An Algorithm that Infers Theories from Facts

Ehud Y. Shapiro[1]

Department of Computer Science
Yale University
New Haven, CT 06520

Abstract

A framework for inductive inference in logic is presented: a *Model Inference Problem* is defined, and it is shown that problems of machine learning and program synthesis from examples can be formulated naturally as model inference problems. A general, incremental inductive inference algorithm for solving model inference problems is developed. This algorithm is based on Popper's methodology of conjectures and refutations [11]. The algorithm can be shown to identify in the limit [3] any model in a family of complexity classes of models, is most powerful of its kind, and is flexible enough to have been successfully implemented for several concrete domains.

The Model Inference System is a Prolog implementation of this algorithm, specialized to infer theories in Horn form. It can infer axiomatizations of concrete models from a small number of facts in a practical amount of time.

/. Introduction.

A *model inference problem* is an abstraction of the problem faced by a scientist, working in some domain under a fixed conceptual framework, performing experiments and trying to find a theory capable of explaining their results.

In a model inference problem we assume some unknown model M for a given first order language L. We distinguish two types of sentences in *L. observational sentences,* which correspond to descriptions of experimental results, and *hypotheses,* which can serve as explanations for these results. The model inference problem is.

> *Given the ability to test observational sentences for their truth in some unknown model M, find a finite set of hypotheses, true in M, that imply all true observational sentences.*

J J. Some Model Inference Problems,

An example of a model inference problem is illustrated in Figure 1. In this example the domain of inquiry is the Integers, and the given first order language contains one constant 0, the successor function X', and three predicates:

X<Y for X is less than or equal to Y, plus(X,Y,Z) for X plus Y is 2 and times(X,Y,Z) for X times Y is Z Assume that w< test whether these relations hold between concrete numbers, is, we can test ground (variable-free) atoms such as 0 plus(0',0',0") and tim*s(0",0",0"") for their truth in *M* It setting, the model inference problem is to find a finite s sentences that are true of arithmetic and imply all true gr< atoms. Figure I shows such a set of sentences. We use the arrow — to stand for "is implied by". The sentence *P- QA* read "f is implied by the conjunction of *Q* and R".

Figure 1: Inferring Arithmetic.

The Domain: Integers.

The Language:
- **0 - zero**
- **X' - the successor of X**
- **X≤Y - X is less than or equal to Y**
- **plus(X,Y,Z) - X plus Y is Z**
- **times(X,Y,Z) - X times Y is Z**

Examples of facts:
- **0≤0' is *true***
- **plus(0,0',0) is *false***
- **times(0",0",0"") is *true***

Theory:

```
X≤X
X≤Y' ← X≤Y

plus(X,0,X)
plus(X,Y',Z') ← plus(X,Y,Z)

times(X,0,0)
times(X,Y',Z) ← times(X,Y,W) & plus(W,X,Z)
```

Note that we do not need to discover all the properties o functions and predicates involved to solve this model infer problem. In particular, the above set of axioms does not cor axioms for associativity of addition, transitivity of the orde relation, etc.. If *T* is a set of sentences true in *M* that implic ground atoms of L true in *M* then T" is called an *atomic-comi axiomatization* of *M.* The set of sentences in Figure 1 ii atomic complete axiomatization of arithmetic. It has I inferred by the Model Inference System from 36 facts ii seconds of CPU time.

Another type of inductive inference problem that natui fits in this framework is *program synthesis from exam* [1,4,14]. The task is to infer a program inductively, g examples of its input-output behavior. This task can be rest as a model inference problem, and in this case the programs t

'This work was supported in part by the National Science Foundation, grim No. MCS8002447.

This paper is an informal summary of the results described in [12].

inferred are *logic programs* [7,8,10].

A logic program is a collection of Horn clauses, which are universally quantified sentences of the form */ * - Qv A Q% a\ ... & Qn* for n>0, where /»and the $Q^{\%}$% are atoms. Such a sentence is read *"P* is implied by the conjunction of the *QY* and is interpreted procedurally "to satisfy goal *P,* satiify goals *QuQt*....fin" If the *(?*s are missing, the sentence reads *"P* is true** or "goal *P* is satisfied**. If *P* is missing, the sentence reads "the g's are false**, or "satisfy the *QY* A collection of Horn clauses can be executed as a program, using this procedural interpretation.

