Yale University
Department of Computer Science

Inductive Inference by Refinement

P. D. Laird

YALEU/DCS/TR-376
May 1986

Abstract

Many algorithms have been found that can identify in the limit any objects in a class of objects, given examples of the object. Automata, grammars, LISP and Prolog programs are among the syntactic expressions that have been used to represent the objects and their examples. A common strategy of most of these algorithms is to take as an initial hypothesis the most general (the least general) expression possible, and to make it less (more) general as counterexamples are obtained.

In his thesis, Ehud Shapiro devised algorithms of this form that infer first-order axioms for theories. He introduced the notion of *refinement operators* to implement the task of making hypotheses less general. In this report the concept of a refinement is generalized and treated more formally. We view a refinement as a recursively enumerable relation on the syntactic algebra that is consistent with the "more general than" ordering. Refinements can be *upward* (generalizing) or *downward* (specializing); they can be complete or partial, and can have a number of other properties which algorithms can utilize to make the inference procedure more efficient. With these definitions, we obtain general algorithm schemas for inductive inference over domains with a suitable ordering relationship between the syntactic and semantic objects. Several specific examples are given, including regular expressions and Boolean functions.

For the particular domain of first-order clause-form sentences, Shapiro's procedures for inferring an Herbrand model are generalized to allow the inference of incomplete theories. By representing refinements as axioms, we can refine the refinements, and thereby obtain a characterization of all refinements over a domain. This notion of *meta-refinements* may render less mysterious the task of designing an efficient refinement for a particular domain.

Finally, we obtain inference techniques which take advantage of "normal form" properties of an algebra, and apply these results to Shapiro's algorithm and others.

This work was partially funded by the National Science Foundation, under grants MCS 8002447 and DCS 8404226.

Contents

1 Background 4

2 Inferring Regular Expressions by Refinement 7

3 Properties and Uses of Refinements 13

3.1 Search Problems 13

3.2 Inductive Inference Problems 17

3.3 Refinements 21

3.4 Inference Algorithms Using Refinements 24

3.5 Improvements 29

3.5.1 Separating $\succ$ and $\approx$ 30

3.5.2 Locally Finite Refinements 31

3.5.3 Bounding Expressions 33

4 Applications to Logic 39

4.1 Inferring Classes of Propositional Models 39

4.2 Inferring Classes of First-Order Models 40

4.3 Inferring Theories: Comparison with Shapiro's Approach 43

4.4 A Refinement for Clause-Form Sentences 44

4.5 Meta-Refinements 53

4.6 Bottom-Up Inference 54

4.7 Normal Forms and Monotonic Operations 58

5 Conclusions 64

6 Acknowledgments 65

A Summary of Shapiro's Algorithm 67

List of Figures

1 Axioms for Equivalence ($\approx$) in Boolean Algebra 16
2 Axioms for Equivalence ($\approx$) of Regular Expressions 18
3 Downward Inference with Separable Refinement 32
4 Locally finite refinement for Boolean expressions 34
5 Downward Inference with Locally Finite Refinement 35
6 Downward Inference with Bounding Expressions 37
7 Example Axioms 42
8 Resolution Proof Tree 49
9 Resolution Proof Tree (with x-clauses) 51
10 Downward Inference of Normal Form Expressions 61
11 Model Inference Algorithm 68

Inductive Inference by Refinement

P. D. Laird *

November, 1985
Revised May 1986

Inductive inference can be described as the process of forming generalizations from partial information. For example, we hear unfamiliar words in context and form hypotheses about their meanings. The subject has been studied from many viewpoints (see [Angluin-83] for an overview); in this report, we consider a class of techniques in which hypotheses are strengthened or weakened in response to examples and counterexamples.

Objectives of the study of inductive inference include making this inductive process effective and identifying those aspects common to all domains over which inferences can be made. To date, studies have focused mainly on domains that can be defined precisely: finite-state automata, grammars, sets, logic, LISP programs, etc. Ultimately, however, we hope to abstract the essential properties of inductive inference in order to apply them to more complex domains, such as natural language.

This report describes work in progress toward this goal. We shall find that many of the clever inference techniques invented for particular domains can be understood as directed searches that take advantage of an order-preserving relationship between the semantic entities and the syntactic representations of these entities. Conversely, we can extend this idea to obtain inference techniques on any domain which enjoys these properties.

*Copyright ©1985 by P. D. Laird.

1 Background

The model for inductive inference that we shall use is the following. An Inductive Inference Machine (I.I.M.) is trying to infer an object ("rule") from a given class of rules. It adopts a hypothesis language in which every rule is represented at least once. It also agrees with an oracle about what constitutes a set of examples, and how those examples are to be presented. Acceptability of the example set entails that any incorrect hypothesis will disagree with some finite initial portion of the presentation.

After all of these details have been resolved, the inference process begins. The I.I.M. calls upon the oracle for an example, computes and outputs its next hypothesis, and returns to the oracle for another example. In some cases the I.I.M. may also be allowed to ask questions of the oracle (such as, "Is x an example of the rule?"). The I.I.M. is said to (*EX-*)*identify R in the limit* if, in the (infinite) sequence of hypotheses $H_1, H_2, \ldots$, there is an n such that H_n denotes R, and H_{n+1} is the same as H_n for all $i > 0$. More generally, the I.I.M. identifies a *class* of rules in the limit if it can identify each rule in the class, given any acceptable presentation of the rule.

Identification in the limit, originally suggested by Gold [Gold-67], is not very practical as a convergence criterion for inductive inference. It serves roughly the same purpose in inductive inference as the recursive property does in the study of algorithms: it guarantees that the process "terminates" (stops changing its mind) with a correct answer, but does not account for its complexity.

A very simple algorithm for identification in the limit is that of *identification by enumeration* ([Gold-67]). Let $H_1, H_2, \ldots$ be an effective enumeration of the possible hypotheses. When examples $e_1, \ldots, e_k$ have been presented, the I.I.M. offers as its next hypothesis the first in the sequence H_i that is compatible with all k examples. Provided the oracle never presents faulty examples, this method will converge in the limit.

Often, however, the domain enjoys a partial order, $\geq$, with the property that, if hypothesis H is incorrect for an example e, then for any hypothesis $H' \geq H$, H' is also incorrect for e. We can take advantage of this property to speed up the convergence by eliminating from the hypothesis space any related by $\geq$ to a failed hypothesis.

For example, let the object be sets of strings over a fixed alphabet Σ, and suppose the hypotheses are regular expressions. As examples, we may take all "signed" regular expressions of the form $\pm(\sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_k)$, where $k > 0$ and $\sigma_i \in \Sigma$ for each $1 \leq i \leq k$. A positive sign indicates that the example denotes a string in the target language L, and a negative sign, that the string is not in L. For the examples $\{+10, +100, -0\}$, compatible hypotheses would include 10^*, 100^*, $1(1+0)^*$, and $(10+100)$, but not 1^*0^* or $(1+0)^*$. An acceptable presentation of the examples is one in which every string occurs at least once. (Of course, each time it must have the same sign.) A presentation in which only positive examples occur is not sufficient to ensure identification in the limit (as shown in [Gold-67]).

Straightforward identification in the limit could proceed by enumerating all regular

expressions over Σ in order of increasing length. There are well-known algorithms for testing whether a regular expression R denotes a language that includes a string w ([Aho-74]). The I.I.M. outputs the first expression in the enumeration which includes all positive examples of $+w$ but no negative examples $-w$. Since every string is eventually presented, an incorrect hypothesis will eventually be shown a counterexample. And since a correct hypothesis is included in the enumeration, the I.I.M. will converge in the limit to a correct expression – in fact, one of least size.

