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

Inductive Inference of Theories From Facts

February, 1981

Research Report 192

Ehud Y. Shapiro

Table of Contents

1. Introduction. 2
  1.1. Some Model Inference Problems. 2
  1.2. Relation to Problems of Scientific Discovery. 5
  1.3. Overview of the Paper. 6
2. Defining the Problem. 8
  2.1. Model Inference Problems and Algorithms. 8
  2.2. An Admissibility Requirement. 9
  2.3. Incorporating Theoretical Terms. 10
3. The Complexity of Model Inference Problems. 11
  3.1. An Enumerative Algorithm. 11
  3.2. A Bound on the Power of Sufficient Algorithms. 13
4. Refuting Hypotheses with Crucial Experiments. 14
  4.1. The Contradiction Backtracing Algorithm. 14
  4.2. On The Refutability of Scientific Hypotheses. 18
  4.3. An Application: Debugging Logic Programs. 18
5. Refining Refuted Hypotheses. 23
  5.1. Refinement Operators. 23
  5.2. Refinement Graphs. 26
  5.3. A Most General Refinement Operator. 27
6. A General, Incremental Model Inference Algorithm. 31
  6.1. Approximating the Source Set of a Model. 31
  6.2. An Incremental Algorithm. 32
  6.3. Some Implementation Issues. 35
7. Concluding Remarks. 36
Acknowledgements. 37
I. Performance of the Model Inference System 38
I.1 Inferring Regular Sets. 38
I.2 Inferring Atomic Complete Axiomatizations of Simple Models. 41
II. Glossary of Symbols. 46
III. A Useless Lemma. 47

List of Algorithms

Algorithm 1: An Enumerative Model Inference Algorithm. 11
Algorithm 2: Backtracing Contradiction with Crucial Experiments. 16
Algorithm 3: An Incremental Model Inference Algorithm. 32

List of Figures

Figure 1-1: Inferring Arithmetic. 2
Figure 1-2: Inferring A Formal Language. 3
Figure 1-3: Inferring Logic Programs. 4
Figure 4-1: Backtracing Contradictions in Propositional Logic. 14
Figure 4-2: Backtracing Contradictions in Predicate Logic. 16
Figure 4-3: A Prolog Program for Backtracing Contradictions. 19
Figure 6-1: Layout for an Incremental Model Inference Algorithm. 31

Inductive Inference of Theories From Facts

February 1981

Research Report 192

Ehud Y. Shapiro

Yale University
Department of Computer Science

Abstract.

This paper is concerned with model inference problems and algorithms. A *model inference problem* is an abstraction of the problem faced by a scientist, working in some domain under some fixed conceptual framework, performing experiments and trying to find a theory capable of explaining their results. In this abstraction the domain of inquiry is the domain of some unknown model M for a given first order language L, experiments are tests of the truth of sentences of L in M, and the goal is to find a set of true hypotheses that imply all true testable sentences.

The main result of this paper is a general, incremental algorithm for solving model inference problems, which is based on the Popperian methodology of conjectures and refutations [Popper 59, Popper 68]. The algorithm can be shown to identify in the limit [Gold 67] any model in a family of complexity classes of models, it is the most powerful of its kind, and is flexible enough to have been successfully implemented for several concrete domains.

This model inference algorithm has two tunable parameters: one determines how complicated the structure of hypotheses is; the other, how complex derivations from the hypotheses can be. Together they determine the class of models that can be inductively inferred in the limit by the algorithm. On the one hand they can be set so that the model inference algorithm can identify in the limit any model with complexity bounded by any fixed recursive function. On the other hand they can be set so that the algorithm, appropriately implemented, can infer axiomatizations of concrete models from a small number of facts in a practical amount of time. The performance of the *Model Inference System* demonstrates this.

The Model Inference System is based on this model inference algorithm, specialized to infer theories in Horn form. It has been implemented in the programming language Prolog [Pereira et al. 78]. As an example, in the domain of arithmetic, the system inferred the set of axioms described in Figure 1-1 below from 36 facts in 27 seconds CPU time. The system has discovered an axiomatization for dense partial order with end points. It has successfully synthesized logic programs [Kowalski 79a] for simple list-processing tasks such as append, reverse and most of the examples described in Summers' thesis [Summers 76]. It has also synthesized logic programs for satisfiability of boolean formulas, binary tree inclusion, binary tree isomorphism and others.

As part of the general algorithm, an algorithm for backtracing contradictions was discovered. This algorithm is applicable whenever a contradiction occurs between some conjectured theory and the facts. By testing a finite number of ground atoms for their truth in the model the algorithm can trace back a source for this contradiction, namely a false hypothesis, and can demonstrate its falsity by constructing a counterexample to it. The existence of such an algorithm seems to be relevant to the philosophical discussion on the refutability of scientific theories [Harding 76], and specialized to Horn theories may be a practical aid for the debugging of logic programs.

1. Introduction.

This paper is concerned with the following type of problem:

> *Suppose we can acquire factual knowledge about a certain domain, which is governed by some unknown rules, or theory, and assume that a language adequate for expressing these rules is given. How can we discover these rules or theory from this factual knowledge?*

A problem like this is faced by a scientist that is working in some domain under a fixed conceptual framework, performing experiments and trying to find a theory true of the domain and capable of explaining their results. A *Model Inference Problem* is an abstraction of this setting. In this section we explain the model inference problem via examples. These examples suggest possible applications for algorithms that can solve model inference problems. We discuss the relation of this work to problems of scientific discovery, and give an overview of the paper.

1.1. Some Model Inference Problems.

An example of a model inference problem is illustrated in Figure 1-1. 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 Z and times(X,Y,Z) for X times Y is Z. Assume we can test whether these relations hold between concrete numbers, i.e. , we can test ground (variable-free) atoms such as 0<0''', plus(0',0',0'') and times(0'',0'',0'''') for their truth in *M*. In this setting, the model inference problem is to find a finite set of sentences that are true of arithmetic and imply all true ground atoms. Figure 1-1 shows such a set of sentences. We use the back arrow ← to stand for "is implied by". The sentence *P* ← *Q* & *R* is read "*P* is implied by the conjunction of *Q* and *R*".

Figure 1-1: 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 of the functions and predicates involved to solve this model inference problem. In particular, the above set of axioms does not contain axioms for associativity of addition, transitivity of the ordering relation, etc.. If *T* is a set of sentences true in *M* that implies all ground atoms of *L* true in *M* then *T* is called an *atomic-complete axiomatization* of *M*. The set of sentences in Figure 1-1 is an atomic complete axiomatization of arithmetic. It has been inferred by the Model Inference System from 36 facts in 27 CPU seconds (see Appendix I, page 41).

We distinguish two types of sentences in the first order language L: *observational sentences*, which correspond to descriptions of experimental results, and *hypotheses*, which can serve as explanations for these results. The scientist's *domain of inquiry* is the domain of some unknown model M of L. *Experiments* performed in this domain are tests of observational sentences for their truth in M. *Facts* are statements of the results of such experiments. A model inference problem corresponds to the problem of scientific discovery:

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

It should be noted that the assumption that the hypothesis language is fixed avoids of one of the major problems in scientific discovery, which is the invention of a new conceptual scheme. In Kuhnian terminology [Kuhn 70], the model inference problem resembles more the puzzle-solving activity of "normal science", than the search for new conceptual schemes that is characteristic of "paradigm shift" periods.

Another model inference problem is illustrated in Figure 1-2. In this example the domain of inquiry is the set of binary strings, and the first order language contains one unary predicate in(X), two successor functions 0(X) and 1(X) and one constant Λ that denotes the null string. In this language the term (1(0(0(1(Λ))))) denotes the string 1001, and 1(0(X)) denotes a string whose prefix is 10. Parentheses are usually omitted. We assume the ability to test whether concrete strings are in some unknown set, and our goal is to find a finite set of true sentences that imply in(S) for every string S in this set and ~in(S) for any other binary string.

Figure 1-2: Inferring A Formal Language.

The Domain: Binary Strings.

The Language:
Λ - the null string
0(X) - concatenating 0 to X
1(X) - concatenating 1 to X
in(X) - X has an even number of 0's and an even number of 1's

Examples of facts:
in(0011) is *true*
in(011) is *false*
in(01011) is *false*
in(010100) is *true*

Theory:
```
in(Λ)
in(00X) ← in(X)
~in(0X) ← in(X)
in(11X) ← in(X)
~in(01X) ← in(X)
~in(1X) ← in(X)
~in(11X) ← in(0X) & ~in(X)
~in(01X) ← in(0X) & ~in(X)
~in(1X) ← in(0X) & ~in(X)
~in(10X) ← in(1X) & ~in(0X) & ~in(X)
~in(00X) ← in(1X) & ~in(0X) & ~in(X)
in(10X) ← ~in(1X) & ~in(0X) & ~in(X)
~in(00X) ← ~in(1X) & ~in(0X) & ~in(X)
in(01X) ← ~in(1X) & ~in(0X) & ~in(X)
```

If the unknown set is simple enough (to be more precise, if and only if it is regular [Angluin & Hoover 80]), then there is such a finite set of axioms in this language. If T is a set of sentences true in M such that for every ground atom $P \in L$, T implies P if P is true in M and $\sim P$ otherwise, then T is called a *ground-complete axiomatization* of M. Figure 1-2 shows a ground complete axiomatization of the set of strings with an even data, and suggested that the conjectural status of scientific theories be recognized. The validity of science can be based on the fact that if its theories are false then eventually they will be *refuted*, although in case they are true they can not, in general, be *confirmed* beyond any doubt.

Popper's approach to the Problem of Induction yields some procedural recommendations for scientific activity. Popper suggests that theoretical scientific activity should be aimed towards producing simple and easily refutable theories which account for the already known data, while the experimental activity should be aimed towards gathering new data with a potential for refuting these proposed theories. Popper claims that this interplay between the theoretical and experimental activity, between the conjectures and the refutations, might lead to some kind of convergence to the truth.

The concept of *verisimilitude* [Popper 68] was invented in an attempt to measure and compare this 'truth-likeness' of competing scientific theories. Informally, the verisimilitude of a set of hypotheses T_1 is greater or equal to the verisimilitude of T_2 if T_1 implies as many true observational sentences as T_2 and no more false observational sentences then T_2. One can systematically assign to theories numerical values that measure their verisimilitude. For convenience these values range between 0 and 1. Under such a measure, a true theory that implies all true observational sentences has verisimilitude 1, and both contradictory and tautologous theories have verisimilitude 0. The process of scientific discovery can be viewed as a everlasting attempt to increase the verisimilitude of its theories.

The concept of verisimilitude is useful for comparing competing *theories*. However, in order to evaluate and compare competing *methodologies of scientific discovery*, or *inductive inference algorithms*, it seems that the focus should be on the scientific discovery process, or algorithm, rather than on single theories. One way of evaluating and characterizing the power of inductive inference algorithms was suggested by Gold [Gold 65], and termed *identification in the limit*. The rationale behind this concept is very similar to the the Popperian one: a finite number of facts about a model can not in general determine it uniquely among all possible models, and since inductive inference algorithms and scientists always base their conjectures on a finite number of facts, both of them are bound to make mistakes. In some sense this is the crux of Popper's thesis. The most one can expect of an inductive inference algorithm is that after examining a finite number of facts about the model, and making a finite number of wrong conjectures, such an algorithm will correctly conjecture a finite set of hypotheses true in the model, which imply all true observational sentences. In such a case we say that the algorithm identifies the model in the limit. Note that such a set of hypotheses is of verisimilitude 1, and an inductive inference algorithm cannot, in general, determine whether it actually has found such a set of hypotheses. The notion of identification in the limit has proven to be fruitful in the recursion- and complexity-theoretic work on inductive inference [Gold 67, Blum & Blum 75, Case & Smith 81].

1.3. Overview of the Paper.

In Section 2 we define more precisely the notions of model inference problem and algorithm. Then, following the approach of [Blum & Blum 75], we examine the complexity of model inference problems. Section 3 defines the class of *h*-easy models, for some recursive function *h*, and describes an enumerative algorithm that can identify in the limit any *h*-easy model for a fixed *h*. It is shown that if an inductive inference algorithm satisfies a simple sufficiency requirement, namely that its current conjecture always accounts for the currently known facts, such an algorithm can identify in the limit only *h*-easy models. These results establish an upper bound on the power of any sufficient inductive inference algorithm, and show that this bound is realizable.

This general complexity theoretic approach is applicable to any computational model, however, and does not make use of any special properties of logic. Motivated by this analysis, we develop some logic-specific algorithms and concepts that can support the construction of more efficient, incremental, inference algorithms that take better advantage of the semantic and syntactic properties of logic. In developing the incremental additional control information should be specified. See [Kowalski 79a] for an introduction to logic programming, [Van Emden & Kowalski 76] for discussion of the operational and denotational semantics of logic programs, [Kowalski 79b] for an elaboration of the distinction between the logic component and the control component of an algorithm, and the Prolog manual [Pereira et al. 78] for details of a concrete implementation of a logic based programming language.

In our setting, if the hypothesis language is restricted to Horn clauses, then an atomic-complete axiomatization of a model is a logic program for computing the predicates in L. Figure 1-3 shows two such programs. In this example, the language L contains the two place function symbol [X|Y] (the Prolog list constructor, the equivalent of the LISP function cons), the constant [] (Prolog's nil) and the two predicates append(X,Y,Z) and reverse(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,b,c,d,e]); the atom reverse(X,Y) is true in M just in case that the list Y is the reverse of the list X, e.g. reverse([a,b,c],[c,b,a]). The Horn clauses in Figure 1-3 are an atomic-complete axiomatization of the model thus defined.

As its turns out, these clauses are also Prolog programs for computing append and reverse. For example, to compute the result of appending the lists [a,b] and [c,d,e], the goal ←append([a,b],[c,d,e],X) is given to the Prolog interpreter loaded with the above program, which returns the binding X = [a,b,c,d,e]. The Model Inference System synthesized the logic program for append in 11 CPU seconds from 34 facts, and a similar program for reverse from 13 facts in 6 CPU seconds (see page 42). More examples of logic programs are given throughout this paper, and the details of their synthesis from examples can be found in Appendix I.

Some of the work in Artificial Intelligence on concept-learning tasks can also be restated as model inference problems. For example, the problem of learning the concept of an arch [Winston 75] from descriptions of arches and non-arches can be restated as follows: the domain of inquiry is all objects built out of blocks. The language contains predicates like block(X), column(X), on(X,Y), arch(X,Y,Z). The inductive inference problem is to find a finite set of sentences that can decide which of these compound objects are arches, given examples of arches and non-arches. One such set of sentences was suggested by Kowalski [Kowalski 79a]:

```
    arch(X,Y,Z) ← block(X) & column(Y) & column(Z) & on(X,Y) & on(X,Z)

    column(X) ← block(X)
    column(stack(X,Y)) ← block(X) & column(Y) & on(X,Y)

    on(X,stack(Y,Z)) ← on(X,Y)
```

From this set of sentences, given that the following are true,

```
        block(a)   on(a,b)
        block(b)   on(b,c)
        block(c)   on(a,d)
        block(d)   on(d,e)
        block(e)   on(e,f)
        block(f)
```

one can prove that arch(a,stack(b,c),stack(d,stack(e,f))). In general, one can decide from these axioms whether a scene is an instance of the arch concept, and it seems reasonable to say that by acquiring such axioms one learns this concept.

1.2. Relation to Problems of Scientific Discovery.

Any interesting solution to a model inference problem must come to terms with a preliminary problem, that is the *Problem of Induction*, which goes back to the philosopher David Hume. The problem of induction is that a finite amount of factual data can never establish beyond doubt a theory with infinitely many consequences. An attractive solution to this problem was suggested by the philosopher Karl R. Popper [Popper 59]. Popper accepted Hume's claim of the unprovability of general scientific theories from factual number of 0's and an even number of 1's. This particular axiomatization was generated from the 4-state acceptor for this set by the algorithm of Angluin and Hoover. An older version of the Model Inference System inferred a more concise ground-complete axiomatization of this set in 3 minutes CPU time from 64 facts (page 38). The axioms discovered by the system contained only one atom in their condition, as opposed to two and three in some of the axioms above. This behavior of the algorithm suggested that any regular set has such a ground-complete axiomatization, a result that will be described elsewhere [Angluin & Shapiro 81].

The following is an atomic-complete axiomatization of the same regular set. Comparing it with the axiomatization in Figure 1-2 might help to clarify the distinction between an atomic-complete and a ground-complete axiomatization of a model. The Model Inference System came up with this axiomatization in 28 CPU seconds and after reading in 35 facts (page 40).

```
in(Λ)
in(00X) ← in(X)
in(11X) ← in(X)
in(010X) ← in(1X)
in(011X) ← in(0X)
in(100X) ← in(1X)
in(101X) ← in(0X)
```

The problem of inductively inferring regular sets was investigated by Gold and Angluin [Gold 67, Gold 78, Angluin 78], among others. It has served both as a motivation and a test case for the development of the model inference algorithm and system.

A third type of inductive inference problem that naturally fits in our framework is taken from the domain of automatic programming, and is usually termed *program synthesis from examples* [Green et al. 74, Summers 76, Summers 77, Bierman 78]. The task is to inductively infer a program, given examples of its input-output behavior. This task can be restated as a model inference problem, and in this case the programs to be inferred are *logic programs*.