Figure 2: Inferring Logic Programs.

The Domain: Lists

The Language: [] - nil
[X|Y] - the "cons" of X and Y
append(X,Y,Z) - appending X to Y is Z
reverse(X,Y) - X is the reverse of Y

Examples of facts:
append([a,b],[c,d,e],[]) is *false*
append([a,b],[c,d,e],[a,b,c,d,e]) is *true*
reverse([a],[b,a]) is *false*
reverse([a,b,c],[c,b,a]) is *true*

Theory:
```
append([],X,X)
append([A|X],Y,[A|Z]) ← append(X,Y,Z)

reverse([],[])
reverse([A|X],Y) ← reverse(X,Z) & append(Z,[A],Y)
```

Figure 2 shows two logic programs. In this example of a model inference problem, the language *L* contains the two place function symbol [X)Y] (the Prolog list constructor, the equivalent of the LISP function cona). the constant [] (Prolog's nil) and the two predicates append(X,Y,Z) and reverae(X.Y). The model *M* for this language is defined as follows: the elements of *M* are all lists constructed from [X|Y] and []; the atom append(X,Y,Z) is true in *M* just in case the list Z is the result of appending the list X to the list Y, e.g. append(|a,b,c),[d,e),[a_fb,c,d,e]); the atom reverae(X.Y) is true in *M* just in case that the list Y is the reverse of the list X, e.g. reverae(Ia,b,c),[c,b,a]). The Horn clauses in Figure 2 are an atomic-complete axiomatization of the model thus defined, and are also Prolog programs for computing append and reverae. The Model Inference System synthesized the logic program for append in 11 CPU seconds from 34 facts* and a similar program for reverae from 13 facts in 6 CPU seconds.

In this paper we restrict ourselves to model inference problems in which the hypothesis language is the Horn clauses and the observational language is the ground atoms of *L.* In [12] we discuss a more general setting, and show that problems of grammatical inference and concept learning can also be formulated as model inference problems. Our restricted model inference problem is the following:

> *Given the ability to test ground atoms for their truth in some unknown model At, find an atomic-complete Horn axiomatization ofM.*

1.2. Solutions to Model Inference Problems.

An algorithm that solves model inference problems it called a *model Inference algorithm.* Such an algorithm tests the truth of ground atoms in the model, and once in a while produces a conjecture, a collection of Horn clauses. Since a finite number of facts about a model can not in general determine it uniquely among all possible models, and since a model inference algorithm always bases its conjectures on a finite number of facts, it is bound to make mistakes. The most one can expect of a model inference algorithm is that after examining a finite number of facts about the model, and making a finite number of wrong conjectures, the algorithm will correctly conjecture an atomic-complete axiomatization of a model and never change its conjecture afterwards. Following Gold [3], we say that in such a case the algorithm *identifies the model in the limit.* Note that a model inference algorithm cannot, in general, determine whether it actually has identified a model.

One inductive inference technique that can be used to solve model inference problems is enumeration. Algorithm I exemplifies this technique.

Algorithm 1: An Enumerative Model Inference Algorithm.
Let $T_1,T_2,T_3,...$ be some enumeration of
all finite sets of Horn clauses of *L*.
Set k to 0.
repeat
Examine the next fact.
while T_k is either too strong or too weak with
respect to the known facts do
Set k to k+1.
output T_k.
forever.

We say that a conjecture T is *too strong* with respect to some model *M* if it implies an observational sentence false in *M.* We say that it is *too weak* if it does not imply an observational sentence true in M. Note that a conjecture can be simultaneously too strong and too weak. The test of whether a conjecture is too strong or too weak is, in general, undecidable. To implement Algorithm 1 we choose some fixed complexity bound and use is to bound the resources allocated to this test. The complexity bound we choose determines the class of models the algorithm can identify in the limit, as explained below.