Languages, being sets of strings, can be partially ordered by inclusion ($\subseteq$). If a hypothesis R includes a string w, and w is presented as a negative example, then R is too general. We can then discard, not only R, but any expression more general than or equivalent to R, such as $R + R$, $R + S$, R^*, etc.. Likewise, if R fails to include a string w given as a positive example, then R is too special, and we can discard along with R any expressions less general than or equivalent to R. Of course, in order to take full advantage of this efficiency, we need to be able to enumerate expressions no more general or no less general than a given expression (or set of expressions).

Mitchell ([Mitchell-82]) describes a class of algorithms which search a space according to a partial order. With his approach, lists are kept of the most general hypotheses which agree with no negative examples, and the least general hypotheses which agree with all positive example, subject to the additional constraint that the most general hypotheses are all more general than the least general hypotheses. At any time, the two lists define the information we have so far from all the examples. In principle, we can select from these hypotheses one which meets other criteria (e.g., , least size) as our preferred hypothesis. With sufficient examples, the two lists should converge to the same single hypothesis, or set of equivalent hypotheses. This *version-space* approach requires that the search space be bounded in both breadth and depth for convergence to be assured.

Shapiro ([Shapiro-81,Shapiro-82]) extends this idea to the (infinite) domain of first-order logic, and devises a program that infers sentences from examples of their logical consequences. The expressive power of first-order logic makes this technique very powerful indeed. He restricts the form of his sentences to those in *clause form* (prenex conjunctive normal form with no existential quantifiers), primarily to take advantage of standard resolution proof procedures. The symbols of the first-order language are fixed. The target of the inference is an Herbrand model, a set of atoms of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and the t_i's are variable-free terms. Examples of the model are atoms of this form, signed + or − according to whether they belong to the model or not. A sentence is correct for the examples if it logically implies all positive and no negative examples.

Shapiro's inference algorithm entails a search for a sentence that characterizes the model. When a sentence disagrees with an example, an algorithm is invoked that identifies a portion of the sentence that is erroneous – e.g., , implies too much or not enough. If it implies too little, a new clause is added to the sentence. If it implies too much, a clause of the sentence is *refined*: a list of related clauses is computed with the property

that these new clauses are no more general than the erroneous clause. These new clauses are then available as replacements for the original clause. The refinement operation helps direct the search by eliminating from consideration clauses that are more general than the one already recognized as too general. And together with a set of algorithms for identifying the false clause or clauses in the sentence, the refinement makes the search algorithm *adaptive* in response to the examples.

A more detailed description of Shapiro's work is presented in the appendix. We shall refer to these ideas several times in the rest of this report.

Shapiro's method is capable of making inductive inferences over many domains for which *ad hoc* algorithms had been devised previously (formal languages, functional programs, etc.), and extends to first-order expressible domains where other algorithms do not apply. But the syntactical form of Shapiro's inferences is limited to first-order sentences. Oftentimes, first-order syntax is not the most convenient and intuitive language for representing objects. For example, regular sets can be defined using clause-form sentences, but regular expressions and finite-state diagrams are usually more useful representations for these sets. The motivating question for the research in this report was whether Shapiro's ideas could be extended or modified for inference over syntactical structures other than first-order logic. In the next chapter, we show how to infer regular expressions using refinements to guide the search. This section serves as an introductory example for the more general ideas which follow, including algorithms which apply to a large class of domains of inference.

2 Inferring Regular Expressions by Refinement

As motivation for the more general results, we consider the problem of inferring regular sets by finding regular expressions denoting them. In [Shapiro-81] regular sets are inferred by finding a first-order axiomatization of the set. It turns out that, with a one-place predicate symbol $L(x)$ denoting membership in the language and a two-place function symbol "$\cdot$" denoting concatenation, a regular language can be axiomatized using only Horn sentences of the form $L(\alpha)$ and $L(\alpha \cdot x) \rightarrow L(\beta \cdot x)$, where α and β are ground strings and x is a variable (denoting an arbitrary string). Furthermore, it can be shown that any such axiomatizaton defines a regular language, and that the number of axioms required may be exponential in the number of states of the minimal deterministic finite automaton that accepts the language. These facts are not obvious, however; and for most people, this syntax is less meaningful than the more conventional regular expressions or state-machine diagrams.

Instead, we prefer to find a regular expression from examples of strings in and out of the corresponding regular language. So let us adapt Shapiro's algorithm to handle the new syntax. We need to specify:

- a most-general expression;
- a refinement ρ for regular expressions;
- a syntactic form for our examples; and
- an algorithm for deciding whether a regular expression denotes a set which includes the example string.

Let $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ be the fixed alphabet over which the regular sets are defined. The language of regular expressions uses $+$ and $\cdot$ to denote union and concatenation operations, with ϕ and ϕ^* as the corresponding identities, and * to denote the Kleene closure. The set Σ^* is the most general regular set since it includes all others over the same alphabet. There are infinitely many regular expressions denoting Σ^*; we arbitrarily select $(\sigma_1 + \ldots + \sigma_k)^*$ and abbreviate it $\top$.

The refinement ρ should have the following properties:

1. ρ is a relation on regular expressions. Equivalently, it can be viewed as a function $\rho : \mathcal{E} \rightarrow 2^{\mathcal{E}}$, where $\mathcal{E}$ is the class of all regular expressions. In addition, we want ρ to be *locally finite*: $\{E' | E' \in \rho(E)\}$ is finite for all E.
2. For all $E_1, E_2 \in \mathcal{E}$, whenever $E_2 \in \rho(E_1)$, then the regular set $|E_2|$ denoted by E_2 is a (proper or improper) subset of the set $|E_1|$ denoted by E_1.

3. ρ is *complete* for the most general expression $\top$ in this sense: The set $\rho^*(\top)$ of all expressions obtainable in a finite number of refinement steps from $\top$ includes at least one expression for every regular set.

A refinement ρ_1 with these properties is as follows. Let $E, E_1, E_2, \ldots$ be regular expressions over Σ. Write $E_1 \rightarrow E_2$ to denote the relation $E_2 \in \rho_1(E_1)$.

1. $E \rightarrow (E + E)$
2. $E^* \rightarrow (E^*)^*$
3. $E^* \rightarrow (E^* \cdot E^*)$
4. $E^* \rightarrow E$
5. $\sigma_i \rightarrow \phi$, for $1 \leq i \leq n$
6. If $E_1 \rightarrow E$, then $E_1 + E_2 \rightarrow E + E_2$ and $E_2 + E_1 \rightarrow E_2 + E$
7. If $E_1 \rightarrow E$, then ${E_1}^* \rightarrow E^*$
8. If $E_1 \rightarrow E$, then $E_1 \cdot E_2 \rightarrow E \cdot E_2$ and $E_2 \cdot E_1 \rightarrow E_2 \cdot E$

It is easy to see that ρ_1 is locally finite, since $\rho(\phi)$ is $\{\phi + \phi\}$, $\rho(\sigma_i)$ is $\{\phi, \sigma_i + \sigma_i\}$ for all i, and inductively if E is of the form $E_1 + E_2$, $E_1 \cdot E_2$, or ${E_1}^*$, then §6 – §8 generate a finite set of reductions, and §1 – §4 each generate at most one. A similar induction can be used to show that every expression in $\rho(E)$ denotes a regular subset of E (property 2 above).

Likewise, we can show that ρ_1 is complete for $\top = \Sigma^*$. (Let $\xrightarrow{n}$ denote one application of rule § n, and $\xrightarrow{*}$ denote zero or more applications of any of rules § $1 - 5$.)

(ϕ) $\Sigma^* \xrightarrow{4} \Sigma \xrightarrow{5,6} (\sigma_1 + \ldots + \sigma_{k-1} + \phi) \xrightarrow{*} (\phi + \ldots + \phi) \equiv \phi$. Thus (an expression equivalent to) $\phi \in \rho^*(\top)$.

(σ_i) $\Sigma^* \xrightarrow{4} \Sigma \xrightarrow{*} (\phi + \ldots + \phi + \sigma_i + \phi + \ldots + \phi) \equiv \sigma_i$, for each i. Thus (an expression equivalent to) $\sigma_i \in \rho^*(\top)$.