Figure 1-3: 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],[a]) is true
                     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)
```

A logic program is a collection of Horn clauses, which are sentences of the form $P \leftarrow Q_1 \& Q_2 \& \ldots \& Q_n$ for $n \geq 0$, where P and the Q's are atoms. Such a sentence is read "P is implied by the conjunction of the Q's", and is interpreted procedurally "to satisfy goal P, satisfy goals $Q_1,Q_2,\ldots,Q_n$". A collection of Horn clauses can be executed as a program, using this procedural interpretation. Usually, for such a program to be efficient, model inference 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?

We say that a set of hypotheses 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.

The *Contradiction Backtracing Algorithm*, described in Section 4, attempts to answer the first question. By performing a finite sequence of experiments that test ground atoms for truth in the model, the algorithm can trace back a source of the contradiction between the conjecture and facts, and single out a false hypothesis.

Tests of the kind performed by this algorithm are known in philosophy of science as *crucial experiments*. Although their importance is recognized by most methodologies, an algorithmic way of sequencing them that *guarantees* singling out a false hypothesis is a novelty. The existence of such an algorithm apparently contradicts a claim of Duhem [Duhem 54], which simply denies its possibility. This claim was stated in support of what later came to be known as the Duhem-Quine thesis [Harding 76] of the irrefutability of scientific theories. The existence of the contradiction backtracing algorithm, and of a general inductive inference algorithm that incorporates it, may renew the philosophical discussion concerning this issue of refutability. The contradiction backtracing algorithm has applications beyond the domain of inductive inference; we propose an application of it to the debugging of logic programs.

Once we have discovered a false hypothesis in the conjecture, the obvious action to take is to remove this hypothesis from it. However, the resulting conjecture may be too weak to imply all the observational sentences already known to be true. In this context the second question arises. A *refinement operator*, described in Section 5, suggests how to add new hypotheses to the conjecture in order to strengthen it. The refinement operator is a parameter in the inference algorithm that can be tuned. A more general refinement operator results in a more powerful, though less efficient algorithm, and vice versa. The notion of completeness of a refinement operator for a class of hypotheses is defined, and several refinement operators, complete for different classes of hypotheses, are described. A most general refinement operator is defined and proved to be complete for any first order language.

Based on the solutions to these two questions, a general, incremental model inference algorithm is developed in Section 6, and proved to identify in the limit any h-easy model. Some implementations issues are also discussed. The concluding remarks to this paper reveal why all this stuff works, and Appendix I describes the performance of the Model Inference System, which implements the algorithm. This system will be discussed more fully in a subsequent paper.

2. Defining the Problem.

In this section we state more precisely the model inference problem, and define the notion of a model inference algorithm. We introduce an admissibility requirement on the observational and hypothesis languages, which reflects Popper's requirement that theories should be refutable by facts, and discuss an extension to the model inference problem which includes the notion of theoretical terms.

2.1. Model Inference Problems and Algorithms.

We assume that the given first order language L is in clausal form (as defined in [Robinson 65]), with finitely many predicate and function symbols. The sentences of L are of the form

$$\{P_1, P_2, ..., P_j\} \leftarrow \{Q_1, Q_2, ..., Q_k\} \quad j, k \geq 0$$

where the P's and Q's are atoms. This notation is equivalent to the standard clausal form notation where the P's are the positive literals and the negated Q's are the negative literals of the clause. The interpretation of such a sentence is that the conjunction of the Q's imply the disjunction of the P's. The Q's are called the *condition* of the sentence, and the P's the *conclusion* of it. An empty condition implies that *at least* one atom in the conclusion is true. An empty conclusion implies that *all* the atoms in the condition are false. $\square$ denotes the empty sentence, false in any model of L. Sentences with only one atom are called *unit sentences*. We do not distinguish between the unit sentence $\{P\}\leftarrow$ and the atom P, and between sentences that are equal up to renaming of variables. Note that as in standard clausal notation all the variables that occur in a sentence are implicitly universally quantified.

We distinguish two subsets of the sentences of L: the *observational language* L_o, and the *hypothesis language* L_h. We assume that $\square \in L_o \subset L_h \subset L$[1], and that both sets are effectively decidable. *Observational sentences* are sentences of L_o and *hypotheses* are sentences of L_h. The *domain of inquiry* is some model M of L, and we assume that there is some device, called an *oracle* for M, that when given an observational sentence α returns '*true*' if α is true in M, '*false*' otherwise. The operation of giving an input to the oracle and reading the answer is called an *experiment* in M. *Facts* about the domain M are pairs of the form $\langle\alpha,V\rangle$, describing results of experiments, where $\alpha \in L_o$ is an observational sentence and $V \in \{true, false\}$ is the truth value of α in M. The set of observational sentences true in a model M is denoted L_o^M. We assume some complete derivation procedure, and use $T \vdash p$ $\{T \nvdash p\}$ to denote that p is {is not} derivable from T. For a set of sentences S, $T \vdash S$ if $T \vdash p$ for every $p \in S$, $T \nvdash S$ otherwise.

Definition 2.1: Let L_o and L_h be subsets of L such that $\square \in L_o \subset L_h \subset L$, and M a model of L. A set of sentences $T \subset L_h$ is an *L_o-complete axiomatization of* M if and only if T is true in M and $T \vdash L_o^M$.

We can now define more precisely a *Model Inference Problem*:

Suppose we are given a first order language L and two subsets of it as defined above, an observational language L_o and a hypothesis language L_h. In addition, assume that we are given an oracle for some unknown model M of L. The Model Inference Problem is to find a finite L_o-complete axiomatization of M.

An algorithm for solving a model inference problem is called a *model inference algorithm*. We make more precise the notion of such an algorithm, adapting the definitions of Gold and the Blums [Gold 65, Blum & Blum 75]. An *enumeration of a model M* is an infinite sequence $F_1, F_2, F_3,...$ where each F_i is a fact about M and every sentence α of L_o occurs in a fact $F_i=\langle\alpha,V\rangle$ for some $i>0$. A *model inference algorithm* is an algorithm that reads an enumeration of a model for a given observational language L_o, one fact at a time, and

[1]We use L to denote both the set of non-logical symbols of the first order language and the set of sentences generated from these symbols. Also, throughout the paper, $A \subset B$ means that A is a subset of, or equal to, B, and $A \subsetneq B$ means that A is a strict subset of B.

once in a while produces as output a finite set of sentences of the hypothesis language L_h, called the *conjecture* of the algorithm.

A model inference algorithm is said to *converge in the limit* given an enumeration of a model if eventually the algorithm outputs some conjecture and never again outputs a different conjecture. A model inference algorithm is said to *identify a model M in the limit* if it converges on every enumeration of M to a conjecture which is an L_o-complete axiomatization of M. Given an enumeration of a model M one can simulate an oracle for M and vice versa, so we do not distinguish between these notions in the abstract treatment of the problem. However, as will be apparent in the following, the ability to make experiments may result in a considerable speedup in the inference process, so the concrete algorithms usually assume that an oracle for M is given.

2.2. An Admissibility Requirement.

In order for a model inference problem to be solvable by a model inference algorithm, an *admissibility requirement* must be made on L_o and L_h, which essentially says that L_o contains enough information to refute any false theory that 'attempts' to be L_o-complete. The admissibility requirement reflects Popper's methodological requirement that theories should be refutable by facts.

Definition 2.2: Let L_o be an observational language and and L_h a hypothesis language as above. We say that the pair $<L_o, L_h>$ is *admissible* if for every model M of L and every $T \subset L_h$, $\{\alpha \epsilon L_o | T \vdash \alpha\} = L_o^M$ implies that T is true in M.

By the admissibility requirement every false theory in L_h that implies L_o^M has a witness for its falsity in the observational language L_o. To see this, note that if $T \vdash L_o^M$ and T is false in M, then there is an $\alpha \epsilon L_o$ false in M such that $T \vdash \alpha$. To see that this property implies admissibility of L_o and L_h, assume that this property holds, and let $T \subset L_h$ be false in M. Then either $T \nvdash L_o^M$ or $T \vdash \alpha$ for some $\alpha \epsilon L_o$ false in M. Both cases satisfy the admissibility requirement, which implies the admissibility of L_o and L_h. The admissibility requirement couples the observational language and hypothesis language together, and implies that the difference in their expressive power should be bounded in such a way that every theory which is *successful*, i.e. implies all true observational sentences and no false ones, should also be *true*. From the admissibility requirement it follows that if L_o and L_h are admissible, then every L_o' and L_h' such that $L_o \subset L_o' \subset L_h' \subset L_h$ are also admissible.

There are two pairs of observational and hypothesis languages which are of interest to us. One is the ground atoms of L as the observational language, with the Horn sentences of L as the hypothesis language. For this pair an L_o-complete axiomatization of M is called an *atomic-complete* axiomatization. Another is the ground unit sentences of L as the observational language, with L as the hypothesis language. For this pair an L_o-complete axiomatization of M is called a *ground-complete* axiomatization. Before showing the admissibility of these pairs, we show that the pair <{atoms of L}, L> is inadmissible. A propositional example suffices. Assume that L contains three atoms P,Q and R. Choose a model M for L in which P and Q are false, R is true. Then $L_o^M = \{\leftarrow\{R\}\}$, and the set of consequences of $T = \{\{R\}\leftarrow, \{P,Q\}\leftarrow\}$ restricted to L_o is equal to L_o^M, but T is false in M. This example and the following two theorems give some insight into the intimate connection between Horn sentences and atomic-complete theories. See also [Van Emden & Kowalski 76] for a discussion of this issue.

Theorem 2.3: Let L be a first order language, L_o={ground atoms of L} and L_h={Horn sentences of L}. Then the pair $<L_o, L_h>$ is admissible.

Proof: Assume that $T \subset L_h$ is false in M and $T \vdash L_o^M$. We show that L_o has a witness for the falsity of T. Since T is false there is a sentence $p = \{P\} \leftarrow \{Q_1, Q_2, ..., Q_n\} \epsilon T$ false in M, which implies that p has a ground instance $p' = \{P'\} \leftarrow \{Q'_1, Q'_2, ..., Q'_n\}$ false in M (see definitions on page 15). Such a sentence is false only if $Q'_1, Q'_2, ..., Q'_n$ are true and P' is false. Since $T \vdash L_o^M$, it follows that $T \vdash Q'_i$ for $1 \leq i \leq n$. Since $p \epsilon T$ it follows that $T \vdash P'$, which is a false observational sentences. Therefore this pair is admissible. ■

Theorem 2.4: Let L be a first order language, L_o={ground unit sentences of L} and $L_h=L$. Then the pair $<L_o, L_h>$ is admissible.

Proof: The analogous proof holds. Assume that $T \subset L_h$ is false in M and $T \vdash L_o^M$. Since T is false there is a sentence $p=\{P_1, P_2,..., P_m\} \leftarrow \{Q_1, Q_2,..., Q_n\} \in T$ false in M, which implies that p has a ground instance $p'=\{P'_1, P_{(2},..., P)_m\} \leftarrow \{Q'_1, Q'_2,..., Q'_n\}$, false in Ma. Such a sentence is false only if $Q'_1, Q'_2,..., Q'_n$ are true and $P'_1, P'_2,..., P'_m$ are false. Since $T \vdash L_o^M$, $T \vdash \{Q'_i\} \leftarrow$ for $1 \leq i \leq n$ and $T \vdash \leftarrow \{P'_i\}$ for $1 \leq i \leq m$. Since $p \in T$ it follows that T is inconsistent, hence it implies every sentence of L, in particular $T \vdash \Box$, which is a false observational sentence. ■

2.3. Incorporating Theoretical Terms.

The admissibility requirement is a necessary, not a sufficient condition for the solvability of a model inference problem. For example, if our domain is the standard model of arithmetic, the first order language L is the language of arithmetic as above and we choose $L=L_h=L_o$, then the pair $<L_o, L_h>$ is admissible, but Gödel's second incompleteness theorem shows that there is no finite consistent set of axioms in L that implies every sentence of L true of arithmetic.

In the case of arithmetic, enriching the hypothesis language L would not make this model inference problem solvable; there are cases, however, in which the unsolvability of the problem is the result of the weakness of the expressive power of the hypothesis language. For example, assume that L is the language of binary strings described in Figure 1-2 (page 3), with one unary predicate symbol in(X), and that the unknown set to be inferred is all palindromes over 0 and 1. A string S is a *palindrome* if it is the reversal of itself. Since the set of palindromes is not regular, the result of Angluin and Hoover [Angluin & Hoover 80] shows that there is no finite set of sentences in this language that is true of palindromes and imply in(S) for every palindrome S.

Intuitively speaking, to understand the concept of a palindrome one must know first the concept of string reversal, and without having this concept or being able to simulate this operation in some way, a person would never succeed in learning the concept of a palindrome just from examples of palindromes and non-palindromes. Terms that denote such concepts, which serve as an aid to the construction of an hypothesis but are not directly observable in the world are called in the literature of philosophy of science *theoretical terms*. If we include in the hypothesis language L_h the predicates append and reverse as theoretical terms, then the task of inferring of an atomic-complete axiomatization of palindromes becomes not just solvable, but easy, since the following axiom suffices

```
in(X) ← reverse(X,X)
```

and reverse can be axiomatized using append, as shown in Figure 1-3 (page 4).

However, if all we see in the world are strings that are in and out of some unknown set, there is no way for us to infer from examples the corresponding axiomatizations of append and reverse. In such a setting, the least we have to assume is that the intended interpretations of append and reverse are known to, or built into, the inference algorithm as "theoretical concepts". The way to incorporate the notion of theoretical predicates into the model inference problem is to require them to have some fixed interpretation, known to the inference algorithm, and assume that this interpretation holds in the unknown model M. We will not formally develop this idea here, though a reference to it is made in the discussion of the refutability of scientific hypotheses (Section 4.2). This concludes the presentation of the model inference problem.

3. The Complexity of Model Inference Problems.

Two inductive inference problems closely related to the model inference problem have been given a considerable amount of attention in the literature: one is the problem of inductively inferring a program for enumerating a recursively enumerable set, given information on whether elements are in or out of that set [Gold 65, Gold 67, Klette and Weihagen 80]; another is the problem of inductively inferring a program for computing a recursive function, given the values of that function [Gold 65, Blum & Blum 75]. Gold [Gold 65] has shown that there is no general effective procedure that solves these problems for all recursively enumerable sets or all recursive functions. The Blums [Blum & Blum 75] describe algorithms that solve the latter problem, given that the functions to be identified are easy to compute under some complexity measure. They also show that under certain assumptions, the algorithms they describe establish an upper bound on the power of any inference algorithm.

The results of the Blums suggest that the natural classes of objects to be inductively inferred in the limit are complexity classes. In this section we obtain a similar result for the model inference problem. The class of *h-easy models* with respect to some recursive function h is defined, and a simple, though inefficient algorithm that can infer in the limit any h-easy model is described. It is shown that if an inductive inference algorithm satisfies a simple *sufficiency requirement*, namely that its current conjecture always accounts for the currently known facts, then the algorithm can identify only h-easy models, for some recursive function h. This establishes an upper bound on the power of any such inductive inference algorithm.

The enumerative inference algorithm described in this section does not take full advantage of the semantic and syntactic properties of first order logic, and a similar enumerative algorithm can be implemented within almost any other computational model. Methods for speeding up the inference procedure that take advantage of properties of logic are described in the following sections.

3.1. An Enumerative Algorithm.

We assume some admissible pair $<L_o,L_h>$. Let $\alpha_1,\alpha_2,\alpha_3,\ldots$ be a fixed effective enumeration of all sentences of L_o, $T_1,T_2,T_3,\ldots$ a fixed effective enumeration of all finite sets of sentences of L_h, and M a model for L. We use $T\vdash_{\overline{n}}\alpha$ $\{T\nvdash_{\overline{n}}\alpha\}$ to denote that T can {cannot} derive α in n derivation steps or less, and assume that for any finite set of sentences $T\subset L$ and any n>0, the set of sentences derivable from T in n derivation steps is computable and finite. We also assume that the derivation procedure is monotonic, that is, if $T\vdash_{\overline{n}}p$ then $T\cup\{q\}\vdash p$, for any p, q and T. One example of such a proof procedure is *resolution* [Robinson 65]. In the following more concrete discussion we assume that resolution is our proof procedure. This choice is immaterial, however, to the more abstract results of this section.

Algorithm 1 implements the following simple idea: hold to your conjecture as long as it you think it agrees with the known facts. Once you have discovered that it does not, search by enumeration for the next conjecture which you think does. Since the check whether a conjecture agrees with a fact is in general undecidable, choose *a priori* some fixed bound on the time you are going to spend performing this check on any given fact.

In order to characterize the set of models Algorithm 1 can infer in the limit we need some complexity-theoretic tools and definitions; see [Machtey & Young 78] for a more complete exposition of these. We associate with each set of axioms T_k a step counting function Φ_k, where $\Phi_k(i) = \min\{n \mid T_k\vdash_{\overline{n}}\alpha_i\}$. This function is a partial recursive function, and the reader can verify that the set of step counting functions $\Phi_1,\Phi_2,\Phi_3,\ldots$ constitutes a complexity measure over the set of characteristic functions $\varphi_1,\varphi_2,\varphi_3,\ldots$ defined by

$$\varphi_k(i) = \begin{cases} 1 \text{ if } T_k\vdash\alpha_i \\ \\ \text{undefined otherwise.}\end{cases}$$

A finite set of sentences T is called *h-easy* if its L_o consequences are easy to derive modulo a total recursive

4. Refuting Hypotheses with Crucial Experiments.

One major source of the infeasibility of Algorithm 1 is its global nature. Whenever it finds that a set of sentences is not an axiomatization of the model it simply discards it and searches through all finite sets of sentences until it finds the next plausible conjecture. In this section we address the first problem that must be solved to develop an incremental inference algorithm. That is, what to do in case the current conjecture is discovered to be *too strong*, i.e., implies a false observational sentence. In such a case one can conclude that at least one of the hypotheses is false. An algorithm is developed that can detect a false hypotheses in a conjecture with false observational conclusions. This algorithm is called the *contradiction backtracing algorithm*, since it can trace a contradiction between a conjecture and the facts back to its source, which is a false hypothesis. The relevance of this algorithm to the philosophical question of refutability of scientific theories is discussed, and its application to debugging logic programs is illustrated.

4.1. The Contradiction Backtracing Algorithm.

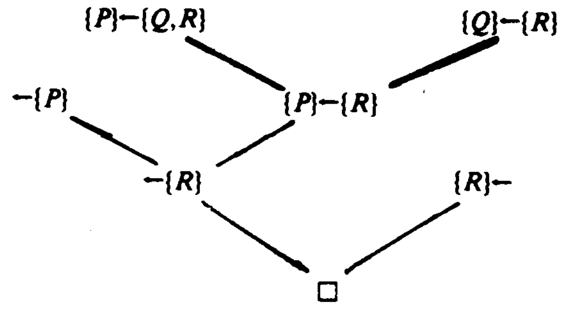
Crucial experiments are experiments that have a potential to decide between competing scientific theories. A successful crucial experiment can eliminate at least one of the competing theories by providing a counterexample to one of its predictions. The important role of crucial experiments in scientific progress is recognized by most schools of philosophy of science. However, there were strong arguments for their limited power as well. As the philosopher Pierre Duhem said, "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" [Duhem 54]. Even Popper, the major advocate of refutations as the vehicle for progress in science says "...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" [Popper 68].

The contradiction backtracing algorithm overcomes, in some sense, this limitation. Although one crucial experiment can, in general, refute only a *collection* of hypotheses, the contradiction backtracing algorithm suggest a way of sequencing crucial experiments, which guarantees singling out a *unique* false hypothesis. This method tells us exactly where the error lies, and does not involve any guesswork. The sequencing method is dynamic, that is, the sequence of experiments to be performed can not determined *a priori*; rather, the outcome of every new experiment suggests its successor, the last of which unambiguously points to a false hypothesis. In addition, the collected results of these experiments provide a *counterexample* to the refuted hypothesis, that is, a sentence which logically follows from the hypothesis and was experimentally determined to be false.

The contradiction backtracing 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 □ from a set of hypotheses and true observational sentences S. 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 $p\epsilon S$ which is false in M.

We first demonstrate what the algorithm does on a propositional calculus example, illustrated in Figure 4-1. In this example S={{P}←{Q,R}, {Q}←{R}, ←{P}, {R}←}.

Figure 4-1: Backtracing Contradictions in Propositional Logic.



the nth fact the condition of the *while* loop will be satisfied, and the value of k will be increased. Hence after examining a finite number of facts Algorithm 1 will discard all of the conjectures T_k for $k<k0$. On the other hand, if the algorithm ever conjectures T_{k0} it will never again output a different conjecture, since T_{k0} is true in M and derives every α_i true in M in $h(i)$ steps. Therefore after examining a finite number of facts the algorithm will conjecture T_{k0}, and never again output a different conjecture. ■

3.2. A Bound on the Power of Sufficient Algorithms.

Algorithm 1 shows that for any recursive function h there is an inductive inference algorithm that can identify in the limit every h-easy model. In this section we show that if an inductive inference algorithm is *sufficient* it can identify in the limit only h-easy models, for some fixed recursive function h. By this we establish an upper bound on the power of any such algorithm.

Definition 3.3: A set of sentences T is said to be *sufficient for the facts* $F_1, F_2, \ldots, F_n$ if for every fact $F_i=\langle\alpha, true\rangle$, $1\leq i\leq n$, T implies α. An inductive inference algorithm is said to be *sufficient* if whenever it is applied to an enumeration $(F_1, F_2, \ldots)$ of some model M and reads the the (n+1)th fact, for some $n\geq 0$, the last conjecture the algorithm outputs is sufficient for $F_1, F_2, \ldots, F_n$.

Algorithm 1 described above is sufficient. This property seems to be a reasonable requirement of an inference algorithm, although the Blums' paper describes some powerful inference algorithms that do not satisfy it. The next theorem shows that if we require an inference algorithm to be sufficient, then h-easy models are all it can infer.

Theorem 3.4: Let I be a sufficient inductive inference algorithm for a language L. Then there is a total recursive function h uniform in I such that if I can identify a model M of L then M is h-easy.

Proof: By the following Lemma 3.5 there is a sufficient algorithm I' uniform in I, such that I' is as powerful as I, and will eventually read in all the facts.

Let S(n) be the set of all consistent finite sequences of facts of the form $\sigma=(\langle\alpha_1, V_1\rangle, \langle\alpha_2, V_2\rangle, \ldots, \langle\alpha_n, V_n\rangle)$ that do not include the empty sentence. For every $\sigma\epsilon S(n)$ define $I'[\sigma]$ to be the index of the last conjecture algorithm I' outputs after reading the facts in σ in that order, and before reading the (n+1)th fact. $I'[\sigma]$ is well defined and computable since I' eventually reads in all the facts.

Since I' is sufficient, for every $\sigma\epsilon S(n)$ and for every fact $F_i=\langle\alpha_i, true\rangle$ in σ, if $k=I'[\sigma]$ then $T_k\vdash\alpha_i$ in some finite number of steps, and therefore $\Phi_k(n)$ is defined and finite for all i such that $V_i=true$, $0\leq i\leq n$. We choose the complexity bound h to be

$$h(n) = \max\{\Phi_k(n) \mid k=I'[\sigma], \sigma\epsilon S(n), V_n=true\}$$

The sequences in S(n) do not contain the empty sentence, therefore for every n there is a $\sigma\epsilon S(n)$ for which $V_n=true$, hence h is total and recursive. We now show that every model that I' can infer in the limit is h-easy. Let M be such a model, and apply I' to its enumeration $\langle\alpha_1, V_1\rangle, \langle\alpha_2, V_2\rangle, \ldots$. Since I' can identify M, after reading n facts for a sufficiently large n it will converge to some L_0-complete axiomatization T_i of M. By the definition of h it follows that for a sufficiently large n, $\Phi_i(n)\leq h(n)$, and therefore M is h-easy. ■

Lemma 3.5: Let I be a sufficient inductive inference algorithm. Then there is a sufficient inductive inference algorithm I' uniform in I such that I' is as powerful as I and eventually reads in all the facts.

Proof: The lemma follows from Minicozzi's union theorem along the same line of argument as in the proof of Theorem 3 in [Blum & Blum 75]. ■

This completes the complexity analysis of the model inference problem.

Algorithm 1: An Enumerative Model Inference Algorithm.