Let *h* be some total computable function from ground atoms to non-negative integers. A finite set of Horn clauses T is called *h-easy* if its atomic consequences are easy to derive modulo h, that is, if *T* derives a in at most *h(a)* derivation steps for almost every (that is, for all except a finite number of) ground atom o such that *T* implies *a.* A model *hi* is *h-easy* if it has a finite atomic-complete h-easy axiomatization. Using this concept, we can characterize the power *of* Algorithm I: if we choose such an *h* to bound the resources allocated to the test in the while loop, then the algorithm can identify in the limit exactly h-easy models.

Although the enumerative approach is *very* powerful, its inherent inefficiency limits its practical applications. Making better use of the syntactic and semantic properties of logic, we

develop an incremental algorithm that overcomes some of this inefficiency without loss of power. In developing the incremental algorithm we focus on two questions:

1. *How to weaken a conjecture if it is discovered to be too strong?*
2. *How to strengthen a conjecture if it is discovered to be too weak?*

If we have a solution to these questions, we could develop an algorithm as in Figure 3.

Figure 3: A Scheme for an Incremental Algorithm.

```
Choose some initial conjecture T.
repeat
    Examine the next fact.
    repeat
        while the conjecture T is too strong do
            Weaken it.
        while the conjecture T is too weak do
            Strengthen it.
    until the conjecture T is neither too strong nor too weak
        with respect to the known facts.
    Output T
forever.
```

2. Refuting Hypotheses with Crucial Experiments,

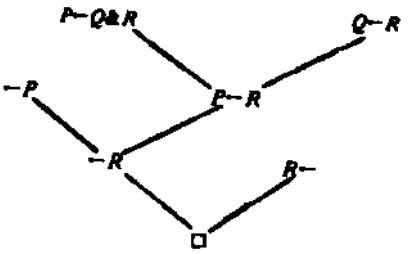
> "The only thing the experiment teaches us is that among the propositions used there is at least one error; but where this error lies is just what it does not tell us**
>
> -- Pierre Duhem,
> *The Aim and Structure of Physical Theory*
>
> "...it has to be admitted that we can often test only a large chunk of theoretical system, and sometimes perhaps only the whole system, and that, in these cases, it is a sheer guesswork which of its ingredients should be held responsible for any falsification**
>
> - Karl R. Popper,
> *Conjectures and Refutations: The Growth of Scientific Knowledge*

If a conjecture is too strong, i.e. it implies a false observational sentence, one can conclude that at least one of its hypotheses is false. In this section we develop the *contradiction backtracing algorithm,* which can detect such an hypotheses by performing *crucial experiments* in the model. The conjecture can then be weakened by removing this false hypothesis from it.

Crucial experiments are experiments that have a potential to decide between competing theories. A successful crucial experiment can eliminate at least one of the theories by providing a counterexample to its prediction. Although one crucial experiment can, in general, refute only a collection of hypotheses, the contradiction backtracing algorithm suggests a way of sequencing crucial experiments, which guarantees singling out a particular false hypothesis. The algorithm can be applied whenever a contradiction is derived between some hypotheses and the facts. Its input is an ordered resolution tree of the empty sentence Q from a set of hypotheses and true observational sentences 5. The algorithm assumes that an oracle for *M,* that can tell the truth of all ground atoms of *L,* is given. The algorithm performs a finite number of experiments in M, bounded by the depth of the derivation tree, and outputs an hypothesis peS which is false in *M.*

We demonstrate what the algorithm does on the propositional calculus example in Figure 4. In this example $S=\{P\leftarrow Q\&R,\ Q\leftarrow R,\ \leftarrow P,\ R\leftarrow\}$. The resolution tree is ordered so that the atom resolved upon appears in the condition of the left son and in the conclusion of the right son of the resolvent.

Figure 4: Backtracing Contradictions in Propositional Logic.