Inductively, if E_1 and E_2 are in $\rho^*(\top)$, then

(union) $\Sigma^* \xrightarrow{1} (\Sigma^* + \Sigma^*) \xrightarrow{*} (E_1 + E_2)$ (using §6).

(star) $\Sigma^* \xrightarrow{2} (\Sigma^*)^* \xrightarrow{*} (E_1)^*$ (using §7).

(concatenation) $\Sigma^* \xrightarrow{3} (\Sigma^* \cdot \Sigma^*) \xrightarrow{*} (E_1 \cdot E_2)$ (using §8).

Rev: May 29, 1986

As "examples" we may take regular expressions denoting sets with a single string, in the form $e = \sigma_{i_1} \cdot \ldots \cdot \sigma_{i_n}$, with association implicitly to the right. The oracle eventually presents every string (at least once) in the form of a signed example: $+e$ indicates that the string is in the target set, $-e$ that it is not. As noted earlier, deciding whether a given expression E includes a string w can be done efficiently. We shall write $E \Rightarrow w$ whenever this holds; otherwise, we write $E \not\Rightarrow w$.

Finally, we have the following algorithm for inferring regular expressions from examples.

Algorithm 2.1 *Inference of Regular Expressions*

Procedure: Initialize:

$H \leftarrow \top$. (H is the current hypothesis.)
$Q \leftarrow$ *emptyqueue*. (Q contains possible future hypotheses.)
examples $\leftarrow$ *emptyset*. (Store all examples.)

do forever:

Call for the next example and add it to the set *examples*.
repeat
if $H \Rightarrow -w$ for some negative example w, **then**
Append $\rho_1(H)$ to the tail of Q.
$H \leftarrow$ *next*(Q). (*next* removes the front element of the queue, or fails if the queue is empty.)
else if $H \not\Rightarrow +w$ for some positive example w, **then**
$H \leftarrow$ *next*(Q).
until H is correct for all words in *examples*.
Output H as the current guess. ◇

The algorithm maintains a current hypothesis, H, and a queue Q of potential hypotheses. Initially, Q is empty, and H contains the most general hypothesis, $\top$. The current hypothesis is tested against all examples seen so far. If H covers an example it should not, it is too general; its refinements $\rho_1(H)$ are added to the queue of possible hypotheses, and a new hypothesis is taken from the front of the queue. If H fails to cover an example it should, H is too specific and is therefore discarded. (Property 2 above implies that all refinements of H are likewise too specific). Note that $\top$ contains all strings, so that the first negative example causes $\rho_1(\top)$ to be enqueued.

Let us now demonstrate the following straightforward result.

Proposition 2.2 *For any regular set R and any acceptable presentation of R, Algorithm 2.1 identifies R in the limit.*

Proof: If some hypothesis H is never refuted by any counterexample, then H must represent R, since every string is eventually presented, and H thus contains all and only strings in R. So Algorithm 2.1 will neither converge to an incorrect hypothesis nor discard a correct one, and we need only show that it eventually finds a correct hypothesis.

From the completeness property of ρ_1 we know that there is a chain of refinement steps $E_0 \rightarrow \ldots \rightarrow E_n$, with $E_0 = \top$ and E_n representing R. When $n = 0$ we have the trivial case where R is Σ^*. So assume n is greater than zero and as small as possible for the given R. Property (2) of ρ_1, together with the minimality of n, ensures that $|E_0| \supseteq |E_1| \supseteq \ldots \supseteq |E_n|$, and that $|E_i| \supsetneq |E_n|$ for each $0 \leq i < n$. Thus there are strings w_i for $0 \leq i < n$) such that w_i is a negative example for E_i; and for every positive example $+w$ of R, $E_i \Rightarrow w$ (for all $0 \leq i \leq n$).

We assume the algorithm diverges, and derive a contradiction. Below, we shall argue that divergence implies that E_n must become the hypothesis H at some point in the execution of the algorithm. Divergence also implies that every hypothesis is eventually discarded as a result of a counterexample. But E_n is correct for all examples and will never be discarded. Thus we are forced to conclude that the algorithm converges.

So assume that every hypothesis is eventually discarded. We argue, by induction on n, that E_n eventually becomes the hypothesis. E_0 is the initial hypothesis. Suppose $H = E_i$, for $i < n$. Since $| E_i | \supsetneq | R |$, the only counterexample for E_i is a string presented as a negative example which is included in the set E_i but not in the set R. There is such a string (w_i). So E_i will be discarded and its refinements added to the end of the Q. Since Q is always finite, and the algorithm diverges, the finite number of expression preceding E_{i+1} will all be tried as hypotheses and discarded. Thus E_{i+1} eventually becomes the current hypothesis.

And from earlier comments, we conclude that the algorithm converges. □

Algorithm 2.1 differs from that of [Shapiro-82] in two significant ways. Shapiro's system discards and refines *clauses*, not entire sentences. In his case "the current hypothesis" is a set of clauses which, taken together, imply all positive examples and no negative ones. Refining the entire expression, instead of just one clause, degrades the efficiency considerably. Also, Shapiro's algorithm is *incremental*: once a hypothesis is discarded, it is never retried (even if there is another refinement path to it). It is possible to modify Algorithm 2.1 to be incremental, but even with this improvement it remains too inefficient to be practical.

"Efficient" has to be interpreted in a relative sense, since refinements typically suffer from explosive growth in the number of hypotheses at each level. Consider, for example, the refinement ρ_1 above, over the alphabet $\Sigma = \{\sigma\}$. Expressions of the form σ^n occur at depths $2n-1$ and more in the graph: $\Sigma^* \rightarrow \Sigma^* \cdot \Sigma^* \rightarrow \cdots \rightarrow (\Sigma^*)^n \rightarrow \sigma(\Sigma^*)^{n-1} \rightarrow \cdots \rightarrow \sigma^n$ is a path of least length. It is also easy to verify that the number of distinct expressions

at each level that are equivalent to Σ^* at least doubles. Since these expressions are all strictly too general, the algorithm will continue refining them instead of simply discarding them. Hence with this refinement, the algorithm will be forced to consider an exponential number of hypotheses before finding an expression equivalent to σ^n. (Of course, another refinement might allow regular expressions to be found more quickly.)

Two questions arise from the above discussions.

1. Can we find more/better refinements than ρ_1 for regular expressions?
2. Could this approach be turned around so that it starts with a *least* general expression (say, ϕ), and uses a *generalizing* refinement γ instead of a *specializing* refinement ρ?

In fact we can do both of these. Later we shall see that there is an enumerable class of refinements varying in their completeness and other properties. We present here, without proof, a generalizing refinement γ_1 complete for ϕ and an algorithm which infers regular expressions in the limit "bottom up" – that is, starting from an initial hypothesis of the empty regular set, represented by ϕ.

Let $E, E_1, E_2, \ldots$ be regular expressions over Σ. Write $E_1 \rightarrow E_2$ to denote the relation $E_2 \in \gamma_1(E_1)$.

1. $E \rightarrow (E + E)$
2. $\phi \rightarrow (\phi \cdot \phi)$
3. $E \rightarrow (E)^*$
4. $\phi \rightarrow \sigma_i$, for $1 \leq i \leq n$
5. If $E_1 \rightarrow E$, then $E_1 + E_2 \rightarrow E + E_2$ and $E_2 + E_1 \rightarrow E_2 + E$
6. If $E_1 \rightarrow E$, then $E_1{}^* \rightarrow E^*$
7. If $E_1 \rightarrow E$, then $E_1 \cdot E_2 \rightarrow E \cdot E_2$ and $E_2 \cdot E_1 \rightarrow E_2 \cdot E$

Note that the reductions that comprise γ_1 are of two types: those which increase the *size* of the expression without changing its meaning (equivalence reductions), and those which generalize the expression to a larger set (generalizing reductions). The latter are the inverse of the specializing reductions for the downward refinement ρ_1.