```
Let $h$ be a total recursive function.
set $S_{false}$ to $\{\Box\}$, $S_{true}$ to $\{\}$ and k to 0.
repeat
    read the next fact $F_n = <\alpha, V>$.
    add $\alpha$ to $S_V$.
    while there is an $\alpha \in S_{false}$ such that $T_k \vdash_n \alpha$
        or there is an $\alpha_i \in S_{true}$ such that $T_k \nvdash_{h(i)} \alpha_i$ do
        set k to k+1.
    output $T_k$.
forever.
```

function h, that is $\Phi_k(n) < h(n)$ for almost every (that is, for all except a finite number of) n>0 such that $T \vdash \alpha_n$. A model M is *h-easy* if there is a k>0 such that T_k is an h-easy L_o-complete axiomatization of M. If M is h-easy then Algorithm 1 can infer M in the limit, that is after examining some finite number of facts, the current conjecture T_k of the algorithm is an L_o-complete axiomatization of the model, and subsequently this conjecture does not change. In other words,

Theorem 3.1: Let h be a total recursive function and M an h-easy model of L. Then Algorithm 1 identifies M in the limit.

In order to prove the theorem, we need the following lemma, which says that if a model has a finite L_o-complete axiomatization that behaves badly (complexity-wise) on a finite number of L_o sentences then there exists another finite L_o-complete axiomatization of M that does better on all these anomalous cases, and does at least as well on all the others.

Lemma 3.2: Let M be an h-easy model for some total recursive function h. Then there is a k such that

1. T_k is true in M.

2. $\Phi_k(n) < h(n)$ for all n>0 such that α_n is true in M.

Proof: Since M is h-easy there is a k1 such that T_{k1} is a finite L_o-complete axiomatization of M, and $\Phi_{k1}(n) < h(n)$ for almost every n>0. To find a k that satisfies both conditions patch T_{k1} by adding to it the sentences α_i for every i>0 such that α_i is true in M and $\Phi_{k1}(i) > h(i)$.

The finite set of sentences thus defined is in L_h, since $L_o \subset L_h$. Let k2 be the index of this set of sentences. Since T_{k1} is an L_o-complete axiomatization of M, and T_{k2} was obtained from T_{k1} by adding only sentences true in M, then T_{k2} is also an L_o-complete axiomatization of M. We know that $\Phi_{k2}(n) < \Phi_{k1}(n)$ for all n, since $T_{k1} \subset T_{k2}$. Furthermore, for every i such that $\Phi_{k1}(i) > h(i)$, $\Phi_{k2}(1) = 1$, since the corresponding sentence is in T_{k2}. Therefore k2 satisfies both conditions. ■

Proof of Theorem 3.1: Let k0 be the smallest k such that

1. T_k is true in M.

2. $\Phi_k(n) < h(n)$ for all n>0 for which α_n is true in M.

Such a k exists by the Lemma 3.2.

Any T_k such that k<k0 does not satisfy at least one of these conditions. If T_k does not satisfy condition 2 then there is an n>0 such that $T_k \nvdash_{h(n)} \alpha_n$ for some α_n true in M. If T_k satisfies condition 2 but not condition 1 then by the admissibility requirement on L_h and L_o there is some α_{n1} false in M such that $T_k \vdash \alpha_{n1}$. Choose n to be the minimal n>n1 such that $T_k \vdash_n \alpha_{n1}$. In either case after reading

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 guaranteed to be 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.

Note that the answer to one experiment determines the next experiment. Also, the number of experiments is exactly the depth of the leaf reached, and the combined results of these experiments constitutes a counterexample to the hypothesis in that leaf. If some of the leaves contain true observational sentences (rather than hypotheses), the number of experiments needed may be smaller.

In the propositional case one can detect a false hypothesis simply by testing all the atoms that appear in the leaves for their truth in M. This simpleminded procedure might increase the number of experiments needed by an exponential factor, but in principle it can perform the same task. In the predicate calculus case the task is more difficult. A universally quantified hypothesis can not be effectively tested over an infinite domain, but if it is false then it has a false ground instance. The simpleminded procedure in the predicate calculus case is to systematically instantiate the hypotheses over increasing portions of the domain, and test all the ground instances generated (assuming they are all in L_o) until a counterexample to an hypothesis, i.e. a false instance of it, is discovered. This algorithm is guaranteed to halt since at least one of the hypotheses in the leaves is false, and such an hypothesis has a false instance that is a result of instantiating it over a finite domain. This approach is, however, clearly infeasible.

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. However the way this counterexample is constructed is slightly more involved. Before describing the algorithm some terminology concerning resolution needs to be made more precise. We follow Robinson [Robinson 65] in these matters. A *substitution* is a finite set (possibly empty) of pairs of the form V/t where V is a variable and t is a term, none of these variables are the same. For any substitution $\theta=\{V_1/t_1,V_2/t_2,\ldots,V_n/t_n\}$ and expression s, the expression $s\theta$ is the result of replacing each occurance of the variable V_i in s, $1 \le i \le n$ by the term t_i. If $t=s\theta$ for a substitution θ then t is called an *instance* of s. If S is a set of expressions then $S\theta=\{s\theta \mid s \in S\}$. If two sentences are instances of one another they are equal up to renaming variables, and unless indicated otherwise, we do not distinguish between them.

Let θ be a substitution. A finite non-empty set of atoms S is *unifiable* by θ if $S\theta$ is a singleton. If $\theta_1=\{t_1/V_1,t_2/V_2,\ldots,t_n/V_n\}$ and θ_2 are two substitutions, then $\theta_1 \circ \theta_2=\theta_1' \cup \theta_2'$ where θ_2' is the set of all elements of θ_2 whose variables are not among $V_1,\ldots,V_n$ and $\theta_1'=\{t_i\theta_2 \mid 1 \le i \le n, t_i\theta_2 \ne V_i\}$. It can be shown that for any string s and two substitutions θ_1 and θ_2, the equality $(s\theta_1)\theta_2=s(\theta_1 \circ \theta_2)$ holds. A substitution θ is a *most general unifier* of S if θ is a unifier for S, and for any other unifier θ_1 for S there is a substitution θ_2 such that $\theta=\theta_1 \circ \theta_2$. Robinson describes an algorithm that computes the most general unifier of any set, if it exists. Paterson and Wegman [Paterson & Wegman 76] describe such an algorithm that operates in linear time.

> *Definition 4.1:* Let $A \leftarrow B$, $C \leftarrow D$ be two sentences of L, $R_1 \subset B$, $R_2 \subset C$. If the sets R_1 and R_2 have a most general unifier θ, $\{P\}=R_1\theta=R_2\theta$ then $[A \cup (C-R_2) \leftarrow (B-R_1) \cup D]\theta$ is a *resolvent* of $A \leftarrow B$ and $C \leftarrow D$, and P is the *atom resolved upon*. The process of generating a resolvent is called *resolution*, with $A \leftarrow B$ as the *left component* and $C \leftarrow D$ as the *right component* of the resolution.

The contradiction backtracing algorithm constructs a counterexample piecewise, and performs only experiments which are relevant for finding such a counterexample. This introduces a further complication into the algorithm. Since in the predicate calculus case the atom P resolved upon need not be ground, one cannot always test its truth directly with the oracle for M. The solution to this problem is to instantiate P to a ground atom before giving it to the oracle. 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.

Algorithm 2: Backtracing Contradiction with Crucial Experiments.

Input: An oracle for a model *M*, and an ordered binary tree of sentences with the following properties:

1. The root is the empty sentence.

2. Every leaf is either an hypothesis or an observational sentence true in *M*, and no two leaves share a variable.

3. Every node which is not a leaf is a binary resolvent of its sons, where the left son is the left component and the right son is the right component of the resolution.

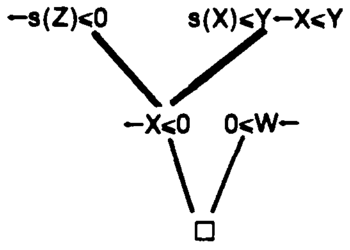
Output: An hypothesis that occurs in a leaf of the resolution tree and is false in *M*.

The Algorithm:

```
Set k to 0, $N_0$ to the root of the tree and $\theta_0$ to {}.
while $N_k$ is not a leaf do
    Let $P$ be the atom resolved upon at $N_k$ with a most general unifier $\theta$.
    Choose a substitution $\theta'$ that instantiates $P\theta_k$ to a ground atom $P_k$.
    Set $\theta_{k+1}$ to $\theta \circ \theta_k \circ \theta'$.
    Test whether $P_k$ is true in $M$.
    case $P_k$ is
        true: Set $N_{k+1}$ to the left son of $N_k$.
        false: Set $N_{k+1}$ to the right son of $N_k$.
    Set k to k+1.
Output $N_k$.
```

The following is an example of the use of the contradiction backtracing by a model inference algorithm. It occurs while the algorithm is trying to infer a finite axiomatization of the relation < over the natural numbers, generated by the constant 0 and the successor function s(X). Assume that the algorithm already conjectured the hypotheses 0<X← and ←s(X)<0, and encountered the fact <s(0)<s(s(0)),*true*>. It suggested the hypothesis s(X)<Y←X<Y, so together with the hypothesis 0<X← the sentence s(0)<s(s(0))← can be derived. However, after adding the new hypothesis the derivation in Figure 4-2 can also be constructed. So let us apply the contradiction backtracing algorithm to it to see which of the three hypotheses involved is detected to be false.

Figure 4-2: Backtracing Contradictions in Predicate Logic.



The atom resolved upon at the root is $0<0$. The oracle is called on $0<0$ and answers '*true*', so the left branch is chosen, N_1 is set to $\leftarrow X<0$ and θ_1 to $\{X/0\}$. The atom resolved upon at N_1 is $s(X)<0$, and $(s(X)<0)\theta_1$ is $s(0)<0$. The oracle answers on this test '*false*', the right branch is chosen with $\theta_2=\{Y/0\}\circ\theta_1=\{X/0,Y/0\}$, and the leaf $N_2=s(X)<Y\leftarrow X<Y$ is detected to be false. The counterexample constituted by the results of the experiments is $s(0)<0\leftarrow 0<0$, which is also $N_2\theta_2$. Note that not every ground instance of the hypothesis $s(X)<Y\leftarrow X<Y$ is false. For example the substitution $\{X/0,Y/s(0)\}$ yields the ground instance $s(0)<s(0)\leftarrow 0<s(0)$ which is true.

Before demonstrating the correctness of the algorithm, let us define more precisely how results of experiments can constitute a counterexample to an hypothesis. We say that a sentence $A'\leftarrow B'$ *subsumes* the sentence $A\leftarrow B$ if there is a substitution θ such that $A'\theta \subset A$ and $B'\theta \subset B$, or, in short, $(A'\leftarrow B')\theta \subset (A\leftarrow B)$ (see Section 5.3 for a more thorough treatment of subsumption). Assume that the ground atoms $P_1,P_2,\ldots,P_k$ were tested. Let B be the subset of the atoms $\{P_1,P_2,\ldots,P_k\}$ found to be true in M, A the subset found false in M. Then the ground sentence $A\leftarrow B$ is false in M, and similarly for every sentence which subsumes it. In such a case $A\leftarrow B$ is refered to as the *counterexample* to p via θ constituted by the outcome of testing $P_1,P_2,\ldots,P_k$.

Lemma 4.2: Let M be a model of L, T an ordered binary resolution tree of the empty sentence from sentences of L. If we apply algorithm 2 to T and the sentence N_k is reached after testing the ground atoms $P_1,\ldots,P_k$, then N_k is false in M and the outcome of these tests constitute a counterexample to N_k via θ_k.

Proof: The proof is by induction on k, the number of tests. If $k=0$ then $N_0=\square$, which is false in M and $\theta_0=\{\}$. The counterexample constituted by no experiments to $\square\theta_0$ is also the empty sentence.

Inductively assume that Algorithm 2 tested the ground atoms $P_1,\ldots,P_k$, reached node N_k, and the outcome of the tests constitutes a counterexample $A\leftarrow B$ to the sentence $N_k=(A'\leftarrow B')$ via the substitution θ_k. Let P be the atom resolved upon at N_k, where the left son of N_k is $A_1\leftarrow B_1\cup R_1$, the right son of N_k is $A_2\cup R_2\leftarrow B_2$, the sets R_1 and R_2 have a most general unifier θ such that $R_1\theta=R_2\theta=\{P\}$ and $A'\leftarrow B'=(A_1\cup A_2\leftarrow B_1\cup B_2)\theta$.

$$\begin{tikzcd}
A_1\leftarrow B_1\cup R_1 \arrow[dr, dash] & & A_2\cup R_2\leftarrow B_2 \arrow[dl, dash] \\
& N_k=A'\leftarrow B'=((A_1\cup A_2)\leftarrow(B_1\cup B_2))\theta &
\end{tikzcd}$$

Assume that $P\theta_k$ is instantiated with a substitution θ' to a ground atom P_{k+1}, θ_{k+1} is set to $\theta\circ\theta_k\circ\theta'$, and P_{k+1} is tested in M. There are two cases:

Case 1: P_{k+1} is true in M. Then algorithm 2 sets N_{k+1} to the left son of N_k which is $A_1\leftarrow B_1\cup R_1$. By the inductive assumption $A\leftarrow B$ is false in M. Since P_{k+1} is true in M the sentence $A\leftarrow B\cup\{P_{k+1}\}$ is also false in M. Since $(A_1\leftarrow B_1)\theta \subset N_k$, the inclusion $(A_1\leftarrow B_1)\theta\theta_k\theta'=(A_1\leftarrow B_1)\theta_{k+1}\subset N_k\theta_k\theta'$ holds, and $N_k\theta_k\theta'$ is equal to $N_k\theta_k$ since the latter is ground. Together with the fact that $N_k\theta_k\subset(A\leftarrow B)$ one can conclude that $(A_1\leftarrow B_1)\theta_{k+1}\subset(A\leftarrow B)$. Also, since $R_1\theta=P$ and $P\theta_k\theta'=P_{k+1}$ then $R_1\theta_{k+1}=P_{k+1}$, and therefore $N_{k+1}\theta_{k+1}=(A_1\leftarrow B_1\cup R_1)\theta_{k+1}\subset(A\leftarrow B\cup\{P_{k+1}\})$

In other words, N_{k+1} subsumes $A\leftarrow B\cup\{P_{k+1}\}$ via θ_{k+1}, therefore N_{k+1} is false in M and the outcome of testing $P_1,\ldots,P_{k+1}$ constitutes a counterexample to N_{k+1} via the substitution θ_{k+1}.

Case 2: P_{k+1} is false in M. The symmetric argument applies. ■

The following theorem is an immediate corollary of Lemma 4.2:

Theorem 4.3: Let M be a model of L, T an ordered binary resolution tree of the empty sentence from sentences of L. If the depth of T is n then algorithm 2 applied to T performs no more than n experiments in M and outputs an hypothesis p false in M. ■

4.2. On The Refutability of Scientific Hypotheses.

Detecting a false hypothesis in a refuted theory seems to be a rather difficult task, and it was even claimed by the philosopher Pierre Duhem to be impossible, since "The physicist can never subject an isolated hypothesis to an experimental test, but only a whole group of hypothesis" [Duhem 54]. I review Duhem's argument and relate it to the assumptions on which the contradiction backtracing algorithm operates. Essentially, Duhem's argument against the possibility of such uniquely falsifying experiments has two components: a logical component and an epistemological one. The logical component of Duhem's argument is that "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. The physicist may declare that this error is contained in exactly the proposition he wishes to refute, but is he sure it is not in another proposition? If he is, he accepts implicitly the accuracy of all the other propositions he has used, and the validity of his conclusion is as great as the validity of his confidence."

This claim of Duhem is true of one crucial experiment. However, as demonstrated by the contradiction backtracing algorithm, once such a contradiction between the facts and the theory is discovered, one can algorithmically decide which additional experiments to perform, so their results will single out a false hypothesis in the refuted theory. In addition, Duhem argues that when a contradiction occurs between the hypotheses and the facts one can conclude that a certain hypothesis is false only by implicitly accepting the accuracy of all other hypotheses used in the derivation. The contradiction backtracing algorithm refutes this claim, by demonstrating that one can conclude the falsity of an hypothesis just by accepting the accuracy of factual judgements concerning *ground (variable-free) atoms only*, that is, sentences which refer to concrete objects and events.

The epistemological component of Duhem's argument is that even factual judgements concerning concrete objects and events do involve some theoretical assumptions: "the physicist who carries out an experiment, or gives a report of one, implicitly recognizes the accuracy of a whole group of theories". An extension to the model inference problem that incorporates the notion of theoretical terms (more precisely, theoretical predicates), was discussed in Section 2.3. The contradiction backtracing algorithm is applicable to this extension as well. Informally, the way the contradiction backtracing algorithm tests ground atoms in the model is as follows: if the predicate in the atoms to be tested is observational, then it can be tested directly in the model. Otherwise we assume that the algorithm has some built in knowledge about the theoretical predicates it uses, and using this knowledge it can decide whether the ground atoms is true or not. In both cases the contradiction backtracing algorithm leads to an hypothesis false in the model. More formally, the assumption is that for every ground atom $P=p(t_1, t_2, \ldots, t_1)$ of L, either $P \epsilon L_o$ or p is a theoretical predicate with a fixed interpretation, known to the inductive inference algorithm.

To summarize the discussion of Duhem's argument, it seems that the existence of the contradiction backtracing algorithm refutes the logical component of his argument:

1. That crucial experiments can refute only a collection of hypotheses.

2. That in order to refute an hypothesis, one has to accept the truth of other hypothesis.

The epistemological component of Duhem's argument still holds, however, if one rejects the possibility of factoring the terms used in scientific theories into *theoretical terms* and *observational terms*, as defined here.

4.3. An Application: Debugging Logic Programs.

The idea of contradiction backtracing can be readily applied to the debugging of logic programs. If a logic program computes the wrong input-output relation, or returns the wrong input on a given output, there is at least one clause in this program that is false under the intended interpretation of its predicates. As an example, from the following program for computing the subsequence relation over lists