The algorithm starts from the root □, and iteratively tests the atoms resolved upon. If the atom is true in M it chooses the left subtree, otherwise the right subtree, until it reaches a leaf. The hypothesis in the leaf is false in *M.* In the illustrated example, assume that the hypothesis $P\leftarrow Q\&R$ is false, which means that *R* and *Q* are true in *M* and *P* is false. The algorithm first tests whether *R* is true in M, and since, by our assumption, it is true, it chooses the left subtree. Next it tests *P,* finding that it is false in the model, and chooses the right branch. Finally it tests *Q,* finds it to be true, chooses the left branch which leads to a leaf, outputs the hypothesis $P\leftarrow Q\&R$ which is false in *M* and terminates.

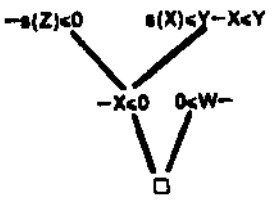
The contradiction backtracing algorithm for a first order language is based on the same idea of detecting a false hypothesis by systematically constructing a counterexample to it, although the way this counterexample is constructed is slightly more involved. The technique used is based on collecting substitutions in a resolution proof, which is similar to the one suggested by Green [5] and is used by the Prolog interpreter.

There is, however, another complication in the predicate calculus case. Since the atom *P* resolved upon need not be ground, one cannot always test its truth directly with the oracle for *M.* The solution is to instantiate *P* to a ground atom before testing it. The choice of how to instantiate *P* is arbitrary, but once it has been made all further experiments should be done with the same substitutions, in order for their results to constitute a counterexample to the hypothesis reached in the leaf.

The following is an example of the use of contradiction backtracing by a model inference algorithm, while it is trying to infer an axiomatization of the relation $\leq$ over the natural numbers. Assume that the algorithm already conjectured the hypotheses $0\leq X\leftarrow$ and $\leftarrow s(X)\leq 0$, and encountered the fact $\langle s(0)\leq s(s(0)), true\rangle$. It suggested the hypothesis $\forall(X)\leq Y\leftarrow X\leq Y$, so together with the hypothesis $0\leq X\leftarrow$ the sentence $s(0)\leq s(s(0))\leftarrow$ can be derived. However, after adding the new hypothesis the derivation in Figure 5 can also be constructed. So we can apply the contradiction backtracing algorithm and find which of the three hypotheses involved is false.

The atom resolved upon at the root is $0\leq 0$. The oracle is called on $0\leq 0$ and answers *Urue*\ so the left branch is chosen. The atom resolved upon at that node is $s(X)\leq 0$, and applying the

Figure 5: Backtracing Contradictions in Predicate Logic.



substitution used for X we get s(0)≤0. Since this atoms is false, the right branch is chosen, and the leaf s(X)<Y←X<Y is detected to be false. The counterexample constituted by the results of the experiments is s(0)≤0←0≤0. Note that not every ground instance of the hypothesis s(X)<Y←X<Y is false. For example s(0)≤s(0)←0≤s(0) is true.

The contradiction backtracing algorithm is applicable just as well to general clauses, and it can be shown that in either case the number of experiments needed to single out a false hypothesis is bounded by the depth of the resolution tree.

Since the Prolog interpreter automatically maintains all the substitutions as the resolution proof progresses, it is extremely simple to implement the contradiction backtracing algorithm in it. In [12] we describe such an implementation and discuss its application to the debugging of logic programs.

3. Refining Refuted Hypotheses.

This section addresses the question of how to strengthen a conjecture that is too weak. For this task we devise *refinement operators.* Intuitively speaking, a refinement operator suggests a logically weaker plausible replacement to a refuted hypothesis. If a conjecture is too weak, it can be strengthened by adding to it refinements of previously refuted hypotheses. The following is an example of one particular refinement operator used by the Model Inference System.