The "bottom-up" equivalent of Algorithm 2.1 is shown below. Given the striking duality between the top-down and bottom-up cases, in both the refinements and the algorithms, we should expect that there is something fundamental going on that we should explore. Indeed, this observation was the primary motivation for much of this research.

Algorithm 2.3 *Inference of Regular Expressions*

Procedure: Initialize:

$H \leftarrow \phi$. (H is the current hypothesis.)
$Q \leftarrow$ *emptyqueue*. (Q contains possible future hypotheses.)
examples $\leftarrow$ *emptyset*. (Store all examples.)

do forever:

Call for the next example and add it to the set *examples*.
repeat
if $H \not\Rightarrow +w$ for some negative example w, **then**
Append $\gamma_1(H)$ to the tail of Q.
$H \leftarrow$ *next*(Q). (fails if the queue is empty)
else if $H \Rightarrow -w$ for some positive example w, **then**
$H \leftarrow$ *next*(Q).
until H is correct for all words in *examples*.
Output H as the current guess. ◇

3 Properties and Uses of Refinements

In this section we examine refinements more abstractly as recursively enumerable binary relations on a domain of expressions, which preserve an underlying semantic ordering. In addition to generalizing the ideas we have already seen, we develop several new examples illustrating how the concepts can be applied.

3.1 Search Problems

We step back briefly from inductive inference to define the "search problem" in such a way as to clarify the relationship between search and inference.

Definition 3.1 *An* abstract search problem *is a* 5*-tuple* $(\mathcal{D}, d_0, \mathcal{E}, h, \mathcal{O})$*, where*

- $\mathcal{D}$ *is a finite or countable set, called the* semantic domain *of objects.*
- d_0 *is a designated element of* $\mathcal{D}$*, (the* target object*).*
- $\mathcal{E}$ *is a countable set of expressions.*
- h*:* $\mathcal{E} \rightarrow \mathcal{D}$ *is a surjective mapping from expressions to objects.*
- $\mathcal{O}$ *is an oracle for the characteristic function of the set* $h^{-1}(d_0) = \{e \in \mathcal{E} \mid h(e) = d_0\}$*. Specifically,* $\mathcal{O}(e) = 1$ *if* $h(e) = d_0$*, and* $\mathcal{O}(e) = 0$ *otherwise.*

Provided the set $\mathcal{E}$ is recursively enumerable, there is an obvious search algorithm that finds d_0 in the limit.

Algorithm 3.2 *Exhaustive Search*

Input:	A recursively enumerable set $\mathcal{E}$ of expressions.
	An oracle $\mathcal{O}$ for $h^{-1}(d_0)$
Output:	An expression e_0 such that $h(e_0) = d_0$.
Procedure:	
	Let $e_1, e_2, \ldots$ be an enumeration of $\mathcal{E}$.
	$i \leftarrow 1$.
	while $\mathcal{O}(e_i) \neq 1$, $i \leftarrow i + 1$.
	Output e_i. ◇

This simple algorithm illustrates several things. First, search can be viewed as a special case of inference in which "examples" are expressions labeled + if they represent the target and − if they don't. The amount of information in a negative example is usually very small. Furthermore, the search algorithm determines the order in which the "examples" are considered, whereas in inductive inference algorithms examples are typically presented by an external source. Finally, the oracle O above serves to remove from consideration the complexity of deciding whether an expression has a particular property. In practice, this is often a hard problem; here we are electing to ignore this aspect.

The function h: $\mathcal{E} \rightarrow \mathcal{D}$ defines the relation between the syntax and the semantics. A well-known theorem of algebra states that h induces an equivalence relation on $\mathcal{E}$: $e_1 \approx e_2$ iff $h(e_1) = h(e_2)$. We shall write $\not\approx$ to denote the complementary relation of $\approx$.

The ability to compute the relation $\approx$ offers the potential for improving Algorithm 3.2. Instead of enumerating $\mathcal{E}$, we enumerate the equivalence classes of $\mathcal{E}$ and select a single representative from each class for presentation to the oracle O. In practice, however, this may not be feasible. $\approx$ may be recursive (as in the case of regular expressions); $\approx$ may be recursively enumerable (expressions for strings in a free semi-group); $\not\approx$ may be r.e. (context-free grammars); or none of these (LISP programs).

Example 3.3 Let $\mathcal{D}$ be the set $\mathbf{N}$ of natural numbers. Let $\mathcal{E}$ be strings in $\{0,1\}^*$. Let h: $\mathcal{E} \rightarrow \mathcal{D}$ interpret each string $b_{n-1}\ldots b_0$ as a binary number, $\sum_{i=0}^{n-1} b_i 2^i$. An effective search procedure is to enumerate the strings of 0's and 1's and ask the oracle about them. Since $\approx$ is decidable in this case, we need ask only about the string 0 and about strings beginning with 1. △

Even if $\approx$ is not decidable, some non-trivial subset of it usually is. Let $\approx_1$ be any decidable equivalence relation contained in $\approx$. With $\approx_1$ we can improve Algorithm 3.2 with respect to the number of oracle queries, at the cost of additional computation to determine instances of $\approx_1$.

Algorithm 3.4 *Exhaustive Search modulo* $\approx_1$

Input: A recursive equivalence relation $\approx_1$
An oracle O for $h^{-1}(d_0)$

Output: An expression e_0 such that $h(e_0) = d_0$.

Procedure:

Let $e_1, e_2, \ldots$ be an enumeration of $\mathcal{E}$.
$i \leftarrow 1$.
while $O(e_i) \neq 1$

commutativity	$e_1 \cap e_2 \approx e_2 \cap e_1$ $e_1 \cup e_2 \approx e_2 \cup e_1$
associativity	$e_1 \cap (e_2 \cap e_3) \approx (e_1 \cap e_2) \cap e_3$ $e_1 \cup (e_2 \cup e_3) \approx (e_1 \cup e_2) \cup e_3$
absorption	$e_1 \cap (e_1 \cup e_2) \approx e_1$ $e_1 \cup (e_1 \cap e_2) \approx e_1$
distributivity	$e_1 \cap (e_2 \cup e_3) \approx (e_1 \cap e_2) \cup (e_1 \cap e_3)$ $e_1 \cup (e_2 \cap e_3) \approx (e_1 \cup e_2) \cap (e_1 \cup e_3)$
bounds	$e_1 \cap \mathbf{F} \approx \mathbf{F}$ $e_1 \cup \mathbf{T} \approx \mathbf{T}$
complement	$e_1 \cap \sim e_1 \approx \mathbf{F}$ $e_1 \cup \sim e_1 \approx \mathbf{T}$
reflexivity	$e_1 \approx e_1$
symmetry	$(e_1 \approx e_2) \rightarrow (e_2 \approx e_1)$
transitivity	$(e_1 \approx e_2) \cap (e_2 \approx e_3) \rightarrow (e_1 \approx e_3)$
substitution	$(e_1 \approx e_1') \rightarrow (e_1 \cap e_2) \approx (e_1' \cap e_2)$ $(e_1 \approx e_1') \rightarrow (e_1 \cup e_2) \approx (e_1' \cup e_2)$ $(e_1 \approx e_1') \rightarrow \sim e_1 \approx \sim e_1'$

Figure 1: Axioms for Equivalence ($\approx$) in Boolean Algebra

$i \leftarrow i + 1$.
while $e_i \approx_1 e_j$ for some j such that $0 < j < i$, $i \leftarrow i + 1$.
Output e_i. ◇

Any equivalence $\approx_1$ which is a subset of $\approx$ is *complete* in the sense that Algorithm 3.2 will converge to a correct expression. By contrast, an equivalence relation $\approx_2$ which properly includes $\approx$ is *incomplete* in that Algorithm 3.4, finding that $e_1 \approx_2 e_2$, may not query $O(e_2)$ when in fact $h(e_2) \neq h(e_1)$. If $h(e_2) = d_0$ then the algorithm may fail to converge. But it is also possible that $\approx_2$ reduces the number of equivalence classes *without* affecting the convergence; for example, we might know in advance that certain expressions do not represent d_0, and if the equivalence classes for some of these were merged by $\approx_2$ the search would still work correctly. In this case it might pay to use the relation even if it were incomplete. We shall see that this property has an analog with refinement relations, in that incomplete relations may nevertheless be useful in algorithms for inductive inference by refinement.