```
subsequence([],[])
subsequence(L1,[X|L2]) ← subsequence(L1,L2).
subsequence([X|L1],[Y|L2]) ← subsequence(L1,L2)
```

one can compute that subsequence([a,b,c],[a,a,c,d]). Under the intended interpretation of the predicate subsequence(X,Y), that is, X is a subsequence of Y, the third clause of the program is false, and the following is a counterexample to it which shows that:

```
    subsequence([b],[a]) ← subsequence([],[])
```

The 'bug' in this clause causes the program to accept any list L1 as a subsequence of any other list L2, provided that L1 is not longer then L2. Applying the contradiction backtracing algorithm to the computation (proof) of subsequence([a],[b]), it will first test subsequence([],[]), find it to be true, then test subsequence([a],[b]), find it to be false, and provide the above counterexample to the third clause of subsequence. The reader is encouraged to find a correct logic program for subsequence. A solution is provided on page 24.

In the following we describe a Prolog program that implements the contradiction backtracing algorithm, for the case where the hypotheses are Horn clauses, the observational language contains all atoms, and the oracle is the user. We provide an annotated listing of a session with this program, in which a user is attempting to debug a faulty quicksort program.

The method of collecting substitutions in a resolution proof, used by the contradiction backtracing algorithm, was suggested first by Green [Green 69]. His goal was to construct an answer to a question formed as a logical sentence, by proving this sentence from hypotheses and collecting the substitutions used in the proof. Prolog uses the very same idea, except that the sentence to be proved is always of the form ←A and the hypotheses are restricted to be Horn clauses. 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. Figure 4-3 contains a full listing of such an implementation. A similar implementation is incorporated in the Model Inference System.

Figure 4-3: A Prolog Program for Backtracing Contradictions.

```
backtrace((P,Q),CE) ← !, backtrace(P,CE), backtrace(Q,CE).
backtrace(P,CE) ← clause(P,Q), backtrace(Q,CE), resolve((P←Q),CE).
backtrace(P,CE) ← P.

resolve((P←Q),(P←Q)) ← var(CE), !, ask(P,V), ( V=false, CE=(P←Q) ; V=true ).
resolve((P←Q),CE).

ask(P,V) ← recorded(fact,(P,V)), !.
ask(P,V) ← repeat, write(P), put(63), nl, read(V), (V=true ; V=false), record(fact,(P,V)), !.
```

For a given atom Atom false in the intended model M, the procedure first tries to construct a proof of Atom, and if it succeeds it returns an instance of a false hypothesis CE (for CounterExample) used in the derivation of Atom. The reader is referred to the Prolog manual [Pereira et al. 78] for details concerning Prolog. In particular note that the internal representation of a sentence $\{P\} \leftarrow \{Q_1, Q_2, \ldots, Q_n\}$ is $P \leftarrow (Q_1,(Q_2,(\ldots,Q_n)\ldots))$, and of $\{P\} \leftarrow$ is P←true.

The procedure backtrace is a simple Prolog interpreter, augmented with a call to resolve after any successful use of a clause. A call backtrace(Atom,CE) first constructs a proof of the atom Atom, if such a proof exists and while 'popping' back from the recursive construction of the proof it calls resolve(P←Q,CE) at each step of the proof that introduces a clause P←Q. The underlying logic of resolve is quite involved. The critical idea is that the first call to resolve((P←Q),CE) in which P is false unifies the variable CE with the clause P←Q. The correctness of this idea is argued inductively as follows: If Q=true, and P is false then the clause P←Q is false, and backtrace returns a false clause in the variable CE. Inductively assume that in all previous calls to resolve, made when backtrace was returning from the recursive construction of the proof, the atom resolved upon P was tested and was found true. Then the condition Q is true. Therefore if P is now tested and is found false then the clause P←Q is false, the variable CE is unified with P←Q and backtrace returns an instance of a false clause. All the fine points in the proof of Lemma 4.2, raised by the need to collect the substitutions used in the construction of the proof and the construction of the counterexample, are taken care of automatically by the unification procedure and the scheme of shared variables built into Prolog.

The following is an execution trace of a user running the program in Figure 4-3, attempting to debug a the following quicksort program. The quicksort program we start with contains several bugs, some trivial and some more subtle.

```
qsort([X|L],L0) ← partition(L,X,L1,L2), qsort(L1,L3), qsort(L2,L4), append(L3,L4,L0).
qsort([],[]).

partition([X|L],Y,L1,[X|L2]) ← partition(L,Y,L1,L2).
partition([X|L],Y,[X|L1],L2) ← X≤Y, partition(L,Y,L1,L2).
partition([],X,[],[]).


append([X|L1],L2,L3) ← append(L1;L2,L3).
append([],L,L).
```

In order to debug a program, we need to know its intended interpretation. qsort(X,Y) is true just in case Y is the list X sorted. partition(L,X,L1,L2) is true if L1 contains the elements of L less than or equal to X and L2 those elements of L which are greater than X. append(X,Y,Z) is true if the list Z is the result of appending the list X to the list Y. In the following trace of execution, lines beginning with '|' were typed by the user, and the rest is the system's output. We try to execute this quicksort program,

```
| ?- qsort([2,1,2],X).

X = []
```

and get an incorrect result. So we call backtrace, which prompts us with queries on the truth of atoms, on which we answer '*true*' or '*false*', according to the above interpretation of qsort, partition and append.

```
| ?- backtrace(qsort([2,1,2],X),CE).
partition([],2,[],[])?
|: true.
partition([2],2,[],[2])?
|: false.

X = [],
CE = (partition([2],2,[],[2]) ← partition([],2,[],[]))
```

Backtrace returned a counterexample to the first clause of partition, which is false since its condition is true while its conclusion is false. The obvious error is that the X > Y check is omitted. We patch this clause by adding the check,

```
partition([X|L],Y,L1,[X|L2]) ← X>Y, partition(L,Y,L1,L2).
```

and try the new program.

```
| ?- qsort([2,1,2],X).

X = []
```

Since it still returns a wrong answer, we call backtrace again.

```
| ?- backtrace(qsort([2,1,2],X),CE).
partition([2],2,[2],[])?
|: true.
partition([1,2],2,[1,2],[])?
|: true.
partition([],1,[],[])?
|: true.
```

```
 partition([2],1,[],[2])?
|: true.
qsort([],[])?
|: true.
append([],[],[])?
|: true.
 qsort([2],[])?
|: false.

X = [],
CE = (qsort([2],[]) ← partition([],2,[],[]), qsort([],[]), qsort([],[]), append([],[],[]))
```

Now CE is bound to a counterexample to the main clause of qsort itself. It is clear that all the subgoals to qsort([2],[]) returned correct answers, we simply forgot to include the element on which the partition was done in the final result. So we do that, by modifying the call to append:

```
qsort([X|L],L0) ← partition(L,X,L1,L2), qsort(L1,L3), qsort(L2,L4), append([X|L3],L4,L0).
```

Note that qsort([],[]) was tested by the algorithm several times, but since after the first test the fact <qsort([],[]),true> was recorded in the data base, the user was bothered with this question only once. As the debugging process progresses, the data base has more facts about the procedures being debugged, and the user intervenes less and less in the process. This incremental behavior of the algorithm might prove very useful in debugging larger systems. We try qsort again.

```
| ?- qsort([2,1,2],X).

X = []

yes
| ?- backtrace(qsort([2,1,2],X),CE).
append([2],[],[])?
|: false.

X = [],
CE = (append([2],[],[]) ← append([],[],[]))
```

and the obvious bug in append is discovered. Note that a program can be debugged as a whole, and there is no need to finish debugging sub-procedures before debugging the main procedure. We correct the first clause of append

```
append([X|L1],L2,[X|L3]) ← append(L1,L2,L3).
```

and try again

```
| ?- qsort([2,1,2],X).

X = [2,1,2]

yes

| ?- backtrace(qsort([2,1,2],X),CE).
append([2],[],[2])?
|: true.
qsort([2],[2])?
|: true.
append([],[2],[2])?
|: true.
append([1],[2],[1,2])?
|: true.
```

```
qsort([1,2],[1,2])?
|: true.
append([1,2],[],[1,2])?
|: true.
append([2,1,2],[],[2,1,2])?
|: true.
qsort([2,1,2],[2,1,2])?
|: false.


X = [2,1,2],
CE = (qsort([2,1,2],[2,1,2])← partition([1,2],2,[1,2],[]), qsort([1,2],[1,2]), qsort([],[]),
            append([2,1,2],[],[2,1,2]))
```

Here it is a little bit harder to see what is wrong, but looking at the clause of qsort we just corrected reveals that we corrected it wrongly. The element on which the partition was done was added to the final result, but in the wrong place: in the head of the list of the numbers smaller then it, instead of in the head of the list of the larger numbers. This can be corrected easily

```
qsort([X|L],L0) ← partition(L,X,L1,L2), qsort(L1,L3), qsort(L2,L4), append(L3,[X|L4],L0).
```

We try qsort again

```
| ?- qsort([2,1,2],X).

X = [1,2,2]

yes
| ?- qsort([5,2,45,3,24,543,2,3,1],X).

X = [1,2,2,3,3,5,24,45,543]

yes
```

and it works!!! The final quicksort program, after debugging, is the following:

```
qsort([X|L],L0) ← partition(L,X,L1,L2), qsort(L1,L3), qsort(L2,L4), append(L3,[X|L4],L0).
qsort([],[]).

partition([X|L],Y,L1,[X|L2]) ← X>Y, partition(L,Y,L1,L2).
partition([X|L],Y,[X|L1],L2) ← X≤Y, partition(L,Y,L1,L2).
partition([],X,[],[]).

append([X|L1],L2,[X|L3]) ← append(L1,L2,L3).
append([],L,L).
```

At least two additional facilities are needed to make a debugger based on contradiction backtracing practical. First, there should be a way for the user to correct wrong facts he or she typed in. Second, the contradiction backtracing algorithm can be applied only in case a call to a program succeeds, but with a wrong answer. If the call simply fails, another approach is necessary. The following section suggests an approach as to how to add clauses to a logic program in case it fails, but other approaches, based on modifying existing clauses rather than adding new ones might be more practical for such a debugging task.

5. Refining Refuted Hypotheses.

Recall the two questions presented on page 6. The contradiction backtracing algorithm solves the first one by detecting a false hypothesis in a *too strong conjecture*. Removing such an hypothesis from a conjecture might result in a *too weak conjecture*; this section addresses the question of how to strengthen such a conjecture. For this task we devise *refinement operators* over the sentences of L_h. Intuitively speaking, a refinement operator suggests a logically weaker plausible replacement to a refuted hypothesis. It can be used in an incremental inductive inference algorithm in the following way: whenever a counterexample to an hypothesis is discovered, modify the refuted conjecture by removing this hypothesis. If the resulting conjecture is too weak, strengthen it by adding other hypotheses, selected from refinements of this and previously refuted hypotheses.

In the following we develop the notions of *refinement* and *refinement operator*. The completeness of a refinement operator for a given hypothesis language is defined, and examples are provided of different refinement operators, complete for different classes of hypotheses languages. Properties of the *refinement graph* induced by a refinement operator on the sentences of L_h are discussed, and a most general refinement operator, which is complete for any first order language L, is described.

5.1. Refinement Operators.

We assume some structural complexity measure *size*, which is a function from sentences of L to natural numbers, with the property that for every $n>0$ the set of sentences of size n is finite. For any set of sentences S and any n>0, define $S(\text{n})$ to be the set $\{p\in S \mid size(p)\leq \text{n}\}$.

Definition 5.1: Let L be a first order language, p and q sentences of L. We say that q is a *refinement* of p if p implies q and $size(p)<size(q)$.

Definition 5.2: A *refinement operator* ρ is a mapping from sentences of L to subsets of their refinements, such that for any $p\in L$ and any n>0 the set $\rho(p)(\text{n})$, that is, the set $\rho(p)$ restricted to sentences of size≤n, is computable.

A refinement operator over L induces a partial order $\leq_\rho$ over L, with the empty sentence $\Box$ as a minimal element: a finite sequence of sentences $p=p_0,p_1,\ldots,p_n=q$ such that $p_{i+1}\in\rho(p_i)$ is called a *finite total ρ-chain*; we say that $p\leq_\rho q$ if there is a finite total ρ-chain from p to q; we say that $p<_\rho q$ if $p\leq_\rho q$ and $q\not\leq_\rho p$. The relation $\leq_\rho$ generalizes to sets of sentences. We say that for any two sets of sentences $T,S\subset L$, $T\leq_\rho S$ if for every $q\in S$ there is a sentence $p\in T$ such that $p\leq_\rho q$. For any sentence p, the set $\{q\in L \mid p\leq_\rho q\}$ is denoted by $\rho^*(p)$. The set $\rho^*(\Box)$ is denoted by ρ^*.

Definition 5.3: Let $S\subset L$ be a set of sentences that includes the empty sentence $\Box$. A refinement operator ρ over L is said to be *complete for* S if $\rho^*=S$.

Recall the intended use of a refinement operator: when an hypothesis is refuted, the inference algorithm searches for a replacement to it among its refinements. Therefore the refinement operator incorporated by such an algorithm is required to be complete for the hypothesis language L_h. Below are examples of concrete refinement operators used by the Model Inference System, which are complete for different classes of hypotheses languages.

Following [Reynolds 70] we define a relation $\rightarrow$, and consider the operator ρ_1 defined by $\rho_1(p)=\{q \mid p\rightarrow q\}$. Let $p\in L$ be a sentence, then $p\rightarrow q$ if one of the following holds:

1. $p=\Box$ and $q=\text{a}(X_1,X_2,\ldots,X_n)$, for some n-place predicate symbol a of L, n≥0, and $X_1,X_2,\ldots,X_1$ are n distinct variables.

2. $p=P$ for some atom P, $q=P\{U/V\}$, where U and V are distinct variables occuring in P.

3. $p=P$ for some atom P, and $q=P\{V/\text{f}(X_1,X_2,\ldots,X_n)\}$ 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 occuring in P.

In the following we adopt Reynolds' definition [Reynolds 70] of the size of a sentence as our concrete structural complexity measure over L: the *size* of a sentence p, $size(p)$, is the number of symbol occurrences in p (excluding punctuation symbols) minus the number of distinct variables occuring in p. We show that ρ_1 is a refinement operator with respect to this measure.

Theorem 5.4: ρ_1 is a refinement operator over L, complete for the atoms of L.

Proof: For any two sentences p and q, if q is an instance of p then $p \vdash q$. By the definition of a size of a sentence, if $q \in \rho_1(p)$ then $size(q)=size(p)+1$, hence ρ_1 is a mapping from sentences to their refinements. It is clear from the definition of ρ_1 that for any $p \in L$ the set $\rho_1(p)$ is computable and finite, hence the set $\rho_1(p)(n)$ is computable for any $n>0$, therefore ρ_1 is a refinement operator.

Theorem 4 in [Reynolds 70] shows that for any atom $q \in L$ there exists a finite total ρ_1-chain from $\square$ to q. This establishes the completeness of the refinement operator for the atoms of L. ■

Axiom systems that contain only atoms are relatively trivial. However, a slight generalization of ρ_1 results in a refinement operator which is complete for a language rich enough to axiomatize some of the examples given above, such as < and plus over integers and append over lists.

Definition 5.5: Let ρ and ρ' be two refinement operators over L. We say that ρ' is *more general* than ρ if $\rho^* \subset \rho'^*$.

Consider the following generalization of ρ_1. Let $p \in L$ be a sentence. Then $q \in \rho_2(p)$ if one of the following holds:

1. $q \in \rho_1(p)$.
2. 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($t_1,t_2,\ldots,t_n$)←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 $1 \le i \le n$.

Reynolds termed sentences of the form $\{P\}$←$\{Q\}$ *transformations*. We call transformations that satisfy condition 2. above *context-free transformations*. Some non-trivial predicates have an atomic complete axiomatization via atoms and context-free transformations, and the Model Inference System used the refinement operator ρ_2 to infer them. These predicates include: the order relation and addition over integers, generated by 0 and the successor function X',

```
X≤X
X≤Y' ← X≤Y

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

the prefix, suffix and subsequence relations over lists, generated by the empty list [] and the list constructor [X|Y],

```
prefix([],X)
prefix([A|X],[A|Y]) ← prefix(X,Y)

suffix(X,X)
suffix(X,[A|Y]) ← suffix(X,Y)

subsequence([],X)
subsequence([A|X],[A|Y]) ← subsequence(X,Y)
subsequence(X,[A|Y]) ← subsequence(X,Y)
```

concatenation relations over lists,

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

conc([],X,[X])
conc([A|X],Y,[A|Z]) ← conc(X,Y,Z)
```

and the subtree relation for binary trees, generated by t(X,Y) and some constants.