Let $p \in L$ be a sentence. Then q is a refinement of p if one of the following holds:

1. **$p=\square$ and $q=a(X_1,X_2,\ldots,X_n)$, for some n-place predicate symbol a of L, $n\geq 0$, and $X_1,X_2,\ldots,X_i$ are n distinct variables.**
2. **$p=P$ for some atom P, $q=P[U/V]$, where U and V are distinct variables occurring in P.**
3. **$p=P$ for some atom P, and $q=P[V/f(X_1,X_2,\ldots,X_n)]$ for some n-place function symbol f of L, $n\geq 0$, where V is a variable that occurs in P and $X_1,X_2,\ldots,X_n$ are distinct variables not occurring in P.**
4. **$p=a(t_1,t_2,\ldots,t_n)$ for some n-place predicate symbol a and terms $t_1,t_2,\ldots,t_n$, and $q=a(t_1,t_2,\ldots,t_n)\leftarrow a(X_1,X_2,\ldots,X_n)$, where $X_1,X_2,\ldots,X_n$ are distinct variables such that X_i occurs in t_i for $1\leq i\leq n$.**

We call sentences of the form $P\leftarrow Q$ that satisfy condition 4. above *context-free transformations.* It can be shown that any context-free transformation can be generated from the empty sentence via a finite sequence of refinements, using this refinement operator. Some non-trivial predicates have an atomic complete axiomatization via atoms and context-free transformations. Examples are the order relation and addition over integers (Figure I), the subsequence relation over lists, concatenation relations over lists (Figure 2), and the subtree relation for binary trees.

With every refinement operator we associate an hypothesis language: the set of sentences that can be generated from the empty sentence via a finite sequence of refinement operations. The hypothesis language associated with the above refinement operator contains all atoms and context-free transformations of *L* We say that one refinement operator is more general than another if the hypothesis language associated with the first contains the hypothesis language associated with the second. There are some immediate generalizations of the refinement operator described above, whose hypotheses languages suffice to axiomatize binary tree isomorphism, multiplication, exponentiation, string reversal and insertion sort. These refinement operators are implemented in the Model Inference System.

The effect of a more general refinement operator is a more powerful, though less efficient algorithm. If the syntactic class of the intended axiomatization is known, one can tailor a refinement operator for that class, thus increasing the efficiency of the algorithm. We may not always have such information, however. To ensure the theoretical completeness of this approach, we show in [12] the existence of a most general refinement operator.

4. A General, IncrementalModelInference Algorithm.

In the last two chapters we have developed mechanisms to weaken and strengthen the logical power of a conjecture when needed. We can instantiate now the algorithm scheme in Figure 3, and obtain Algorithm 2.

Algorithm 2: An Incremental Model Inference Algorithm.

```
Set T to [□].
repeat
    Examine the next fact.
    repeat
        while the conjecture T is too strong do
            Apply the contradiction backtracing algorithm,
            and remove from T the refuted hypothesis.
        while the conjecture is too weak do
            Add to T refinements of previously
            refuted hypotheses.
    until the conjecture T is neither too strong nor too weak
        with respect to the known facts.
    Output T.
forever.
```

As in Algorithm I, the tests in the while and repeat loops are, in general, undecidable. To implement them we choose some fixed complexity bound *h* and use is to bound the resources allocated to these tests. Another unspecified part in the algorithm is which refinements to add in the second while loop. One possible approach is to add them in a breadth-first order. That is, all hypotheses generated by two refinement will be added before those generated by three refinements, etc. . The main theorem proved in [12] is that using this approach, Algorithm 2

can identify in the limit any A-easy model, or, in other words, that it is as powerful as Algorithm I.

Another theoretical result obtained in [12], based on the work of the Blums*[2], says that under some constraints, this is as powerful as model inference algorithms can get. We say that a model inference algorithm is s*ufficient,* if whenever it examines a new fact, the last conjecture it has output implies all the observational sentences it already knows to be true. It can be shown that if a model inference algorithm is sufficient, then there exists a recursive function *h* such that A-casy models is all it can identify. Since Algorithm I is sufficient, these results establish that it is the most powerful of its kind.

5. *Relation to Other Work on Machine Lemming.*

The approach to inductive inference in logic presented here follows the direction set by Gold [3], which attempted to formulate problems of machine learning in a precise way, and to devise valid criteria for success of solutions to such problems. It is hoped that the model inference problem provides a natural setting for the continuing AI work on machine learning. The framework proposed here makes the results and theoretical tools developed in the recursion- and complexity-theoretic research in inductive inference applicable to the more concrete and experimental work in AI, and provides a solid basis for further development. It should be emphasized that the model inference algorithm described here is only one possible approach to inductive inference in logic, and other approaches to machine learning may use this theoretical framework with equal success.