Any effectively enumerable relation $\approx$ is finitely axiomatizable in a first-order system. The proof system we shall employ is one in which all axioms are sentences in clause form, with resolution as the only rule of inference.

Example 3.5 Let D be $P(P(\{X_1, \ldots, X_n\}))$, the sets of subsets of n objects. For expressions denoting elements of D, we can use Boolean expressions over the propositional symbols $x_1, \ldots, x_n$. That is, $\mathcal{E}$ is the set of well-formed expressions using the constants **T**, **F**, and x_i (for $1 \leq i \leq n$); the one-place symbol $\sim$ (denoting complementation); and the two-place symbols $\cup$ and $\cap$ (denoting union and intersection, resp.). The expression **T** denotes all subsets; **F** represents the empty set; and $x_1 \cap \sim x_2$ denotes subsets containing X_1 but not X_2.

The algebra of Boolean expressions is well known. In Figure 1 we exhibit an axiomatization based on the equational presentation given in [Birkhoff-77], in a form suitable for resolution proofs of equivalence. Each clause is implicitly preceded by universal quantification over its variables.

If for two well-formed Boolean expressions e_1 and e_2 we can prove from these axioms that $e_1 \approx e_2$, then e_1 and e_2 denote the same class of subsets. For example, $x_1 \cap \sim x_1$ and **F** both denote the empty set.

In general, finitely presented algebras axiomatize the equivalence ($\approx$) of expressions in the algebra. △

Example 3.6 Let D be the class of regular subsets of some finite alphabet Σ. Let $\mathcal{E}$ be the regular expressions over the alphabet Σ. Unlike Boolean expressions, there is no finite set of equations that completely characterizes equivalence of regular expressions ([Red'ko-64]). Salomaa ([Salomaa-66]) defines two non-equational finite axiom schemes

for regular expressions which are consistent and complete for ground expressions (i.e., expressions without variables). A clause-form axiomatization corresponding to his F_1 system is given in Figure 2. This system can be used to prove, for example, that $(1(110+0110)^*)^* = ((1((0+\phi^*)110)^*)^*)^*$, but not that $e_1{}^* = e_1{}^{**}$ for all expressions e_1. But our interest will be confined to refinements, for which all expressions are ground expressions.

The language in Figure 2 uses two predicate symbols: $\approx$ for equivalence, and *newp* for *"not-having-the-empty-word-property"*. The symbols $+$, $\cdot$, and * are as usual, except that $\cdot$ is often implicit (as in $\alpha\beta$). △

3.2 Inductive Inference Problems

We could turn a search problem into an inference problem by replacing the query oracle ("Is $h(e) = d_0$?") with an example oracle EX, which returns a signed expression $\pm e$ such that "$+e$" means $h(e) = d_0$ and "$-e$" means $h(e) \neq d_0$. There is no strategy for such an inference problem, unless it is to smoke a cigar while waiting for the oracle to return a positive example.

A much more interesting way to define an abstract inference problem is to utilize the services of an oracle for examples that returns only *partial information* about the relationship between an expression e and the target d_0.

Definition 3.7 *An* abstract inference problem *is a* 6*-tuple* $(\mathcal{D}, d_0, \mathcal{E}, h, ASK, EX)$*, where*

- $\mathcal{D}$ *is a finite or countable set partially ordered by* $\geq$.
- d_0 *is a designated target element of* $\mathcal{D}$.
- $\mathcal{E}$ *is a countable set of expressions.*
- h: $\mathcal{E} \rightarrow \mathcal{D}$ *is a surjective mapping from expressions to objects.*
- ASK *is an oracle for* $\geq$ *such that* $ASK(e_1, e_2) = 1$ *if* $h(e_1) \geq h(e_2)$*, and* 0 *otherwise.*
- EX *is an oracle for examples of* d_0*, such that if* $EX() = +e$ *then* $d_0 \geq h(e)$*, and if* $EX() = -e$ *then* $d_0 \not\geq h(e)$.

Note that the inverse relation $\leq$ is also a partial order, and induces a dual abstract inference problem, including an oracle EX for examples of expressions e such that $d_0 \leq h(e)$ or $d_0 \not\leq h(e)$. Note also that $h(e_1) \geq h(e_2)$ and $h(e_2) \geq h(e_1)$ imply $h(e_1) = h(e_2)$ and therefore $e_1 \approx e_2$. Finally, the oracle ASK, like its counterpart in the search problem, serves mainly to remove from consideration the complexity of deciding $h(e_1) \geq h(e_2)$.

A_1	$e_1 + (e_2 + e_3) \approx (e_1 + e_2) + e_3$
A_2	$e_1(e_2e_3) \approx (e_1e_2)e_3$
A_3	$e_1 + e_2 \approx e_2 + e_1$
A_4	$e_1(e_2 + e_3) \approx e_1e_2 + e_1e_3$
A_5	$(e_1 + e_2)e_3 \approx e_1e_3 + e_2e_3$
A_6	$e_1 + e_1 \approx e_1$
A_7	$\phi^*e_1 \approx e_1$
A_8	$\phi e_1 \approx \phi$
A_9	$e_1 + \phi \approx e_1$
A_{10}	${e_1}^* \approx \phi^* + {e_1}^*e_1$
A_{11}	${e_1}^* \approx (\phi^* + e_1)^*$
A_{12}	$e_1 \approx e_1e_2 + e_3 \wedge newp(e_2) \rightarrow e_1 = e_3{e_2}^*$
reflexivity	$e_1 \approx e_1$
symmetry	$(e_1 \approx e_2) \rightarrow (e_2 \approx e_1)$
transitivity	$(e_1 \approx e_2) \cap (e_2 \approx e_3) \rightarrow (e_1 \approx e_3)$
substitution	$(e_1 \approx e_1') \rightarrow (e_1 + e_2) \approx (e_1' + e_2)$
	$(e_1 \approx e_1') \rightarrow (e_1e_2) \approx (e_1'e_2)$
	$(e_1 \approx e_1') \rightarrow {e_1}^* \approx {e_1'}^*$
newp	$newp(\sigma_i)$ (for all $\sigma_i \in \Sigma$)
	$newp(e_1) \wedge newp(e_2) \rightarrow newp(e_1 + e_2)$
	$newp(e_1) \rightarrow newp(e_1e_2)$
	$newp(e_1) \rightarrow newp(e_2e_1)$

Figure 2: Axioms for Equivalence ($\approx$) of Regular Expressions

In our notation, we shall write $x \sim y$ to indicate that x and y are comparable with respect to $\geq$. By $x \not\sim y$ we shall mean that x and y are incomparable: neither $x \geq y$ nor $y \geq x$. Also, let us adopt the more mnemonic form $ASK(e_1 \geq e_2?)$ in preference to $ASK(e_1, e_2)$.

So far, our only access to information about the partial order on $\mathcal{D}$ is via the oracles. In particular, we know nothing about how (or even if) the syntactic structure of the expressions relates to the partial order on $\mathcal{D}$.

Definition 3.8 *The oracle $EX()$ gives a* complete presentation *of d_0 if for every $e \in \mathcal{E}$ such that $d_0 \geq h(e)$, $EX()$ eventually returns "$+e$" at least once, and for every $e' \in \mathcal{E}$ such that $d_0 \not\geq h(e')$, $EX()$ eventually returns "$-e'$" at least once.*

The following algorithm is the counterpart to Algorithm 3.4.

