused MIS to fail to learn both *subset* and *union* even when the eager search strategy was employed. Memory ran out! The refinement operator used for MIS gives priority to expanding the variables in contrast to adding goals to a clause. The variable [X|Xs] will be expanded to [X,Y|Zs] making it difficult to learn a program which calls numerous goals. Both *subset* and *union* make use of the predicates *member* and *nonmember*.

A nice feature of MISST can be seen in the synthesis of the program for *union.* If *nonmember* and member are previously known to the system, the user does not interact with the system from the time the initial declarations are made until the list of possible clauses are generated. The ability to free the user from system queries during the clause generation stage will generalize for any target program that has all of its called goals previously synthesized.

The weakness of MISST is the volume of clauses produced. The removal of the duplicate clauses produced by MISST, before they are asserted as hypothesized clauses, is the most expensive operation in the system. After the duplicate clauses have been removed it is still necessary to feed the system counterexamples to weed out the inappropriate clauses. However, judicious selection of counterexamples will limit the number required by the system. A possible solution to this problem is to modify MISST to keep all the generated clauses in a list to be processed one at a time

Due to the combinatorial explosion of clause generation, a restriction was made in MISST to not allow an expanded variable [X|Xs] to have both X and Xs in the same goal in the body of the clause. This restriction made it impossible to synthesize a program to remove repeated elements in a list. If the above restriction is removed, *union* will explode with generated clauses. A possible solution to this is to have different search strategies similar to what is found in MIS.

6 Discussion

We believe this work will have a greater impact upon learning when knowledge is introduced to direct the generation of skeletons. This is truly the bottleneck in MISST. We may need to add knowledge in the form of a relationship between the number of elements in lists, or which list is to be recursed upon, or something else to minimize the number of possible skeletons being generated. If we think of skeleton creation as the 'getting started' process of learning, then MISST's difficulty is that it is does not know how to get started. This is a typical problem for learning systems.

The knowledge required to determine which of all the possible skeletons are appropriate for a particular learning exercise is not currently captured in MISST. One possible approach to reduce the complexity of clause generation without requiring too much from the user would be to ask the user to give the position of the argument(s) to be recursed upon. For *union,* the number of clauses generated would be cut in half.

A promising source of insight for structuring skeleton creation is the work of [Deviile, 1990]. The knowledge included in his concept of logic specification overlaps with the knowledge in MISST described in section 4 and also contains extra information, for example multiplicity. Deville's use of induction schemes in logic descriptions may also be relevant since modifying the logical description of a skeleton can be interpreted as a shifting to a weaker bias as described in [Utgoff, 1986]. The required heuristic methods for deciding exactly how to modify the skeleton creation process needs to be determined.

Work done by [Muggleton and Buntine, 1988] presents a framework for Prolog induction which introduces new predicates into the language it is supplied. Inverting resolution is the tool used to weaken the bias description. At this time, we do not know how usefull this will be for skeleton generation.

Acknowledgments: We thank the ProSE discussion group for stimulating conversation. The work of Leon S. Sterling was supported in part by NSF grant CCR-9000387.

References

[Deviile, 1990] Y ves Deviile. *Logic Programming.* Addison-Wesley, 1990.

[Kirschenbaum and Sterling, 1990] Marc Kirschenbaurn and Leon S. Sterling. Prolog Programming Using Skeletons and Techniques. CAISR Technical Report TR-90-109, Case Western Reserve University, Submitted to the ACM Transactions of Programming Languages and Systems.

[Muggleton and Buntine, 1988] S. Muggleton and W. Buntine. Towards Constructive Induction In First-Order Predicate Calculus. In *The Turing Institute* TIRM-88-031, An Academic Associate of the University of Strathclyde.

[Lakhotia and Sterling, 1990] Arun Lakhotia and Leon S. Sterling. Stepwise Enhancement A Variant method of Incremental Programming. In *Proc. Conference on Software Engineering,* Skokie, IL, June, 1990.

[Shapiro, 1983] Ehud Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[Sterling and Shapiro, 1986] Leon S. Sterling and Ehud Y. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Utgoff, 1986] Paul E. UtgofT *Machine Learning - An Artificial Intelligence Approach Volumn IL* Morgan Kaufmann Publishers, Inc., 107-148, 1986.

[Wirth, 1971] Nicholas Wirth. Program Development by Stepwise Refinement. *Communications of the ACM,* 14(4):221-227, 1971.