The algorithm described here is most similar to the model-directed, top-down approach of the Version Spaces algorithm of Mitchell [9]. Both the Version Spaces algorithm and Algorithm 2 converge by finding some hypotheses that "match" the data, although the notions of "matching" used by the two are quite different: In the Version Spaces algorithm the pattern should match the instances. In algorithm 2 the hypotheses should agree with the facts. The question whether a pattern matches an instance is always decidable, and usually by a fast algorithm; on the other hand the corresponding question of whether a theory agrees with a fact may be undecidable, and in such a case can only be approximated by some resource-bounded computation.

In most of the recent work on program synthesis from examples the target programming language is Lisp [1,4,6,14]. Several approaches were used; Smith [13] provides a good survey of them. We have compared the performance of the Model Inference System, restricted to infer list-processing logic programs, to the works of Summers[14] and Biermann [I]. We summarize briefly the results of this comparison.

Most of the example Lisp programs synthesized by Summers* system THESYS[14] have equivalent logic programs which are context-free transformations. Some of the more complex functions, can be axiomatized using term-free transformations with auxiliary predicate. An example of one is pack, a program that packs a list of lists into one list.

packUMD-
*P*ek(M* Y],Z> - pack(Y,W) & append<X,W,Z)

Using the appropriate refinement operator, the Model Inference System inferred most of the examples described in his thesis, in less than one minute of CPU time. For example, it has inferred the program for pack above in 9 CPU seconds and from 25 facts, most of them negative. Summers does not give statistical information on the performance of his system, but it seems that the number of positive facts needed by the systems is comparable. THESYS does not need negative facts.

Biermann's system for the synthesis of regular Lisp programs from examples [I] is strongly influenced by Summers' method, although it has an enumerative component which Summers' system does not. Biermann gives a structural definition of the class of programs synthesized by his algorithm, and provides more information on the performance of his system. The simpler examples described in his paper can also be axiomatized by context-free transformations. For the more complex examples, context-free transformations with an auxiliary predicate suffice. This class is strictly contained in the class of term-free transformations used to infer some of Summers* examples.

As to the performance of the two systems in this domain, most of programs were synthesized by Biermann*s system from one example. The Model Inference System needed anywhere between 6 and 25 facts. Biermann's system needed between a fraction of a second to half an hour for these examples. The time taken by the Model Inference System on the same examples ranged between 2 and 38 seconds. The systems behaved similarly on the examples: what is harder for Biermann's system is also harder for the Model Inference System. The program that took Biermann's system half an hour to synthesize collects all first elements in a list of Lisp-atoms and lists. The Model Inference System synthesized the program following program for this task from 26 facts and in 38 seconds.

hoads([],U) -
heads(((X]Y]|Z].[X]WJ) - heads(Z.W)
haads([XjY),Z) - atom(X) & heads(Y.Z)

The actual timing figures are not very informative; what should be noted is the major difference in the growth rate. The systems' behavior suggest that the asymptotic time complexity of the Model Inference System compares favorably with Biermann's for this class of functions.

The Model Inference System has synthesized several programs that, as far as I know, have not been synthesized from examples before. Among them are programs for exponentiation, binary tree isomorphism and satisfiability of boolean formulas.

The most important difference between the Lisp systems and the Model Inference System is that the former usually incorporate some hard-wired synthesis algorithm, which can synthesize only a fixed class of functions. Generalizing such an algorithm is not a trivial task, as the work of Kodratoff [6] on generalizing Summers' method shows. The Model Inference System, on the other hand, incorporates the refinement operator as a parameter. To illustrate the flexibility of this approach, note that one refinement operator is sufficient for synthesizing almost all the examples of Summers. To get a more efficient inference of the restricted class of functions inferred by Biermann, a more

specific refinement operator was designed. The implementation of the new refinement operator required about 10 minutes of thought and rewriting *five* lines of Prolog code.