Algorithm 3.9 *Identification by Enumeration*

Input: A recursively enumerable set $\mathcal{E}$ of expressions.
An oracle $ASK(e_1 \geq e_2?)$ for $h(e_1) \geq h(e_2)$?
An oracle $EX()$ for a complete presentation of d_0

Output: A sequence of expressions $H_1, H_2, \ldots$ such that h_n is correct for the first n examples.

Procedure:

Let $e_1, e_2, \ldots$ be an enumeration of $\mathcal{E}$.
$i \leftarrow 1$.
examples $\leftarrow$ *emptyset.*
do forever:
    examples $\leftarrow$ *examples* $\cup\, EX()$ (get next example)
    while $ASK(e_i \geq e?) = 1$ for some negative example $-e$ **or**
    $ASK(e_i \geq e?) = 0$ for some positive example $+e$,
        $i \leftarrow i + 1$.
    Output e_i as the next hypothesis.

Note that ASK is used only to relate an arbitrary expression e_i to an example expression; the full power of the oracle to relate two arbitrary expressions is never required. This will always be the case for the ASK oracles in this paper. If we wished, we could have defined the oracle with this restriction.

Theorem 3.10 *Algorithm 3.9 identifies d_0 in the limit.*

Proof: Suppose the algorithm outputs some expression e_i infinitely often. Then $h(e_i) \geq d_0$, since if e_0 is any expression with the property $h(e_0) = d_0$, then $+e_0$ is a positive example which must be presented at least once, and the **while** condition ensures that $ASK(e_i \geq e_0?) = 1$. Also, $h(e_i) \leq d_0$: otherwise, $-e_i$ would be presented as a negative example, and $ASK(e_i \geq e_i?)$ would be 0. Hence $h(e_i) = d_0$, showing that the algorithm cannot converge to an incorrect expression.

Furthermore, the algorithm must converge to some expression; for among the positive examples is an expression e_0 for d_0 which occurs somewhere in the enumeration of $\mathcal{E}$. □

We observe that the enumeration of $\mathcal{E}$ in Algorithm 3.9 is not needed if the algorithm does not have to issue a guess after every example. Instead, it can simply wait until there is some expression e_k among the positive examples such that $ASK(e_k \geq e_i?) = 1$ for all positive examples e_i; it then outputs e_k as its guess. In effect, it lets the oracle EX do the enumeration for it.

In practice, a complete presentation of the target d_0 is not always needed. With regular expressions, for example, we have seen that EX need give as examples only expressions denoting singleton regular sets. In other domains, however, the entire set of examples may be required.

Definition 3.11 *Let d_0 be the target object of an abstract inference problem. A* sufficient set of examples *for d_0 is a signed subset S of $\mathcal{E}$ such that the set $\mathcal{E}_S = \{e \in \mathcal{E} \mid h(e) \geq$ all positive examples and $h(e) \not\geq$ any negative examples in $S\}$ is precisely the set of expressions e for which $h(e) = d_0$. In other words, S determines d_0 up to equivalence (modulo $\approx$).*

Example 3.12 Let Q be the set of non-negative rational numbers partially ordered by $\leq$ in the usual sense. The designated element is some rational number $q_0 \geq 0$. As expressions we use integer ratios p/q. Examples take the forms "$p/q \leq q_0$" and "$p/q \not\leq q_0$". As a sufficient set S of examples for q_0, we may take $\{r \mid r < q_0$ and r is in lowest terms $\}$ as positive examples, and $\{s \mid s > q_0$ and s is in lowest terms $\}$ as negative examples. The set $\mathcal{E}_S$ is then the set of ratios equal to q_0. A sparser set of examples could be formed from two sets of ratios: one which approaches q_0 as a limit from below (positive examples), one approaching it from above (negative). △

Example 3.13 Let $\mathcal{D}$ be a subclass of the partial recursive functions on $\mathbf{N}$, partially ordered by: $f_1 \geq f_2$ iff, for all $x \in \mathbf{N}$, $f_1(x) = f_2(x)$ whenever $f_2(x)$ is defined. $\mathcal{E}$ can be any enumerable representation (such as Turing machines) for which every function in $\mathcal{D}$ has at least one representation. Fix a total function $f_0 \in \mathcal{D}$. A sufficient set of examples for f_0 is the class of functions f_i such that $f_i(i) = f_0(i)$, and $f_i(j)$ is undefined for $j \neq i$. A presentation usually takes the form of pairs $(i, f_i(i))$. Clearly, $f_0 \geq f_i$ for all i, and

f_0 is uniquely determined (up to equivalence modulo h) by the values f_i. Only positive examples are necessary in this case.

Inference of functions using this model has been the subject of many studies, including [Blum-75], [Pitt-84], and [Summers-77]. Of course, much of the difficulty associated with this type of inference problem lies within the oracle *ASK*. △

It is not necessary to define examples such that the target expression dominates ($\geq$) its positive examples: we could just as well take as positive examples those that dominate the target. The definition of a sufficient set then applies to the dual partial order $\leq$. In general it will depend upon the domain as to whether it is more convenient for examples to be smaller or larger than the object they exemplify, and we shall encounter both situations. During the exposition, however, we shall proceed as if the examples are below the target.

Corollary 3.14 *Algorithm 3.9 identifies d_0 in the limit when EX is an oracle for a sufficient set of examples.*

(For, the **while** clause ensures that the algorithm converges to the first expression in the enumeration that also belongs to the set $\mathcal{E}_S$.) □

At the risk of exhausting the reader's patience, we have cast inductive inference problems in a general form which includes a large class of particular problems as instances. Gold's "identification by enumeration" algorithm is a natural outgrowth of the definitions, as perhaps the simplest possible procedure for identification in the limit. The close relationship between inductive inference problems and search problems is also brought out when presented in this way. To some extent, conventional search algorithms, such as binary search, can be viewed as special case of a more general inductive search problem on a partially ordered set. While we will not pursue this idea further in this report, it probably merits further study.

3.3 Refinements

Sometimes the syntax of the expressions in $\mathcal{E}$ bears little or no relation to the algebraic structure of $\mathcal{D}$. (For example, $\mathcal{E}$ could be integers written as Roman numerals.) But in most cases of interest, there is a partial ordering or a quasi-ordering [1] of $\mathcal{E}$ that is closely related to that of $\mathcal{D}$. For example, if regular expression R_1 denotes a set $|R_1|$ and R_2 denotes $|R_2|$, then $R_1 + R_2$ denotes $|R_1| \cup |R_2|$. Let $\succeq$ be an ordering on regular expressions with the property that $R_1 + R_2 \succeq R_1$ and $R_1 + R_2 \succeq R_2$ for any expressions R_1, R_2. Then the map h from regular expressions to regular sets preserves this aspect of the ordering: $h(R_1 + R_2) \supseteq h(R_1)$ and $h(R_1 + R_2) \supseteq h(R_2)$.

[1] A *quasi-ordering* is a binary relation that is reflexive and transitive but not antisymmetric. We shall use the term *ordering* ambiguously to designate either a quasi-ordering or a partial ordering.

Definition 3.15 *Let $\succeq$ be an ordering of $\mathcal{E}$, $\geq$ a partial ordering of D, and h a mapping from $\mathcal{E}$ to D. Then h is said to be an* order homomorphism *if $h(e_1) \geq h(e_2)$ whenever $e_1 \succeq e_2$. If in addition $e_1 \succeq e_2$ whenever $h(e_1) \geq h(e_2)$, then h is called an* order quasi-isomorphism.

Example 3.16 The algebra of Boolean expressions was defined in Example 3.5. A quasi-ordering on the expressions is obtained by defining $e_1 \succeq e_2$ iff $e_1 \cap e_2 \approx e_2$. It is easily shown that $\succeq$ is indeed a quasi-ordering, and that an equivalent characterization is to define $e_1 \succeq e_2$ iff $e_1 \cup e_2 \approx e_1$.