```
subtree(X,X)
subtree(X,t(Y,Z)) ← subtree(X,Y)
subtree(X,t(Y,Z)) ← subtree(X,Z)
```

Theorem 5.6: ρ_2 is a refinement operator over L, complete for the atoms and context-free transformations of L.

Proof: For any two sentences p and q, such that $p=\{P\}\leftarrow$ and $q=\{P\}\leftarrow\{Q\}$, $p\vdash q$ and $size(p)<size(q)$, hence ρ_2 is a mapping from sentences to their refinements. For any $p\in L$ the set $\rho_2(p)$ is finite, hence the set $\rho_2(p)(n)$ is computable for any $n>0$, therefore ρ_2 is a refinement operator.

The previous theorem shows that ρ_1 is complete for the atoms of L. Hence for any context-free transformation $\{P\}\leftarrow\{Q\}$ there is a finite total ρ_1-chain from $\square$ to $\{P\}\leftarrow$, which is also a ρ_2-chain. By the definition of ρ_2, $(\{P\}\leftarrow\{Q\})\in\rho_2(\{P\}\leftarrow)$. Hence there exists a finite total ρ_2-chain from $\square$ to $\{P\}\leftarrow\{Q\}$, which establishes the completeness of ρ_2 for the atoms and context-free transformations of L. ■

There are two simple generalizations of context-free transformations; we introduce them here through examples, and leave it to the reader to define a complete refinement operator for them. Such refinement operators were used by the Model Inference System to infer axiomatizations such as the examples below. The first is multiple context-free Horn sentences, with which we can axiomatize predicates like dense partial order, with 0 and 1 as endpoint and m(X,Y) interpreted as a point between X and Y,

```
0<X
X<1
X<X
m(X,Y)<X ← Y<X
m(X,Y)<Y ← X<Y
m(X,Y)<Z ← X<Z, Y<Z
```

binary tree isomorphism

```
isomorphic(X,X)
isomorphic(t(X1,Y1),t(X2,Y2)) ← isomorphic(X1,X2), isomorphic(Y1,Y2)
isomorphic(t(X1,Y1),t(X2,Y2)) ← isomorphic(X1,Y2), isomorphic(Y1,X2)
```

and satisfiability of boolean formulas over true and false.

```
satis(true)
satis(and(X,Y)) ← satis(X), satis(Y)
satis(or(X,Y)) ← satis(X)
satis(or(X,Y)) ← satis(Y)
satis(not(X)) ← unsatis(X)

unsatis(false)
unsatis(or(X,Y)) ← unsatis(X), unsatis(Y)
unsatis(and(X,Y)) ← unsatis(X)
unsatis(and(X,Y)) ← unsatis(Y)
unsatis(not(X)) ← satis(X)
```

The second generalization of context-free transformations is term-free transformations with auxiliary predicate. Within this class one can axiomatize, among others, multiplication and exponentiation,

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

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

list reversal,

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

the subset relation,

```
    subset([],X)
    subset([A|X],Y) ← subset(X,Y), member(A,Y)
```

and insertion sort

```
    sort([],[])
    sort([A|X],Y) ← sort(X,Z), insert(A,Z,Y)

    insert(A,[],[A])
    insert(A,[B|X],[A,B|X]) ← A<B
    insert(A,[B|X],[B|Y]) ← B<A, insert(A,X,Y)
```

[Angluin & Shapiro 81] will report on a fuller characterization of the expressive power of these classes of axioms.

5.2. Refinement Graphs.

For any first order language L, a refinment operator ρ over L induces a *refinement graph* $G_\rho(\rho^*,E)$. This is a directed, acyclic graph, where the vertices are the sentences of ρ^*, and there is an edge $(p,q)\in E$ if and only if $q\in\rho(p)$ and $p\in\rho^*$. The graph is acyclic since $(p,q)\in E$ implies that $size(p)<size(q)$. Let p and q be two sentences in ρ^*. A *path* in G_ρ from p to q is a finite sequence of sentences $p=p_0, p_1,..., p_n=q$ such that $(p_i,p_{i+1})\in E$ for $0\leq i<n$. Note that a path in G_ρ is exactly a finite total ρ-chain. We say that q is *reachable* from p in G_ρ if there is a path from p to q in G_ρ; p is the *predecessor* of q if $(p,q)\in E$, and in such a case q the *successor* of p.

The following are some properties of G_ρ:

1. A sentence q is reachable in G_ρ from a sentence $p\in\rho^*$ if and only if $p\leq_\rho q$, and therefore only if $p\vdash q$.
2. The empty sentence $\Box$ is not reachable from any $p\in\rho^*$.
3. Every $p\in\rho^*$ is reachable from $\Box$.
4. If a sentence $p\in\rho^*$ is true in a model M for L, then all its successors in G_ρ are also true in M; if p is false in M, then all its predecessors in G_ρ are false in M.

For any given model M for L, we define the set of minimal sentences with respect to $\leq_\rho$ which are true in M.

Definition 5.7: Let M be a model of L, and ρ a refinement operator over L. A sentence $p\in\rho^*$ is called a *source sentence of the model M* with respect to ρ if p is true in M and every sentence $q\in\rho^*$ such that $q<_\rho p$ is false in M. The set of source sentences of M with respect to ρ is called the *source set of the model M*, and is denoted L_ρ^M.

By the definition of a source sentence, $p\in\rho^*$ is a source sentence if all sentences in every path from the empty sentence $\Box$ to p in G_ρ are false in M. An infinite sequence of sentences $p_0,p_1,...$ is called an *infinite ascending (or descending) ρ-chain* if $p_i<_\rho p_{i+1}$ (or $p_{i+1}<_\rho p_i$) for $i\geq 0$. Since $q\in\rho(p)$ implies that $size(p)<size(q)$ there are no infinite descending ρ-chains in L. Hence for every $p\in L$ the set $\{q \mid q\leq_\rho p\}$ is finite, and for every sentence $p\in\rho^*$ true in M which is not a source sentence there is a source sentence q such that $q<_\rho p$. The source set of a model has the following property:

Theorem 5.8: Let L be a first order language, M a model of L, $\Box\in L_o\subset L_h\subset L$ and ρ a refinement operator over L which is complete for L_h. If $L_h(k_0)$ contains a finite L_o-complete axiomatization of M, for some $k_0\geq 0$, then $L_\rho^M(k_0)$ is also such an axiomatization.

Proof: Let $T\subset L_h$ be a finite L_o-complete axiomatization of M, k_0 the maximal size of any sentence in T. If T is not a subset of $L_\rho^M(k_0)$, then there is a sentences $p\in T$ which is not a source sentence. By definition of a source set of a model, there is a source sentence $q\in L_\rho^M(k_0)$ such that $q<_\rho p$.

Replace p in T by q. Since $q <_\rho p$ it follows that $q \vdash p$. Since q is true in M and T is an L_o-complete axiomatization of M the resulting set of sentences is also an L_o-complete axiomatization of M. Repeat the replacement step until all the sentences in the resulting axiomatization T' are source sentences. T' is a finite L_o-complete axiomatization of M and a subset of $L_\rho^M(k_0)$. Since $L_\rho^M(k_0)$ is finite and is true in M, it follows that $L_\rho^M(k_0)$ is also a finite L_o-complete axiomatization of M. ■

Note that if T_1 and T_2 are finite sets of sentences such that $T_2 <_\rho T_1$, then there exists some j such that $T_1 \vdash_j T_2$, and therefore if M is an h-easy model of L, then the source set of M contains an h'-easy finite L_o-complete axiomatization of M, for $h'(n)=h(n)+j$. However, for the correctness of the following inductive inference algorithm, ρ has to satisfy a stronger requirement:

Definition 5.9: A refinement operator ρ is said to be *conservative* with respect to $\vdash$ if for any two finite sets of sentences T_1 and T_2 such that $T_1 <_\rho T_2$ and any $p \in L$, $T_2 \vdash_n p$ implies that $T_1 \vdash_n p$.

If the proof procedure is resolution [Robinson 65] and T_1 subsumes T_2, then $T_2 \vdash_n \Box$ implies that $T_1 \vdash_n \Box$, and because resolution proofs are by way of contradiction, for any $p \in L$, $T_2 \vdash_n p$ implies that $T_1 \vdash_n p$. Since for all the refinement operators ρ described in this paper the relation $<_\rho$ is a subset of the subsumption relation, they are all conservative with respect to resolution.

Corollary 5.10: Under the same assumption as Theorem 5.8, if ρ is conservative with respect to $\vdash$, h is a total recursive and $T \subset L_h(k_0)$ is a finite L_o-complete h-easy axiomatization of M for some $k_0>0$, then $L_\rho^M(k_0)$ is also such an axiomatization.

5.3. A Most General Refinement Operator.

In this section we define, for any given first order language L, an operator ρ_0 and prove it to be a refinement operator complete for L, that is $\rho^*=L$. The existence of a most general refinement operator is interesting mostly for theoretical reasons. In practice, some information on the structure of the set of hypotheses to be inferred is usually known. In such a case we always prefer a less general refinement operator which is complete for this class, to guarantee more efficient inference of the models of interest.

In the following, a partial order $<_0$ is defined for any first order L, with $\Box$ as the minimal element. A refinement operator ρ_0 is defined, and a sketch of an algorithm for computing it is provided. Theorem 5.14 states that ρ_0 is complete for L, and is the main result of this section. To establish this result we develop some logical tools, and prove (Theorem 5.17) that $<_{\rho_0}$ is equal to $<_0$.

We begin with some definitions. A sentence p=A_1←B_1 *subsumes* a sentence q=A_2←B_2 via a substitution θ if $A_1\theta \subset A_2$ and $B_1\theta \subset B_2$, or in short $(A_1 \leftarrow B_1)\theta \subset A_2 \leftarrow B_2$. Define $p \cong q$ to stand for p subsumes q and q subsumes p. The relation $\cong$ is an equivalence relation; we use $[p]$ to denote the equivalence class of p under it. The subsumption relation induces a partial ordering on the set of equivalence classes of L with the empty sentence $\Box$ as a minimal element. Also, if p subsumes q then $p \vdash q$. For example, the sentences {plus(X,Y',Z')}← and {plus(X,Y',Z'),plus(X,Y',W)}← are equivalent; they both subsume the sentences {plus(X,Y',Z')}←{plus(Z,Y,Z')} and {plus(X,0',X')}←, and are subsumed by {plus(X,Y,Z)}←. Note that p subsumes q *does not* imply $size(p) \leq size(q)$, hence there is no refinement operator ρ such that $p <_\rho q$ if and only if p subsumes q. Plotkin [Plotkin 71a] gives examples of infinitely strictly descending chains under subsumption, a property that $<_\rho$ can not have. The subsumption relation can be extended to sets: for any two sets of sentences S and T, S subsumes T if for every $q \in T$ there exists $p \in$S such that p subsumes q.

The following definitions and results are by Plotkin. [Plotkin 70, Plotkin 71b, Plotkin 71a]. A set of atoms S is said to be *reduced* if S ⊂ S' and S≅S' implies that S=S'. In other words, S is reduced if it is not equivalent to a proper subset of itself. A sentence p=A←B is reduced if both A and B are reduced. As it turns out, if both p and q are reduced and $p \cong q$ then p and q are equal up to renaming variables. Plotkin describes an algorithm that, given a sentence p, computes a reduced sentence q such that $p \cong q$. This algorithm can be used to test sentences for their equivalence. In this section, 'a sentence' will mean 'a reduced representative of the equivalence class of this sentence'.

Definition 5.11: A substitution θ is said to *decrease* a set of atoms S if $|S\theta| \lhd |S|$. A substitution is said to *decrease* a sentence p=A←B if it decreases either A or B.

Definition 5.12: Let p and q be two sentences of L. Then $p \leq_0 q$ if there exists a substitution θ that does not decrease p, and $p\theta \subset q$.

By the definition of $\leq_0$, $p \leq_0 q$ implies that p subsumes q, but not vice versa. Note that $p \leq_0 p$ via the empty substitution, and $p \leq_0 q$ and $q \leq_0 p$ implies that p is equal to q up to renaming of variables. The relation $\leq_0$ is transitive, since $p_1\theta_1 \subset p_2$ via a non-decreasing substitution θ_1 and $p_2\theta_2 \subset p_2$ via a non-decreasing substitution θ_2 implies that $\theta_1 \circ \theta_2$ is non-decreasing for p_1 and $p_1\theta_1 \circ \theta_2 \subset p_3$. Also note that $\Box \leq_0 p$ for every $p \epsilon L$ via the empty substitution. Hence $\leq_0$ is a partial order on the (equivalence classes of) sentences of L, with $\Box$ as a minimal element. From the above definitions it is clear that for two reduced sentences p and q, $p <_0 q$ implies $size(p) < size(q)$. Therefore under $\leq_0$ there are no infinite strictly descending chains.

Let P and Q be atoms of L and U a set of atoms or a sentence. We say that P is *more general than Q with respect to U* if there exists a substitution θ such that $P\theta = Q$ and $U\theta = U$. Let A←B be a reduced sentence. We say that P is a most general atom such that A←B ∪ $\{P\}$ is reduced if for any atom Q such that Q is more general than P with respect to A←B, the sentence A←B ∪ $\{Q\}$ is not reduced. Similarly for A ∪ $\{P\}$←B.

Ddfinition 5.13: Let p=A←B be a reduced sentence of L. Then $q \epsilon \rho_0(p)$ if exactly one of the following holds:

1. $q = p\theta$, where θ={V/W} does not decrease p and both variables V and W occur in p.
2. $q = p\theta$, where θ={V/f(X$_1$,...,X$_n$)} does not decrease p, f is an n-place function symbol, V occurs in p and every X$_i$, $1 \leq i \leq n$, is a distinct variable that does not occur in p.
3. q=A ∪ $\{P\}$←B, where P is a most general atom with respect to A←B for which A ∪ $\{P\}$←B is reduced.
4. q=A←B ∪ $\{P\}$, where P is a most general atom with respect to A←B for which A←B ∪ $\{P\}$ is reduced.

Theorem 5.14: ρ_0 is a complete refinement operator over L.

Proof: If $q \epsilon \rho_0(p)$ then p subsumes q, and therefore $p \vdash q$. Both substitutions in cases 1 and 2 are non-decreasing, and they increase the size of p by exactly 1. In cases 3 and 4 an atom is added to the sentence without changing the rest of it, hence $q \epsilon \rho_0(p)$ implies $size(p) < size(q)$, therefore ρ_0 is a mapping from sentences to their refinements.

It is clear that cases 1 and 2 hold only for finitely many sentences q, but in general there are infinitely many atoms P that satisfy cases 3 and 4. In the following we describe an effective procedure for finding all such atoms of size less than some fixed bound, hence the set $\rho(P)(n)$ is computable, therefore ρ_0 is a refinement operator over L.

Theorem 5.17 shows that if $p \leq_0 q$ then there is a finite total ρ_0-chain from p to q. Since $\Box \leq_0 q$ for any sentence $q \epsilon L$ it follows that ρ_0 is complete for L. ■

We sketch an algorithm for computing the set $\rho(p)(n)$, given a sentence p and $n>0$, then proceed to prove that ρ_0 is complete. Computing non-reducing substitutions as in cases 1 and 2 is straightforward, and only finitely many sentences can be generated by these operations. For cases 3 and 4 one has to find a most general atom P with respect to A←B, the sentence to be refined, such that the resulting sentence A ∪ $\{P\}$←B (or A←B ∪ $\{P\}$) is reduced. This set is in general infinite. For example, for the sentence

{p(X,Z)}←{p(X,f(Y)),p(f(Y),Z)}

any of the following infinite list of atoms

p(f(X),f(Y)), p(f(f(X)),f(Y)), p(f(X),f(f(Y))), p(f(f(X)),f(f(Y))),...

satisfy case 4 in the definition of ρ_o. However, all such atoms of size$\leq$k can be systematically generated in the in the following way: choose an atom $a(X_1,X_2,...,X_n)$, where a is an n-place predicate symbol of L and $X_1,X_2,...,X_n$ are distinct variables not occuring in $A \leftarrow B$. Successively choose a variable V that occurs in P and perform one of the following operations:

1. Choose a variable U that occurs in P but not in $A \leftarrow B$, and set P to $P\{V/U\}$.

2. Choose an n-place function symbol f for $n>0$ and set P to $P\{V/f(Z_1,...,Z_n)\}$, where Z_i, $1 \leq i \leq n$ are variables that do not occur in P or in $A \leftarrow B$.

3. Choose a variable U that occurs in $A \leftarrow B$, and set P to $P\{V/U\}$.

until either $A \cup \{P\} \leftarrow B$ is reduced, or $size(P)>k$. If the second condition holds, then fail. Otherwise, verify that the sentence $A \cup \{Q\} \leftarrow B$ is not reduced, for any atom Q such that $Q \leftarrow P$ via a substitution that does not change $A \leftarrow B$. If so, return P.

Note that each of the results of applying operations 1 and 2 to an atom P is an atom P' such that $P \leftarrow P'$, and in such a case $size(P')=size(P)+1$. Operations 3 can be performed only finitely many times, bounded by the number of variables in $A \leftarrow B$. Hence the algorithm terminated on every computation path. Since operations 1 and 2 are sufficient to generate all atoms of L, up to renaming variables, we are guaranteed that every atom of size$\leq$n will be generated by this procedure. Operation 3 provides the mechanism for the appropriate renaming of the variables in P. Therefore the set of atoms generated by the successful computations of this algorithm is the required set.

The following two lemmas and theorem establish that ρ_o is complete for L.

Lemma 5.15: Let p and q be two sentences such that $p\theta=q$ for some substitution θ that does not decrease p. Then there is a finite total ρ_o-chain from p to q.

Proof: This lemma is a generalization of Theorem 4 in Reynolds' paper. Examination of the proof of this theorem shows that it can be applied to p and $p\theta$ to obtain such a finite total chain. ■

Lemma 5.16: Let $p=A_1 \leftarrow B_1$ and $q=A_2 \leftarrow B_2$ be two reduced sentences such that $p \subset q$. Then there is a finite total ρ_o-chain from p to q.

Proof: The proof is by induction on n, the sum of the number of atoms in the difference sets A_2-A_1 and B_2-B_1. If n=0 then $p=q$ and the empty chain satisfies the lemma. Assume that the lemma is true for some $n\geq0$, and that there are n+1 atoms in A_2-A_1 and B_2-B_1. Let P be an atom in A_2-A_1, and A_2' be A_2-$\{P\}$ (The symmetric argument applies to $P\epsilon(B_2\text{-}B_1)$ in case $A_2=A_1$). By the inductive assumption there is a total ρ_o-chain $p=p_0,p_1,...,p_j=A_2\text{-}\{P\} \leftarrow B_2$ from p to $A_2' \leftarrow B_2$.

If P is a most general atom with respect to $A_2' \leftarrow B_2$ such that $A_2' \cup \{P\} \leftarrow B_2$ is reduced, then by definition of ρ_o, $(A_2 \leftarrow B_2)\epsilon\rho_o(A_2' \leftarrow B_2)$, and the lemma is proved. Otherwise there is an atom Q such that Q is more general than P with respect to $A_2' \leftarrow B_2$, and $A_2' \cup \{Q\} \leftarrow B_2$ is reduced. Let P' be the most general such atom. By definition of ρ_o, $(A_2' \cup \{P'\} \leftarrow B_2)\epsilon\rho_o(A_2' \leftarrow B_2)$. By the choice of P' there is a substitution θ such that $(A_2' \cup \{P'\} \leftarrow B_2)\theta=(A_2' \cup \{P\} \leftarrow B_2)\theta=A_2 \leftarrow B_2$. This implies that θ does not decrease $A_2' \cup \{P'\} \leftarrow B_2$, hence by Lemma 5.15 there is a finite total ρ_o-chain $p_{j+1},p_{j+2},...,p_k$, where $p_{j+1}=(A_2' \cup \{P'\} \leftarrow B_2)$ and $p_k=A_2' \cup \{P\} \leftarrow B_2=A_2 \leftarrow B_2$. Therefore the finite total ρ_o-chain $p_0,p_1,...,p_j,p_{j+1},...,p_k$ satisfies the lemma. ■

Theorem 5.17: $q \leq_o p$ if and only if $p \leq_{\rho_o} q$.

Proof: If $p \leq_{\rho_o} q$ then there exists a finite total ρ_o-chain $p=p_0,p_1,...,p_n=q$ such that $p_{i+1}\epsilon\rho_o(p_i)$, for $0 \leq i<n$. By definition of ρ_o, for every i, $0 \leq i<n$, there is a non-decreasing substitution (possibly empty) θ_i such that $p_i\theta_i \subset p_{i+1}$. Hence $p_i \leq_o p_{i+1}$ for $0 \leq i<n$, and by transitivity of $\leq_o$ it follows that $p \leq_o q$.

If $p \leq_o q$ then there is a substitution θ such that $p\theta \subset q$. By Lemma 5.15 there is a finite total ρ_o-chain $p=p_0,p_1,...,p_k=p\theta$. By Lemma 5.16 there is a finite total ρ_o-chain $p\theta=p_k,p_{k+1},...,p_n=q$, so $p_0,p_1,...,p_k,p_{k+1},...,p_n$ is a finite total ρ_o-chain from p to q, and $p \leq_\rho q$. ■

This establishes that ρ_o is a most general refinement operator, complete for any first order language L.

6. A General, Incremental Model Inference Algorithm.

The method used by the enumerative Algorithm 1 can be viewed as a simpleminded implementation of the Popperian idea of choosing the simplest explanatory conjecture, and holding to it until it is discovered to disagree with some fact. When this happens, one has to look for a replacement to this conjecture, preferably the next simplest conjecture that agrees with the facts. The implementation of this idea in Algorithm 1 is, however, clearly infeasible. A much better implementation can be carried out if, instead of throwing away the old conjecture and looking for a new one from scratch, one can make local modifications to the old set of hypotheses and adapt it to cope with new facts. This section describes how, using the contradiction backtracing algorithm and a refinement operator, one can locally modify conjectures to agree with new facts, and obtain a model inference algorithm that progresses in a piecewise, incremental way. Figure 6-1 sketches such an algorithm.

Figure 6-1: Layout for an Incremental Model Inference Algorithm.