6. Concluding Remarks.

This paper has presented a general, incremental algorithm that infers theories from facts. Its theoretical analysis shows that it is comparable to some of the most powerful algorithms known from the complexity-theoretic approach to inductive inference. Its implementation is comparable to existing systems for inductive inference and program synthesis from examples. 1 believe that these results were made possible by the use of first order logic as the underlying model of computation.

Here are some of the reasons for the success of logic as a medium for inductive inference:

Logic has natural semantics. If a Turing Machine computes an incorrect result on a certain input, there is no sense in which one of the transitions in its finite control is "wrong" For every such candidate to be a "wrong" transition, one can always patch the Turing Machine without changing this transition, so it will behave correctly on this input. On the other hand, if a set of logical axioms has a false conclusion, there is a natural sense in which at least one of the axioms is *strictly false.* This fact enables the existence of error detecting algorithms such as contradiction backtracing.

Logic has an intimate relation between its syntax and semantics. This is the reason why there are natural ways to weaken the logical (computational) power of a refuted hypothesis, or, in other words, why natural and easy-to-compute refinement operators exist.

Logic is monotonic and modular. Altering an axiomatization by adding or removing axioms has clear effects on the expressive (computational) power of this axiomatization. There are not many practical programming languages for which such syntactic alterations to a program have predictable effects on what it computes.

Logic is a programming language that separates logic and control. It seems that one of the reasons for the efficiency of the Model Inference System is that it infers only the "logic component" of a program and leaves the "control component" unspecified [8]. The logic component of a program contains more than its specification, and the task of imposing control on a logic program is similar to the task of program optimization. The problems of program optimization and program synthesis from examples are hard enough by themselves to justify refraining from solving them simultaneously. We propose separating the task of synthesizing efficient programs from examples to two sub-tasks: inference of (sometimes inefficient) programs from examples, and program optimization.

Acknowledgements.

Dana Angluin and Drew McDermott supervised this work. I also thank Ernie Davis, John Ellis, Bob Kowalski, Bob Nix, Chris Riesbeck, David Warren and Steve Wood for their various contributions.

References

[I] Alan W. Biermann.
The Inference of Regular Lisp Programs from Examples.
IEEE Transactions on Systems, Man. and Cybernetics 8, August, 1978.

[2] Lenore Blum and Manuel Blum.
Towards a Mathematical Theory of Inductive Inference.
Information and Control 28, 1975.

[3] E. M. Gold.
Language identification in the limit.
Information and Control 10:447-474, 1967.

[4] Green C. C. et al.
Progress Reprt on Program Understanding Systems.
Technical Report Stan-CS-74-444, Computer Science Department, Stanford University, 1974.

[5] C. Cordell Green.
Theorem Proving by Resolution as a Basis for Question Answering.
In B. Meltzerand D. Mi chie, editor, *Machine Intelligence 4,* pages 183-205. Edinburgh University Press, Edinburgh, 1969.

[6] Yves Kodratoff.
A Class of Functions Synthesized from a Finite Number of Examples and a Lisp Program Scheme.
International Journal of Computer and Information Science 8(6):489-52l, 1979.

[7] Robert A. Kowalski.
Logic for Problem Solving.
Elsevier North Holland Inc., 1979.

[8] Robert A. Kowalski.
Algorithm = Logic + Control.
C ACM 22, July, 1979.

[9] Tom Michael Mitchell.
Version Spaces: An Approach to Concept Learning.
Technical Report STAN-CS-78-711, Stanford Artificial Intelligence Laboratory, December, 1978.

[10] L. Pereira, F. Pereira and D. Warren.
User's Guide to DECsystem-10 PROLOG.
Technical Report 03/13/5570, Labortorio Nacional De Engenharia Civil, Lisbon, September, 1978.

[II] Karl R. Popper.
The Logic of Scientific Discovery.
Basic Books. New York, 1959.

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

[13] Douglas R. Smith.
A Survey of Synthesis of LISP Programs from Examples.
In *International Workshop on Program Conduction, Chateo de Bonas.* INRIA, 1980.

[14] Philip Dale Summers.
Program Construction from Examples.
PhD Thesis, Yale University, 1976.
Computer Science Dept. research report No. 51.