We can also show that h is an order homomorphism: If $e_1 \succeq e_2$, then $e_1 \cap e_2 \approx e_2$, i.e., , $h(e_1 \cap e_2) = h(e_2)$. But $h(e_1 \cap e_2) = h(e_1) \cap' h(e_2)$ (where $\cap'$ is set intersection on D). Hence $h(e_1) \supseteq h(e_2)$.

This argument can be reversed to show that $h(e_1) \supseteq h(e_2)$ implies $e_1 \succeq e_2$. Thus we can show that h is also an order quasi-isomorphism. Δ

Just as the relation $\approx$ on $\mathcal{E}$ enables us to speed up searches by eliminating an entire class of expressions with a single test, a suitable relation $\succeq$ on $\mathcal{E}$ can speed up inference by allowing us to eliminate from consideration the class of expressions related to another expression by $\succeq$. If e is found to be too general, i.e., $ASK(e \geq -x?) = 1$ for some negative example $-x$, then all expressions $e' \succeq e$ can be eliminated. Similarly, if e is not general enough — $ASK(e \geq +x?) = 0$ — then all expressions $e' \preceq e$ can be skipped. And as with the $\approx$ relation, we need a finite representation of $\succeq$ in order to work with it.

Henceforth, we assume (unless stated otherwise) that h: $\mathcal{E} \rightarrow D$ is an order homomorphism.

Definition 3.17 *A* downward refinement *of $\mathcal{E}$ is a finitely axiomatizable subset ρ of $\mathcal{E} \times \mathcal{E}$ such that $e_1 \rho e_2$ implies $h(e_1) \geq h(e_2)$.*

We also have the dual definition:

Definition 3.18 *An* upward refinement *of $\mathcal{E}$ is a finitely axiomatizable subset ρ of $\mathcal{E} \times \mathcal{E}$ such that $e_1 \gamma e_2$ implies $h(e_1) \leq h(e_2)$.*

We denote by $\rho(e)$ the set $\{e' \mid e \rho e'\}$. $\rho(e)$ can be recursively enumerated by listing the pairs $e_1 \rho e_2$ and selecting those for which $e_1 = e$. ρ^* represents the reflexive-transitive closure of ρ. Similar remarks apply to $\gamma(e)$ and to γ^*.

Definition 3.19 *Let e be any expression in $\mathcal{E}$. A downward refinement ρ is* complete for *e if $h(\rho^*(e)) = \{d \mid d \leq h(e)\}$. Likewise, γ is* complete for *e if $h(\gamma^*(e)) = \{d \mid d \geq h(e)\}$. A refinement that is complete for every expression in $\mathcal{E}$ is simply termed* complete.

Intuitively, ρ is complete for e if we can obtain at least one representation of every semantic item $d \leq h(e)$ by repeatedly refining e.

Finally, a refinement ρ (or γ) induces a natural ordering $\succeq$ on $\mathcal{E}$ as follows.

Definition 3.20 *If ρ (γ) is a downward (upward) refinement on $\mathcal{E}$, then the associated ordering $\succeq$ ($\preceq$) is ρ^* (γ^*).*

Several easy observations follow from these definitions. First, h is an order homomorphism: if $e_1 \succeq e_2$ then $e_1 \; \rho^* \; e_2$ and hence $h(e_1) \geq h(e_2)$; similarly for $\preceq$. Also, if $e_1 \preceq e_2$ and $e_1 \succeq e_2$ then $h(e_1) \leq h(e_2)$ and $h(e_1) \geq h(e_2)$, so that $e_1 \approx e_2$. Note, however, that $e_1 \succeq e_2$ does *not* imply that $e_2 \preceq e_1$, since these two orderings may be associated with unrelated refinements ρ and γ, respectively. Given a downward refinement, its inverse ρ^{-1} is an upward refinement, and similarly for γ^{-1}. From an axiomatization of ρ we obtain an axiomatization of $\gamma = \rho^{-1}$ by conjoining the axiom:

$$e_1 \, \rho \, e_2 \rightarrow e_2 \, \gamma \, e_1$$

(and dually). Is the inverse of a complete refinement is also complete? Not in general: let $\mathcal{E} = \{e_1, e_{21}, e_{22}, e_{31}, e_{32}\}$, with $e_{21} \approx e_{22}$ and $e_{31} \approx e_{32}$; then if $\rho(e_1) = e_{21}, \rho(e_{21}) = e_{31}, \rho(e_{22}) = e_{32}$, ρ^{-1} is not complete. However, we have the following:

Lemma 3.21 *If ρ is a downward refinement and h is an order quasi-isomorphism for $\succeq$, then ρ and ρ^{-1} are complete refinements. (And dually).*

Proof: Given any $e_1 \in \mathcal{E}$, we let d be any object in D such that $h(e_1) \geq d$. Since h is onto, there exists $e_2 \in \mathcal{E}$ such that $h(e_2) = d$. Since h is an order quasi-isomorphism, $h(e_1) \geq h(e_2)$ implies $e_1 \succeq e_2$, so that $e_2 \in \rho^*(e_1)$. Hence ρ is complete.

Now let $\gamma = \rho^{-1}$. Suppose $d' \geq h(e_1)$. If $h(e_2') = d'$, then we have shown that $\rho^*(e_2')$ contains e_1. Thus $\gamma^*(e_1)$ contains e_2'. Hence γ is also complete. □

Example 3.22 We obtain refinements for Boolean expressions by adding to the axioms in Figure 1 the following:

$$e_1 \cap e_2 \approx e_2 \rightarrow e_1 \, \rho \, e_2$$

$$e_1 \cap e_2 \approx e_2 \rightarrow e_2 \, \gamma \, e_1$$

Having already noted that h is an order quasi-isomorphism for the ordering $e_1 \succeq e_2$ iff $e_1 \cap e_2 \approx e_2$, we conclude from the preceding lemma that ρ and γ are complete.

Similarly, we obtain complete refinements for Regular expressions by adding to the axioms in Figure 2 the following:

$$e_1 + e_2 \approx e_2 \rightarrow e_1 \rho e_2$$
$$e_1 + e_2 \approx e_2 \rightarrow e_2 \gamma e_1$$

△

3.4 Inference Algorithms Using Refinements

Let us now formulate inference algorithms that take advantage of a complete refinement to improve the basic identification-by-enumeration approach. With the ordering comes the notion of upward and downward directions, so algorithms for both directions are given. Note that, unlike the regular-expression example given above, no assumptions are made about the syntactic domain beyond those necessary for the refinement. This generality leads to more complex algorithms. Subsequently we shall add new definitions and assumptions and thereby obtain simpler, more useful, algorithms.

There are two principal difficulties to be overcome. First, although the sets $\gamma(e)$ and $\rho(e)$ are r.e. for any fixed e, they may not be finite sets. So instead of computing all refinements of e and adding them to the queue, the algorithm dovetails the computation of $\gamma(e)$ or $\rho(e)$ with the other iterations. Later we shall assume finiteness properties of refinements and simplify the general algorithm, but for the present we are retaining as much generality as possible.

Second, the existence of an ordering does not imply the existence of largest or smallest elements in the domain $\mathcal{E}$. So instead of selecting a minimal (or maximal) element as the initial hypothesis, as was done above for regular expressions, the algorithm still relies on some arbitrary enumeration of the expressions to generate hypotheses until such time as it is able, using the information about the ordering given in the examples, to locate an expression above (or below) the target. Thereafter it can refine its way up (or down) as for regular expressions. Later we shall assume the existence of bounding elements and simplify the algorithm.