```
Set $T$ to {□}.
repeat
    read 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 facts read so far).
forever.
```

The question whether a conjecture is too strong or too weak is in general undecidable, and we can only approximate it by some resource-bounded computation. Incorporating some fixed recursive function h as the bound on the complexity of derivations results in an algorithm capable of inferring in the limit any h-easy model. Since an algorithm based on this layout is sufficient (see Definition 3.3), Theorem 3.4 (page 13) assures us that such an algorithm can infer in the limit only h-easy models, for some fixed recursive function h, and hence it is the most powerful of its kind.

This algorithm raises another question: which refinements should be added when the theory is too weak? As it turns out, the precise way this idea is implemented is immaterial to the correctness of the algorithm, as long as it is 'exhaustive' in some natural way. In practice, however, this question is crucial, and good heuristics for ordering the addition of refinements to the conjecture may result in a considerable speedup in the convergence of the algorithm.

In the rest of this section we describe a more detailed version of this algorithm, and show that it can infer in the limit any h-easy model. A discussion of some implementation issues follows.

6.1. Approximating the Source Set of a Model.

Corollary 5.10 (page 27) suggests that the right place to look for an L_o-complete axiomatization of a model is in its source set. The source set of a model is, in general, infinite. We show a way of approximating it to any required degree.

Let $G_\rho(L_h,E)$ be the refinement graph of some refinement operator ρ, which is complete for L_h. A *marking* m of G_ρ is a set of sentences of L_h, thought of as marked *'false'*. A marking m of G_ρ is a *consistent marking* if for any q marked *'false'* and any $p \epsilon L_h$ such that $p \leq_\rho q$, p is also marked *'false'*. A marking m is *consistent with* M if it is a consistent marking, and every sentence p marked *'false'* is false in M.

Definition 6.1: Let m be a consistent marking of G_ρ. A sentence $p \epsilon \rho^*$ is called a *source sentence of the marking* m if p is not marked *'false'* and every sentence $q \epsilon \rho^*$ such that $q <_\rho p$ is marked *'false'* in m. The set of source sentences of a marking m is called the *source set of the marking*[2] m, and is denoted L_ρ^m.

If a marking m is consistent with a model M, then L_ρ^m, the source set of the marking is an approximation of L_ρ^M, the source set of the model. Before quantifying this notion of approximation, we show that if m is a consistent marking, then the set $L_\rho^m(k)$ is computable for any k>0. To see this, consider some sentence $p \epsilon L$. If p is marked *'false'* or $size(p)$>k then $p \notin L_\rho^m(k)$. Otherwise consider all sentences q such that $(q,p) \epsilon E$. There are only finitely many of them. If all of them are marked *'false'* then $p \epsilon L_\rho^m$, otherwise $p \notin L_\rho^m$.

Lemma 6.2: Let ρ be a refinement operator over L complete for L_h, $G_\rho(L_h,E)$ the refinement graph of ρ, M a model of L and m a marking of G_ρ consistent with M. Then for any k>0, $L_\rho^m(k) \leq_\rho L_\rho^M(k)$. If, in addition, every sentence of $L_h(k)$ which is false in M is marked *'false'*, then $L_\rho^m(k)=L_\rho^M(k)$. If all false sentences of L_h are marked *'false'* then $L_\rho^m=L_\rho^M$.

Proof: Under the assumptions of the lemma, $L_\rho^m \leq_\rho \{p\}$ for any sentence $p \epsilon L_h$ true in M, by the definition of L_ρ^m. In particular, if $size(p) \leq$k, for some k>0, then $L_\rho^m(k) \leq_\rho \{p\}$. Since L_ρ^M is true in M, it follows that $L_\rho^m(k) \leq_\rho L_\rho^M(k)$.

Assume, in addition, that every sentence of $L_h(k)$ false in M is marked *'false'*. By definition of L_ρ^m, it follows that $L_\rho^m(k)$ is true in M. Since $L_\rho^M \leq_\rho \{p\}$ for any p true in M, it follows that $L_\rho^M(k) \leq_\rho L_\rho^m(k)$. We claim that together with the fact that $L_\rho^m(k) \leq_\rho L_\rho^M(k)$, the equality $L_\rho^m(k)=L_\rho^M(k)$ follows. Clearly if $L_\rho^m(k)=L_\rho^M(k)$ for all k>0, then $L_\rho^m=L_\rho^M$.

Proof of the claim: Let p be a sentence in $L_\rho^M(k)$. Since $L_\rho^m(k) \leq_\rho L_\rho^M(k)$, there is some $q \epsilon L_\rho^m(k)$ such that $q \leq_\rho p$. Since $L_\rho^M(k) \leq_\rho L_\rho^m(k)$, there is $p' \epsilon L_\rho^M(k)$ such that $p' \leq_\rho q$. By the transitivity of $\leq_\rho$ it follows that $p' \leq_\rho p$. By definition of the source set of a model, it is impossible that $p' <_\rho p$, hence $p' = p$, which implies that $p=q$. The symmetric argument applies to a sentence p in $L_\rho^m(k)$, and together they prove the claim. ■

6.2. An Incremental Algorithm.

With these notions we can now describe the algorithm. Algorithm 3 assumes that $<L_o, L_h>$ is an admissible pair of languages, ρ is a refinement operator complete for L_h, conservative with respect to resolution, $\alpha_1,\alpha_2,\alpha_3,...$ a fixed effective enumeration of all sentences of L_o, $F_1,F_2,F_3,...$ an enumeration of some model M of L and h a total recursive function.

We prove that under these assumptions, the following theorem holds:

Theorem 6.3: If M is an h-easy model for L, then Algorithm 3 identifies M in the limit.

A model inference algorithm is said to identify a model in the limit if, after reading in a fact F_n for some n>0, it outputs an L_o-complete axiomatization of M, and never again outputs a different conjecture. We prove that Algorithm 3 identifies M in the limit by showing that if k_0 is the minimal k for which $L_\rho^M(k_0)$ is a finite L_o-complete h-easy axiomatization of M, then Algorithm 3 eventually increases k up to k_0 and does not

[2]The source set of a marking is closely related to the set G of most general patterns that do not match negative instances, in Mitchell's Version Spaces Algorithm [Mitchell 78].

Algorithm 3: An Incremental Model Inference Algorithm.

```
set k to 0, $S_{false}$ to {$\square$} and $S_{true}$ to {}.
mark $\square$ 'false'.
repeat
    read the next fact $F_n = <\alpha, V>$ and add $\alpha$ to $S_V$.
    repeat
        while $L_\rho^m(k) \vdash_n \alpha$ for some $\alpha \in S_{false}$ do
            apply the contradiction backtracing algorithm
            and mark the refuted hypothesis 'false'.
        while $L_\rho^m(k) \nvdash_{h(i)} \alpha_i$ for some $\alpha_i \in S_{true}$ do
            increase k by 1.
    until neither of the while loops is entered.
    output $L_\rho^m(k)$.
forever.
```

increase it further, and that $L_\rho^m(k_0)$ converges pointwise[3] to $L_\rho^M(k_0)$. To do so we establish the following facts about the algorithm, assuming it is applied to an enumeration of an *h*-easy model:

- If $L_\rho^M(k_0)$ is an L_o-complete, *h*-easy axiomatization of *M*, then it does not increase k beyond k_0.
- If k_0 is the minimal k for which $L_\rho^m(k_0)$ is an *h*-easy L_o-complete axiomatization of *M*, then it increases k up to, at least, k_0.
- If k is increased up to, and no more than, k_0, then $L_\rho^m(k_0)$ pointwise converge to $L_\rho^M(k_0)$.

Together these facts prove Theorem 6.3.

Lemma 6.4: Any marking *m* made by Algorithm 3 applied to an enumeration of a model *M* is consistent with *M*.

Proof: We prove the lemma by induction on the number of marks. In the initial marking only $\square$ is marked '*false*'. Since no edge $(q,\square)$ is in *E* this marking is consistent, and since $\square$ is false in any model, this marking is consistent with *M*. A sentence is marked '*false*' by the algorithm only if it is refuted by the contradiction backtracing algorithm. Assume that at some stage the marking of G_ρ is consistent with *M*, and a sentence $p \in L_\rho^m$ is refuted by the contradiction backtracing algorithm. By definition of L_ρ^m, a sentence *p* is in L_ρ^m only if every sentence $q \in L_h$ such that $(q,p) \in E$ is also marked '*false*', hence by the definition of a consistent marking, the marking of *p* in G_ρ as '*false*' is consistent. By the correctness of the contradiction backtracing algorithm this marking is consistent with *M*. ■

Lemma 6.5: If Algorithm 3 is applied to an enumeration of a model *M*, and $L_\rho^M(k_0)$ is an *h*-easy L_o-complete axiomatization of *M*, then the algorithm increases k up to no more than k_0.

Proof: By Lemma 6.4, any marking of the graph G_ρ made by the algorithm is consistent with *M*. Together with Lemma 6.2 this implies that for any $k>0$, $L_\rho^m(k) \preceq_\rho L_\rho^M(k)$. Since ρ is conservative with respect to resolution, after increasing k up to k_0 the condition of the second *while* loop can not be satisfied again, therefore k cannot be increased beyond k_0. ■

[3]A sequence of sets $S_1, S_2, S_3, \ldots$ is said to converge pointwise to a set S if and only if for every $a \in S$ there is an n_1 such that $a \in S_n$ for every $n > n_1$, and for every $a \notin S$ there is an n_2 such that $a \notin S_n$ for every $n > n_2$.

Lemma 6.6: Algorithm 3 applied to an enumeration of an h-easy model M eventually reads in all the facts.

Proof: We show that if M is an h-easy model, then the inner *repeat* loop terminates on every new fact. Assume a new fact F_n is read, for some $n>0$; we count the total number of times the two *while* loops can be executed.

Since M is an h-easy model of L, it has some finite L_o-complete h-easy axiomatization. Let T be such an axiomatization of M, that is, for any sentence $\alpha_i \epsilon L_o^M$. $T \vdash_{h(i)} \alpha_i$. Let k_0 be the maximum size of any sentence of T. Since ρ is conservative with respect to resolution, by corollary 5.10, $L_\rho^M(k_0)$ is also an h-easy L_o-complete axiomatization of M.

By Lemma 6.5, k is not increased beyond k_0, hence the second *while* loop can be executed only finitely many times. Since $L_h(k_0)$ is finite, at any given time $L_\rho^m(k) \subset L_h(k_0)$, and every iteration of the first *while* loop marks at least one sentence of $L_h(k_0)$ *'false'*, it follows that the first *while* loop can be executed only finitely many times. Hence any iteration of the inner *repeat* loop terminates, and it is executed for only finitely many iterations. ■

Almost the same argument shows that Algorithm 3 reads in all the facts when applied to an enumeration of *any* model. The reason is that at any given point the set of S_{true} of true observational sentences define an h-easy (almost everywhere false) model of L.

Lemma 6.7: If for some $k_0>0$, Algorithm 3 increases k up to and no more than k_0, then $L_\rho^m(k_0)$ converges in the limit pointwise to $L_\rho^M(k_0)$.

Proof: By Lemma 6.2 it is sufficient to prove that for any sentence $p \epsilon L_h$ of size$<k_0$, if p is false in M then eventually it will be marked *'false'*. We show that if $L_\rho^m(k_0)$ contains a false sentence (that is, $L_\rho^m(k_0) \neq L_\rho^M(k_0)$), then eventually the contradiction backtracing algorithm will be applied, and some sentence $p \epsilon L_\rho^m(k_0)$ will be marked *'false'*. Since for any marking m, $L_\rho^m(k_0) \subset L_h(k_0)$ and the latter is finite, this process can happen only finitely many times, and eventually all false sentence in $L_h(k_0)$ will be marked.

Consider some marking m made by the algorithm, after k is increased up to k_0. If $L_\rho^m(k_0)=L_\rho^M(k_0)$ the lemma holds, so assume, by way of contradiction, that $L_\rho^m(k_0)$ contains a false sentence, and m does not change anymore. Since $L_\rho^m(k_0) <_\rho L_\rho^M(k_0)$ and $L_\rho^M(k_0)$ is an L_o-complete axiomatization of M, it follows that $L_\rho^m(k_0) \vdash L_o^M$, where L_o^M is the set of observational sentences true in M. By the admissibility requirement (page 9), $L_\rho^m(k_0)$ has a witness for its falsity $\alpha_i \epsilon L_o$ for some $i>0$, that is, $L_\rho^m(k_0) \vdash_j \alpha_i$ for some $j>0$, and α_i is false in M. Let $n=\max\{i,j\}$. Then $L_\rho^m(k_0) \vdash_n \alpha_i$, and after reading the nth fact F_n, the set S_{true} contains α_i. At this point the condition of the first *while* loop is satisfied, the contradiction backtracing algorithm is called, refuting some $p \epsilon L_\rho^m(k_0)$, which results in marking p *'false'*, in contradiction to the assumption that m does not change. Therefore $L_\rho^m(k_0)$ pointwise converges to $L_\rho^M(k_0)$. ■

Lemma 6.8: If k_0 is the minimal k for which $L_\rho^m(k_0)$ is an h-easy L_o-complete axiomatization of M, then Algorithm 3 increases k up to at least k_0.

Proof: Assume, by way of contradiction, that Algorithm 3 applied to an enumeration of an h-easy model M increases k up to, and no more than, some k' for which $L_\rho^M(k')$ is not an h-easy axiomatization of M. By Lemma 6.7, $L_\rho^m(k')$ pointwise converges to $L_\rho^M(k')$. Since, by choice of k', $L_\rho^M(k')$ is not an h-easy L_o-complete axiomatization of M, there is an $\alpha_j \epsilon L_o$, for some $j>0$, such that $L_\rho^M(k') \not\vdash_{h(j)} \alpha_j$. Let n be the least index i such that the algorithm increases k up to k' before reading F_i, and let n_0 be $\max\{n,j\}$. Then after reading the n_0th fact, $\alpha_j \epsilon S_{true}$ and $k=k'$, hence the condition of the second while loop is satisfied, and k is increased beyond k', in contradiction to our assumption. Hence k is increased up to, at least, k_0, and the lemma is proved. ■

Proof of Theorem 6.3: Let M be an h-easy model of L, k_0 the minimal $k>0$ such that $L_\rho^M(k_0)$ is an h-easy, L_o-complete axiomatization of M. Such a k_0 exits by corollary 5.10. Assume that Algorithm 3 is applied to an enumeration of M.

By Lemmas 6.5 and 6.8, algorithm 3 increases k up to, and no more than, k_0. By Lemma 6.7 $L_\rho^m(k_0)$ pointwise converges to $L_\rho^M(k_0)$. Since $L_\rho^M(k_0)$ is finite, there is some $n>0$ such that after Algorithm 3 reads in the nth fact, $L_\rho^m(k)=L_\rho^M(k_0)$, and both m and k do not change anymore. Hence after reading in the nth fact Algorithm 3 outputs an h-easy L_o-complete axiomatization of M, and never again output a different conjecture. In other words, Algorithm 3 identifies M in the limit. ■

By proving Theorem 6.3 we have established that Algorithm 3 is as powerful as Algorithm 1, and, since Algorithm 3 is sufficient, Theorem 3.4 says that it is the most powerful of its kind.

6.3. Some Implementation Issues.

The specialization of Algorithm 3 to the inference of Horn theories and its implementation in Prolog will be discussed in a future paper. Here we just point out the major issues involved in such an endeavor.

One issue that has to to be addressed to implement the algorithm efficiently is how to compute quickly the effect of adding or removing hypotheses from the conjecture. The problem is to find an efficient way to test whether the removal of an hypothesis from the conjecture (by marking it '*false*'), results in the unprovability of some sentence in S_{true}, previously provable from the hypotheses, and to test whether the addition of an hypotheses to the conjecture (by increasing k) results in some sentence in S_{false}, previously unprovable from the hypotheses, now being provable under the current complexity bound.

The first of these questions can be solved by maintaining *logical dependencies* between the hypotheses and the facts, that is, recording which hypotheses one used in the derivations of which facts (see [Charniak et al. 80] for an ensemble of implementation techniques for this task). I do not know of a general efficient solution to the second question, which avoids trying again to prove all the sentences in S_{false}, although The Model Inference System incorporates some heuristics that apply in case the hypotheses are Horn clauses.

Optimizing the number of hypotheses in the conjecture is another problem to be solved. The model inference algorithm is guaranteed to minimize the maximal size of the hypotheses in the conjecture. The conjecture may contain many superfluous hypotheses that do not increase its logical power. This behavior also may influence the efficiency of the algorithm, since the complexity of the tests in the condition of the *while* loops grows with the number of hypotheses in the conjecture. There is a simple algorithm to locally optimize the conjecture, in case the hypotheses are Horn clauses. The version of the Model Inference System on which the statistics were made does not yet incorporate the full optimizer, so the reader can find, in some cases, superfluous hypotheses in the final conjecture produced by it.

Another issue to be addressed is the amount of information the oracle (user) is allowed to supply the system. For example, if the system knows that in the predicate sort(X,Y) to be inferred the type of the variables is "list of integers", the generation of many syntactically correct but semantically wrong axioms can be avoided by the refinement operator. Type specification is not yet incorporated in the Model Inference System.

There are many heuristics that can improve the efficiency of the model inference algorithm, in terms of time and the number of facts needed for convergence. For example, if the pair of hypotheses language and observational language is decidable, i.e., the question whether $T \vdash \alpha$ is decidable for any $T \subset L_h$ and any $\alpha \in L_o$, then the complexity bound h can be ignored. In case L_h contains Horn sentences of restricted form (such as all the classes described in Section 5) and the proof procedure is backward chaining (a restricted form of resolution, used by Prolog), a simple test for duplicate occurrences of goals on the stack will do. These and related topics will be discussed more fully in a future paper.

7. 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 surpasses existing systems for inductive inference and program synthesis from examples. I believe that these encouraging results where made possible through using first order logic as the underlying model of computation.

Here are some of the reasons for the success of logic as a media 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 the contradiction backtracing algorithm.

Logic has an intimate relation between its syntax and semantics. For example, replacing a variable by a term, or unifying two variables in a sentence, always results in a logically weaker (or equivalent) sentence. The same is true for adding atoms to the condition or the conclusion of a sentence. 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: if you add axioms you have more consequences (input-output relations); if you remove axioms you have fewer consequences (input-output relations). There are not many practical programming languages for which such syntactic alterations to a program have predictable effects on what it computes.

As a programming language, logic features separation of 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 [Kowalski 79b], and leaves the "control component" unspecified. The logic component of a program contains more than its specification, as the differences between the quicksort logic program (page 22) and the insertion sort program (page 26) show. 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. Such a division of labor is easier to carry out if the optimization task is a mapping between well defined classes of objects. Such is the case with the mapping induced by imposing control on logic programs.

As a programming language, logic features equivalence of program and data. Logical facts can readily be part of the logic program to be inferred. This sense of equivalence is stronger than the one used to describe other programming languages, like Lisp. This equivalence has not been used, so far, in the model inference algorithm or in the Model Inference System, in order to focus on the more theoretical issues of inductive inference. However, in a system which interacts more gracefully with the user, in the spirit of McCarthy's Advice Taker [McCarthy 63], such a property might prove to be extremely useful. Such a system would be oriented not only to the inference of theories from facts but also to the interactive creation, acquisition and debugging of theories from human advice. A uniform language for specifying "low level" details (truth of ground atoms), and "high level" advice (proposed hypotheses) would enable devising algorithms and systems that treat uniformly all human advice.

Acknowledgements.

The development of the ideas in this paper spans into the past a bit further than NSF grant MCS8002447, which I acknowledge.

While in high school, Moshe Kroy introduced me to Logic and Philosophy of Science. His exciting lectures on Popper's philosophy were a traumatic experience in my intellectual development. Dr. Kroy suggested that if we could test epistemological theories and theories of the mind, then they would become science, rather than metaphysics, and that it might be possible to cary out this task with a computer. This may be the reason for my leaving the homeland, philosophy, for the promised land of computer science.

Years later, by then an undergrad in Tel Aviv University, Amiram Yehudai told me that I did not have to invent Inductive Inference, since such a discipline already existed. He pointed me to the Blums' paper, which Jonathan Stavi helped me to decipher. Prof. Stavi also helped me relate Gold's and Blums' results to inductive inference in logic.

At Yale, I could not have been luckier having Dana Angluin as my advisor. Truly thanking her would make this acknowledgement longer than the paper, so I reserve the honor to for a forthcoming, lengthier document. I will just say that there is nothing more trustable than a proof Dana says is OK. If anything in this paper appears to be to the contrary, it is probably because I rewrote it afterwards.

Drew Mcdermott provided crucial insights into logic hacking, in moments of baffle and bewilderment. Although he prefered my system learning about arches and non-arches, rather than inferring atomic-complete axiomatizations of dense partial orders, he was, nevertheless, encouraging all the way.

Bob Kowalski warned me, just in time, that there will be Sirens in my way, singing "I am a cute, good-looking, interesting, quick-shot, irrelevant theoretical question", and that I should never succumb. So whenever I have imagined hearing one, I plugged my ears and did some more Prolog hacking, until it was over. I have failed only once (twice?), and this is a lesson I will never forget. At least until next time.

Even if I have disturbed him in midst of the most intense hack, Bob Nix was always willing to answer any question, be it on English grammar or combinatorics, or listen to my latest new idea. He read the final draft very carefully, and made sure that there were no errors left.

Ernie Davis and Chris Riesbeck read a previous draft, and provided audible feedback. The fact that even Ernie got lost in Section 6 was a conclusive evidence to me that I should rewrite it. I hope it passes Ernie's Test this time.

Without the outstanding implementation of Prolog we have received from David Warren at Edinburgh, this work would have remained yet another theoretical exercise. The pace with which their system matured into a practical and efficient programming tool can not be attributed to the wonders of Logic alone.

Finally, our Tools group provided a comfortable environment for both hacking and writing, and I am thankful for that. Special thanks to Steve Wood, whom I bugged until Z did (almost) exactly what I wanted, and to John Ellis, who wrote the Scribe interface to the typesetter, and then answered a hundred questions about how to use it.

I. Performance of the Model Inference System

The Model Inference System is implemented in the programming language Prolog [Pereira et al. 78]. It was developed hand in hand with the algorithms, problems in the former motivating improvements in the later. The system has had four stable incarnations so far. The first inferred ground complete axiomatizations of regular sets; the second inferred atomic complete axiomatizations of such sets; and the third inferred atomic complete axiomatizations of a first order language with an arbitrary alphabet, provided the axioms were context-free transformations. The fourth incarnation of the system enabled the inclusion of more general refinement operators, and could infer multiple context-free Horn sentences and term-free transformations with auxiliary predicate. So far it has succeeded in inferring atomic complete axiomatizations of dense partial order with endpoints, binary tree isomorphism, list reversal and list subset, multiplication and exponentiation, and satisfiability of boolean formulas.

A fifth implementation is on its way, featuring the possibility of concurrent inference of arbitrarily many programs; typing specification, to be used by the refinement operators to restrict the generation of semantically wrong hypotheses; an algorithm that locally optimizes the conjecture, and improved heuristics for adding hypotheses and minimizing the number of queries (experiments).

Following are examples of application of the different incarnations of the system, with some relevant statistics. In the first two incarnations the programs were all interpreted under the Edinburgh Prolog-10 interpreter version 1.32. In the third and fourth the programs were compiled using the in-core compiler of the Edinburgh Prolog-10 version 3. The computer is a DECsystem-2060, running Prolog-10 under the compatibility package PA1050. In the following, the running time in CPU seconds measured the time the system took to converge to a correct and sufficient axiomatization of the intended model. The decision to terminate the inference process was made by me, since it is an inherent property of an inductive inference algorithm that *it* can never tell whether it has converged. Note that since this decision was subjective, there may be some errors in the axiomatization below (I am aware of one). The reader is encouraged to find them. *Facts* are the number of facts read therein, and *hypotheses generated* is the number of hypotheses generated by the refinement operator during the inference.

In the earlier versions of the system it would simply go through all possible atoms, in increasing order of complexity, asking the user (or a built-in oracle for the model, when the user was tired) whether they are true in the model. The inefficiency of this approach became more evident as the models got more and more complex, since true atoms are scarce in such models. Therefore in later versions of the system the user is capable of specifying some initial set of facts about the models, hence these implementations were more "fact efficient".

I.1 Inferring Regular Sets.

The first order language for this task consisted of the two predicates in(X) and out(X). The terms of the language were the null string Λ, and strings constructed from a variable or Λ and the successor functions 0(X) and 1(X). For convenience these where simulated by lists of 0's and 1's. The hypotheses language contained Horn clauses, where the only structural restriction on them was that the size of the terms in the condition would not exceed the size of the term in the conclusion. This restriction releases us from a need to impose complexity bound on the size of the proofs. In the following axiomatizations, the number associated with an axiom is its sequential number in the order the axioms were generated by the refinement operator.

Inferring Ground Complete Axiomatizations of Regular Sets.

Strings of even number of 1's and even number of 0's.

157 CPU seconds, 64 facts, 613 hypotheses generated.

```
(in(Λ) ← true), 3
(out(0) ← true), 9
(out(1) ← true), 23
(out(0X) ← in(X)), 12
(in(00) ← true), 37
(out(0X) ← in(1X)), 14
(in(11) ← true), 76
(out(1X) ← in(X)), 26
(in(00X) ← in(X)), 40
(out(00X) ← out(X)), 52
(out(01X) ← in(X)), 171
(out(01X) ← in(0X)), 173
(out(11X) ← out(X)), 91
(in(01X) ← in(10X)), 191
(out(1X) ← in(0X)), 28
(in(10X) ← in(01X)), 70
(out(10X) ← out(01X)), 289
(in(11X) ← in(X)), 79
(in(010X) ← in(001X)), 219
(in(011X) ← in(0X)), 305
```

Strings of even parity.

16 CPU seconds, 20 facts, 102 hypotheses generated.

```
(in(Λ) ← true), 3
(in(0) ← true), 9
(out(1) ← true), 23
(in(0X) ← in(X)), 12
(out(0X) ← out(X)), 20
(out(1X) ← in(X)), 26
(in(1X) ← out(X)), 34
```

Strings that contain 11 as a substring.

41 CPU seconds, 50 facts, 141 hypotheses generated.

```
(out(Λ) ← true), 3
(out(1) ← true), 6
(in(11X) ← true), 18
(out(0) ← true), 23
(out(0X) ← out(X)), 27
(in(0X) ← in(X)), 45
(out(10) ← true), 72
(out(10X) ← out(X)), 76
(in(1X) ← in(X)), 19
```

Strings that contain 11 or 00 as a substring.

25 CPU seconds, 31 facts, 188 hypotheses generated.

```
(out(Λ) ← true), 3
(out(0) ← true), 6
(in(00X) ← true), 17
(out(1) ← true), 23
(in(11X) ← true), 32
(in(0X) ← in(X)), 19
(out(10) ← true), 67
(in(1X) ← in(X)), 33
(out(10X) ← out(0X)), 79
(out(01X) ← out(1X)), 49
```

Strings that are of even number of ones or contain 00 as a substring.

64 CPU seconds, 69 facts, 383 hypotheses generated.

```
(in(Λ) ← true), 3
(in(0) ← true), 9
(in(00X) ← true), 10
(out(1) ← true), 23
(in(1X) ← out(X)), 34
(in(0X) ← in(X)), 12
(out(10) ← true), 73
(in(100X) ← true), 93
(out(01) ← true), 37
(out(01X) ← out(1X)), 49
(in(11X) ← in(X)), 53
(out(11X) ← out(X)), 111
(in(10X) ← in(01X)), 98
(out(101X) ← out(X)), 175
```

Inferring Atomic Complete Axiomatizations of Regular Sets.

The inefficiency of inferring ground complete axiomatizations of regular sets was due not so much to the algorithm as to the fact that I was looking at the wrong problem. Logical theories are essentially non-deterministic computational devices. A non-deterministic Turing machine has the privilege of succeeding in a computation only when it is accepting a string. For rejecting a string it merely has to fail to accept it, and there is no need for it to explicitly succeed in a rejecting computation, a requirement that a ground complete theory of a set has to satisfy. This consideration naturally led to the reformulation of the regular set inference problem: find an *atomic complete* axiomatization of the intended model, that is an set of axioms that prove in(s) for every s in the intended regular set, and fail to prove in(s) for strings that are not in that set. The first order language for this task consists of only one predicate in(X), with the same terms as before.

The results of the previous application of the system suggested that regular sets can be axiomatized with transformations only. Therefore the hypothesis language was restricted to such axioms. Applying the system to the inference of atomic-complete axiomatizations of the regular sets as before led to remarkable improvements in the running time of the system, the number of facts needed and the number and complexity of the axioms found.

Strings of even number of 1's and even number of 0's.

28 CPU seconds, 35 facts, 75 hypotheses generated.

```
(in(Λ) ← true), 4
(in(00X) ← in(X)), 13
(in(11X) ← in(X)), 19
(in(010X) ← in(1X)), 41
(in(011X) ← in(0X)), 48
(in(100X) ← in(1X)), 61
(in(101X) ← in(0X)), 67
```

Strings of even parity.

8 CPU seconds, 17 facts, 29 hypotheses generated.

```
(in(Λ) ← true), 4
(in(0X) ← in(X)), 5
(in(11X) ← in(X)), 24
(in(10X) ← in(1X)), 15
```

Strings that contain 11 as a substring.

11 CPU seconds, 35 facts, 12 hypotheses generated.

```
(in(11X) ← true), 7
(in(0X) ← in(X)), 9
(in(1X) ← in(X)), 5
```

Strings that contain 11 or 00 as a substring.

14 CPU seconds, 52 facts, 12 hypotheses generated.

```
(in(00X) ← true), 6
(in(11X) ← true), 11
(in(0X) ← in(X)), 5
(in(1X) ← in(X)), 9
```

Strings that are of even number of ones or contain 00 as a substring.

10 CPU seconds, 18 facts, 24 hypotheses generated.

```
(in(Λ) ← true), 4
(in(0X) ← in(X)), 5
(in(00X) ← true), 6
(in(100X) ← true), 16
(in(11X) ← in(X)), 19
(in(10X) ← in(1X)), 15
```

I.2 Inferring Atomic Complete Axiomatizations of Simple Models.

In the first two stages of the development of the system described above, the first order language used by the system was fixed. In the third stage the language became part of the input to the system. The input is the function and constant symbols of L, and the predicate $p(X_1,X_2,...,X_l)$ to be inferred. This version used the refinement operator ρ_2 (page 24), which is complete for atoms and context-free transformations. The fourth version incorporated some more general refinement operators, which were complete for multiple context-free Horn sentences and term-free transformations with auxiliary predicate. In case the system inferred an axiomatization that uses an auxiliary predicate (such as times, which uses plus), the predicate to be used was also specified in the input. The statistics below are on the performance of this version. The two numbers following the axioms are their sequential numbers as they were generated by the refinement operator, and the number of refinement operations needed to generate them.

Arithmetic.

Ordering Relation.

le(X,Y) ← X is less than or equal to Y.
2 CPU seconds, 10 facts, 14 hypotheses generated.

```
(le(X,s(Y)) ← le(X,Y)), 10, 2
(le(X,X) ← true), 1, 1
(le(0,X) ← true), 2, 1
```

Addition.

plus(X,Y,Z) ← X plus Y is Z.
5 CPU seconds, 13 facts, 67 hypotheses generated.

```
(plus(X,X,0) ← true), 11, 1
(plus(0,X,X) ← true), 14, 1
(plus(s(X),Y,s(Z)) ← plus(X,Y,Z)), 52, 3
```

Multiplication, using addition.

times(X,Y,Z) – X times Y is Z.
20 CPU seconds, 13 facts, 205 hypotheses generated.

```
(times(0,X,0) – true), 12, 1
(times(s(X),Y,Z) – times(X,Y,W),plus(W,Y,Z)), 61, 2
```

Exponentiation, using multiplication.

exp(X,Y,Z) – X to the Y is Z.
28 CPU seconds, 18 facts, 252 hypotheses generated.

```
(exp(X,s(Y),Z) – exp(X,Y,W),times(W,X,Z)), 129, 2
(exp(X,0,s(0)) – true), 78, 2
(exp(0,s(X),0) – true), 148, 2
```

Dense Partial Order with Endpoints.

Partial ordering.

le(X,Y) – X to less than or equal to Y.
10 CPU seconds, 60 facts, 61 hypotheses generated.
Note: m(X,Y) is interpreted as some point between X and Y.

```
(le(X,X) – true), 7, 1
(le(X,m(X,1)) – true), 21, 3
(le(X,1) – true), 5, 1
(le(0,X) – true), 1, 1
(le(m(X,Y),Z) – le(Y,Z),le(X,Z)), 57, 4
(le(m(X,Y),Y) – le(X,Y)), 56, 3
(le(m(X,Y),X) – le(Y,X)), 58, 3
(le(m(X,0),X) – true), 49, 3
```

List Processing Programs.

Member.

member(X,Y) – X is a member of the list Y.
2 CPU seconds, 23 facts, 13 hypotheses generated.

```
(member(X,[Y|Z]) – member(X,Z)), 6, 3
(member(X,[X|Y]) – true), 7, 3
```

Prefix.

prefix(X,Y) – X is a prefix of Y.
4 CPU seconds, 17 facts, 52 hypotheses generated.

```
(prefix([],X) – true), 1, 1
(prefix([X|Y],[X|Z]) – prefix(Y,Z)), 50, 5
```

Suffix.

suffix(X,Y) – X is a suffix of Y.
3 CPU seconds, 17 facts, 26 hypotheses generated.

```
(suffix(X,[Y|Z]) – suffix(X,Z)), 19, 3
(suffix(X,X) – true), 5, 1
(suffix([],X) – true), 1, 1
```

Subsequence.

subsequence(X,Y) — X is a subsequence of Y.
8 CPU seconds, 50 facts, 63 hypotheses generated.

```
(subsequence(X,[Y|Z]) — subsequence(X,Z)), 22, 3
(subsequence(X,[Y|X]) — true), 20, 3
(subsequence([],X) — true), 1, 1
(subsequence([X|Y],[X|Z]) — subsequence(Y,Z)), 51, 5
(subsequence([X],[X|Y]) — true), 45, 5
```

Subset, using member.

subset(X,Y) — X is a subset of Y.
31 CPU seconds, 82 facts, 152 hypotheses generated.

```
(subset(X,[Y|Z]) — subset(X,Z)), 34, 3
(subset(X,[Y|X]) — true), 32, 3
(subset([],X) — true), 1, 1
(subset([X|Y],Z) — subset(Y,Z),member(X,Z)), 113, 3
(subset([X|Y],[X|Z]) — subset(Y,Z)), 103, 5
(subset([X],[X|Y]) — true), 97, 5
```

Append.

append(X,Y,Z) — Z is the result of appending X to Y.
10 CPU seconds, 33 facts, 106 hypotheses generated.

```
(append(X,[],X) — true), 12, 1
(append([],X,X) — true), 11, 1
(append([X|Y],Z,[X|W]) — append(Y,Z,W)), 99, 5
```

Conc.

conc(X,Y,Z) — Z is the result of concatenating the symbol Y to the list X.
7 CPU seconds, 29 facts, 66 hypotheses generated.

```
(conc([],X,[X]):-true), 26, 3
(conc([X|Y],Z,[X|W]):-conc(Y,Z,W),true), 64, 5
```

Reverse, using conc.

reverse(X,Y) — X is the reverse of Y.
6 CPU seconds, 13 facts, 104 hypotheses generated.
Note: The axiomatization of reverse using append (page 4) is not term-free, hence could not be inferred using the refinement operators currently implemented.

```
(reverse([],[]) — true), 6, 1
(reverse([X|Y],Z) — reverse(Y,U),conc(U,X,Z)), 32, 3
```

Last.

last(X,Y) — X is the last element in the list Y.
4 CPU seconds, 12 facts, 18 hypotheses generated.

```
(last(X,[X]) — true), 17, 3
(last(X,[Y|Z]) — last(X,Z)), 8, 3
```

Double every second element.

dbl2nd(X,Y) — Y is the list X, where every other element occurs twice.

64 CPU seconds, 49 facts, 229 hypotheses generated.

```
(dbl2nd([],[]) ← true), 6, 1
(dbl2nd([X,Y|Z],[X,Y,Y|W]) ← dbl2nd(Z,W)), 222, 11
(dbl2nd([X],[Y]) ← true), 97, 5
```

Iota, or tails.

iota(X,Y) ← Y is the list of tails of X.
18 CPU seconds, 30 facts, 149 hypotheses generated.

```
(iota([],[]) ← true), 6, 1
(iota([X|Y],[[X|Y]|Z]) ← iota(Y,Z)), 100, 7
(iota([X|Y],[[X|Z]]) ← iota(Y,Z)), 97, 7
```

Pack, or one level flatten, using append.

pack(X,Y) ← Y is the list of elements of the lists of X.
9 CPU seconds, 24 facts, 105 hypotheses generated.

```
(pack(X,[]) ← true), 3, 1
(pack([X|Y],Z) ← pack(Y,W),append(X,W,Z)), 36, 3
```

Oddp.

Oddp(X) ← X is of odd length.
3 CPU seconds, 13 facts, 17 hypotheses generated.

```
(oddp([X,Y|Z]) ← oddp(Z)), 14, 5
(oddp([X]) ← true), 5, 3
```

Pair.

pair(X,Y,Z) ← Z is a list of pairs of elements of X and Y.
62 CPU seconds, 61 facts, 226 hypotheses generated.

```
(pair(X,[],X) ← true), 16, 1
(pair([X|Y],[Z|W],[p(X,Z)|U]) ← pair(Y,W,U)), 211, 9
```

Unlabeled Binary tree predicates

Subtree relation.

subtree(X,Y) ← X is a subtree of Y.
9 CPU seconds, 40 facts, 32 hypotheses generated.
Note: t(X,Y) is interpreted as a binary tree whose left subtree is X and whose right subtree is Y.

```
(subtree(X,t(Y,Z)) ← subtree(X,Y)), 23, 3
(subtree(X,t(X,Y)) ← true), 20, 3
(subtree(X,t(Y,Z)) ← subtree(X,Z)), 24, 3
(subtree(X,t(Y,X)) ← true), 21, 3
(subtree(X,X) ← true), 5, 1
(subtree(0,X) ← true), 1, 1
```

Tree isomorphism.

isotree(X,Y) ← X is isomorphic to Y.
20 CPU seconds, 110 facts, 117 hypotheses generated.
Note: t(X,Y) is interpreted as a binary tree whose left subtree is X and whose right subtree is Y.

```
(isotree(X,X) ← true), 5, 1
(isotree(t(X,Y),t(Z,W)) ← isotree(Y,W),isotree(X,Z)), 108, 6
(isotree(t(X,Y),t(Z,Y)) ← isotree(X,Z)), 103, 5
(isotree(t(X,Y),t(Z,W)) ← isotree(Y,Z),isotree(X,W)), 41, 6
(isotree(t(X,Y),t(Z,X)) ← isotree(Y,Z)), 44, 5
(isotree(t(X,Y),t(Y,Z)) ← isotree(X,Z)), 38, 5
(isotree(t(X,Y),t(Y,X)) ← true), 35, 5
```

Satisfiability of Boolean Formulas.

Satisfiability, using unsatisfiability.

satis(X) ← X is satisfiable.
10 CPU seconds, 40 facts, 66 hypotheses generated.

```
(satis(1) ← true), 2, 1
(satis(not(X)) ← unsatis(X)), 14, 3
(satis(not(0)) ← true), 7, 2
(satis(and(X,Y)) ← satis(Y),satis(X)), 54, 4
(satis(and(1,X)) ← true), 39, 3
(satis(or(X,Y)) ← satis(X)), 33, 3
(satis(or(X,Y)) ← satis(Y)), 35, 3
(satis(or(X,1)) ← true), 28, 3
(satis(or(1,X)) ← true), 23, 3
```

Insatisfiability, using satisfiability.

unsatis(X) ← X is unsatisfiable.
10 CPU seconds, 45 facts, 67 hypotheses generated.

```
(unsatis(0) ← true), 1, 1
(unsatis(not(X)) ← satis(X)), 14, 3
(unsatis(not(1)) ← true), 8, 2
(unsatis(and(X,Y)) ← unsatis(X)), 63, 3
(unsatis(and(X,Y)) ← unsatis(Y)), 65, 3
(unsatis(and(X,0)) ← true), 57, 3
(unsatis(and(0,X)) ← true), 52, 3
(unsatis(or(X,Y)) ← unsatis(Y),unsatis(X)), 39, 4
(unsatis(or(0,0)) ← true), 38, 3
```

II. Glossary of Symbols.

$\circ$	function composition
$\leftarrow$, $P \leftarrow Q$	implication, P if Q
$\rightarrow$	covering relation
$\subset$	subset or equal
$\subsetneq$	strict subset
$\cup$	set union
$\cap$	set intersection
$\geq$	greater than or equal to
$\leq, \nleq$	{not} less than or equal to
&	and
$\neq$	not equal
$\sim$	not
$\vdash$, $p \vdash q$, $\nvdash$	p {not} derives q
$\vdash_n$, $\nvdash_n$, $p \vdash_n q$	p {not} derives q in n derivation steps or less
$\square$	the empty sentence
$\cong, \ncong$	{not} equivalent
α, α_1	observational sentences
θ, θ_1	substitutions
$\epsilon, \notin$	set membership
Φ_k	the kth step counting function
Λ	the null string
ρ, ρ_1, ρ_0	refinement operators
$\leq_\rho, <_\rho, \nleq_\rho, \nless_\rho$	partial order induced by ρ
$\rho(p)$	the refinements of p generated by ρ
$\rho^*(p)$	the transitive closure of p under ρ
ρ^*	the transitive closure of $\square$ under ρ
G_ρ	the refinement graph induced on ρ^* by ρ
$\leq_o, <_o, \nleq_o, \nless_o$	a partial order defined on sentences
L	a first order language
L_h	an hypothesis language
L_o	an observational language
L_o^M	observational sentences true in M
M	a model
L_ρ^M	the source set of a model
m	a marking
L_ρ^m	the source set of a marking
p, p_1, q_1	sentences
T, S, T_1, T_n	sets of sentences
A_1, A_2, B_1, B_2	sets of atoms
P, P_1, Q, Q_1	predicates
t, f, t_1, f_1	terms
X, Y, X_1, Y_1	variables
$h, h(n)$	complexity bound

III. A Useless Lemma.

During the attempts to prove that ρ_0 (defined in section 5.3) is a most general refinement operator, I proved the following lemma. This lemma turned out not to be of any use in the proof. However, since I worked so hard to find its right formulation, and then to prove it, I feel the paper will not be complete without it.

Lemma 7.1: Let $S'=S \cup \{P\}$ be a set of atoms such that S is reduced and $P \notin S$. Then S' is not reduced if and only if exactly one of the following holds:

1. There is a substitution θ and an atom $Q \in S$ such that $Q\theta=P$, $P\theta=P$ and $S'\theta \subset S'-\{Q\}$.

2. There is substitution θ such that $P\theta \in S$ and $S\theta \subset S$, or, in other words, $S'\theta \subset S$.

Proof: The *if* direction is a straightforward application of the definition of a reduced set: if one of the cases holds for some substitution θ then $S'\theta \subsetneq S'$, and S' is not reduced.

To prove the *only if* direction, assume that $S \cup \{P\}$ is not reduced. Then there there is a substitution θ such that $S'\theta \subsetneq S'$. There are two cases:

Case 1: $P=P\theta^k$ for some $k>0$. Since $S'\theta \subsetneq S'$ implies $S'\theta^k \subsetneq S'$, at least one atom $Q \in S$ is mapped into P by θ^k, that is $Q\theta^k=P$, otherwise $S\theta^k \subsetneq S$, in contradiction to the assumption that S is reduced. The substitution θ^k satisfies the first clause of the lemma.

Case 2: $P \neq P\theta^k$ for all $k>0$. If no $Q \in S$ is mapped into P by θ, that is $Q\theta \neq P$ for all $Q \in S$, then θ satisfies the second clause of the lemma. Otherwise, let $Q_1, Q_2, \ldots, Q_n$ be all the atoms in S mapped into P by some multiple of θ, that is, $Q_i\theta^{k_i}=P$ for some $k_i>0$. For every such atom Q_i, k_i is the only multiple of θ by which it is mapped into P, otherwise there is a k such that $P\theta^k=P$, contrary to the assumption of this case. Let $k = \max\{k_1,k_2,\ldots,k_n\}+1$. Then $P \notin S'\theta^k$, hence $S'\theta^k \subset S$, and the second clause of the lemma holds. ■

References

[Angluin & Hoover 80]
Dana Angluin and Douglas N. Hoover.
Regular Prefix Relations.
Submitted.
1980.

[Angluin & Shapiro 81]
Dana Angluin and Ehud Shapiro.
Structural Complexity vs. Expressive Power of Logic Programs.
In preparation.
1981.

[Angluin 78] Dana Angluin.
On the complexity of minimum inference of regular sets.
Information and Control 39:337-350, 1978.

[Bierman 78] Alan W. Bierman.
The Inference of Regular Lisp Programs from Examples.
IEEE transactions on Systems, Man, and Cybernetics 8, August, 1978.

[Blum & Blum 75]
Lenore Blum and Manuel Blum.
Towards a Mathematical Theory of Inductive Inference.
Information and Control 28, 1975.

[Case & Smith 81]
Case J. and Smith C.
Comparison of Identification Criteria for Mechanized Inductive Inference.
to appear.
1981.

[Charniak et al. 80]
E. Charniak, C. K. Riesbeck, D. V. McDermott.
Artificial Intelligence programming.
Lawrence Earlbaum Associates, Hillsdale, New-Jersy, 1980.

[Duhem 54] Pierre Duhem.
The Aim and Structure of Physical Theory.
Princeton University Press, 1954.
Originally published in French in 1906.

[Gold 65] E. Mark Gold.
Limiting Recursion.
Journal of Symbolic Logic 30, 1965.

[Gold 67] E. M. Gold.
Language identification in the limit.
Information and Control 10:447-474, 1967.

[Gold 78] E. Mark Gold.
Complexity of Automaton Identification From Given Data.
Information and Control 37:302-320, 1978.

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

[Green 69] C. Cordell Green.
Theorem Proving by Resolution as a Basis for Question Answering.
In B. Meltzer and D. Mitchie, editor, *Machine Intelligence 4*, pages 183-205. Edinburgh University Press, Edinburgh, 1969.

[Harding 76] Sandra G. Harding (ed.).
Can Theories be Refuted? Essays on the Duhem-Quine Thesis.
D. Reidel Publishing Company, 1976.

[Klette and Weihagen 80]
R. Klette and R. Wiehagen.
Research in the Theory of Inductive Inference by GDR Mathematicians - A Survey.
Information Sciences 22:149-169, 1980.

[Kowalski 79a] Robert A. Kowalski.
Logic for Problem Solving.
Elsevier North Holland Inc., 1979.

[Kowalski 79b] Robert A. Kowalski.
Algorithm = Logic + Control.
CACM 22, July, 1979.

[Kuhn 70] Thomas S. Kuhn.
The Structure of Scientific Revolutions.
The University of Chicago, 1970.

[Machtey & Young 78]
Michael Machtey and Paul Young.
An Introduction to the General Theory of Algorithms.
Elsevier North Holland, 1978.

[McCarthy 63] John McCarthy.
Programs With Common Sense.
In Marvin Minsky, editor, *Semantic Information Processing*, chapter 7. The MIT Press, Cambridge, Mass., 1963.

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

[Paterson & Wegman 76]
Paterson M. S. & Wegman M. N.
Linear unification.
In *8'th Annual ACM symposium on Theory of Computing*, pages 181-186. ACM, 1976.

[Pereira et al. 78] 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.
Provisional version.

[Plotkin 70] G. D. Plotkin.
A Note on Inductive Generalization.
In B. Meltzer and D. Mitchie, editor, *Machine Intelligence 5*, pages 153-165. Edinburgh University Press, Edinburgh, 1970.

[Plotkin 71a] Gordon D. Plotkin.
Automatic Methods of Inductive Inference.
PhD Thesis, Edinburgh University, August, 1971.

[Plotkin 71b] G. D. Plotkin.
A Further Note on Inductive Generalization.
In B. Meltzer and D. Mitchie, editor, *Machine Intelligence 6*, pages 101-124. Edinburgh University Press, Edinburgh, 1971.

[Popper 59] Karl R. Popper.
The Logic of Scientific Discovery.
Basic Books, New York, 1959.

[Popper 68] Karl R. Popper.
Conjectures and refutations: The Growth of Scientific Knowledge.
Harper Torch Books, New York, 1968.

[Reynolds 70] J. C. Reynolds.
Transformational Systems and the Algebraic Structure of Atomic Formulas.
In B. Meltzer and D. Mitchie, editor, *Machine Intelligence 5*, pages 135-153. Edinburgh University Press, Edinburgh, 1970.

[Robinson 65] J. A. Robinson.
A Machine Oriented Logic Based on the Resolution Principle.
JACM 12, January, 1965.

[Summers 76] Philip Dale Summers.
Program Construction from Examples.
PhD Thesis, Yale University, 1976.
Computer Science Dept. research report No. 51.

[Summers 77] Phillip D. Summers.
A Methodology for LISP Program Construction from Examples.
JACM 24:161-175, January, 1977.

[Van Emden & Kowalski 76]
M. H. Van Emden and R. A. Kowalski.
The Semantics of Predicate Logic as a Programming Language.
JACM 23:733-742, October, 1976.

[Winston 75] P. H. Winston.
Learning Structural Descriptions from Examples.
In P. H. Winston, editor, *The Psychology of Computer Vision*,. McGraw-Hill, New York, 1975.