The upward and downward cases are inherently different. The examples tell us expressions e such that $h(e) \leq d_0$, but not expressions e' such that $h(e') \geq d_0$. Hence, in the upward direction, the first positive example is an expression that bounds the target from below and enables the algorithm to ignore all hypotheses but upward refinements of that example. In the downward direction, however, the negative examples may include both expressions which bound the target from above and those which are incomparable to it. The algorithm cannot be sure that there is even one upper bound among the negative examples, and even if there is an upper bound, it cannot identify which examples are upper bounds. Hence it refines downward every expression not known to be too small. There are three enumerations being dovetailed in this algorithm: the enumeration of all expressions $\mathcal{E}$; the enumeration of $\rho(e)$ for particular expressions e; and the succession of refinements $\rho(e), \rho(\rho(e)), \ldots$ for these expressions. Of course, the algorithm would converge with only the first of these (it then is just identification by enumeration), but

with the refinements we expect it to converge more quickly. In summary, the upward direction seems to be more interesting than the downward for the fully general case, when examples lie below ($\leq$) the target.

In the algorithms to follow, the notations $\gamma(e, n)$ and $\rho(e, n)$ are used to denote the expressions, if any, in $\gamma(e)$ and $\rho(e)$ obtained in exactly n computation steps [2]. Although $\gamma(e, n)$ may not yield any expression for a particular value of n, every expression in $\gamma(e)$ is $\gamma(e, n)$ for *some* n (and likewise for $\rho(e, n)$). In the downward case, we extend the notation so that $\rho(e, 0) = e$.

Algorithm 3.23 *Inference by Upward Refinement*

Input: An enumeration $\mathcal{E} = e_1, \ldots$ of the expressions.
A complete upward refinement, γ
An oracle $ASK(e_1 \geq e_2?)$ for $\geq$
An oracle $EX()$ for a sufficient set of examples of d_0

Output: A sequence of expressions $H_1, H_2, \ldots$, such that H_i is correct for the first i examples.

Procedure:
$Q \leftarrow$ *emptyqueue*(). (Queue elements are pairs, $[e, n]$, where $e \in \mathcal{E}$ and $n > 0$)
examples $\leftarrow$ *emptyset*()
$i \leftarrow 1$
$H \leftarrow e_1$ (current hypothesis initialized to first expression in the enumeration)
repeat
  Call $EX()$ for a new example, x
  Add x to *examples*.
  if x is a negative example, **then**
    comment (so far we have gotten only negative examples)
    while $ASK(H \geq e?) = 1$ for any example e
      $i \leftarrow i + 1$
      $H \leftarrow e_i$
    Output H
until x is a positive example.
comment (at last we have a lower bound x for the target and we can start refining our way upward)

[2] A computation step is any suitable discrete measure of computational work, such as a resolution operation on a set of sentences or a single application of the state-transition relation for Turing machines.

$H \leftarrow x$
Output H.
do forever
    Add an example $EX()$ to *examples*
    while H disagrees with some example
        if *too_specific*(H) and **not** *too_general*(H)
        then add $[H, 1]$ to Q. (commence dovetailing the refinement $\gamma(H)$)
        $H \leftarrow$ *next_hypothesis*()
    Output H

where
next_hypothesis() is:
    repeat
        Remove the next entry from the head of Q. Let it be $[e, n]$ (representing $\gamma(e, n)$).
        Add $[e, n+1]$ to Q (to continue the dovetailing)
        $H \leftarrow \gamma(e, n)$ (this may or may not yield an expression)
    until H has a new value

too_specific(H) is:
    if $ASK(H \geq e?) = 0$ for some positive example $+e \in$ *examples*,
    then return **true else** return **false**
too_general(H) is:
    if $ASK(H \geq e?) = 1$ for some negative example $-e \in$ *examples*,
    then return **true else** return **false** ◇

Theorem 3.24 *Algorithm 3.23 identifies d_0 in the limit.*

Proof: The argument combines elements from the proofs of Theorems 2.2 and 3.10. The algorithm tests every hypothesis against every example; with our assumptions about the sufficiency of the examples, we can conclude that eventually any incorrect hypothesis will be discarded.

Suppose the target object d_0 is minimal with respect to $\geq$, so that there may be no positive examples. Then the algorithm will never escape from the **repeat** loop. But this loop coincides with that of Algorithm 3.9, which we already have shown to converge correctly.

Otherwise, a positive example does occur, and becomes the first hypothesis. All subsequent hypotheses are γ-refinements of this expression, which we label E_0. Completeness of γ ensures that there is a smallest integer $n \geq 0$, and a chain of expressions $E_0, \ldots, E_n$ such that $h(E_n) = d_0$, and for all $0 \leq i < n$, $E_i \in \gamma(E_{i-1})$. Recalling that the computation of γ is dovetailed, we know there are integers $k_1, \ldots, k_n$ (assumed to be as small as possible) such that $E_{i+1} = \gamma(E_i, k_{i+1})$ for $0 \leq i < n$. k_{i+1} represents the number of computation steps required to compute E_{i+1} from E_i using γ.

As we did in the proof of Theorem 2.2, we assume the algorithm diverges and argue that E_n must eventually become the current hypothesis. This leads to the contradiction that E_n, a correct hypothesis, must be discarded.

We argue, again by induction, that E_i eventually becomes the hypothesis H, for $0 \leq i \leq n$. For E_0 this is clear. Assume E_i is the hypothesis ($i < n$). Since $E_i \prec E_n$, the only counterexample for which it fails is a positive example (for which E_i will be too specific). Thus $[E_i, 1]$ is added to Q. Q is finite; and since, by assumption, every hypothesis is ultimately discarded, the routine ***next_hypothesis*** is called infinitely often. Hence $[E_i, 1]$ will reach the front of the queue, and $[E_i, 2]$ will be placed on the end. Continuing the argument in this fashion, we conclude that $[E_i, j]$ will be enqueued for all $j \geq 1$ and will make its way to the front of the queue, to be considered as the next hypothesis. In particular, $[E_i, k_{i+1}]$ will be at the front, and the next hypothesis will be $\gamma(E_i, k_{i+1})$, which is just E_{i+1}.

We conclude that E_n will become the next hypothesis. □

Algorithm 3.25 *Inference by Downward Refinement*

Input: An enumeration $\mathcal{E} = e_1, \ldots$ of the expressions.
A complete downward refinement, ρ
An oracle $ASK(e_1 \geq e_2?)$ for $\succeq$
An oracle $EX()$ for a sufficient set of examples of d_0

Output: A sequence of expressions $H_1, H_2, \ldots$, such that H_i is correct for the first i examples.

Procedure:

$Q \leftarrow$ *emptyqueue*(). (Queue elements are pairs, $[e, n]$, where $e \in \mathcal{E}$ and $n > 0$.)
examples $\leftarrow$ *emptyset*().
$i \leftarrow 1$
$H \leftarrow e_1$ (current hypothesis)
do forever
    Add an example $EX()$ to *examples*
    while H disagrees with some example
        $i \leftarrow i + 1$
        Add $[e_i, 0]$ to Q (dovetail the enumeration with the refinements)
        if *too_general*(H) and **not** *too_specific*(H)
            then add $[H, 1]$ to Q (begin refining H)
        $H \leftarrow$ *next_hypothesis*()
    Output H

where

next_hypothesis() is:
    repeat
        Remove the next entry from the head of Q. Let it be $[e, n]$.
        if $n > 0$ **then** add $[e, n + 1]$ to Q (continue dovetailing)
        $H \leftarrow \rho(e, n)$ (this may or may not yield an expression)
    until H has a new value

too_specific(H) is:
    if $ASK(H \geq e?) = 0$ for some positive example $+e \in$ *examples*,
    then return **true else** return **false**

too_general(H) is:
    if $ASK(H \geq e?) = 1$ for some negative example $-e \in$ *examples*,
    then return **true else** return **false** ◇

Theorem 3.26 *Algorithm 3.25 identifies d_0 in the limit.*

Proof: The proof follows closely that for Theorem 3.24, with the following differences. Since we are refining downward, we need an expression representing an upper bound for d_0. But positive examples provide us with only lower bounds, and negative examples do not distinguish the upper bounds from the incomparable expressions. So this particular algorithm relies on the enumeration of $\mathcal{E}$ to produce a bound for d_0.

We are never sure that Q contains an upper bound for d_0, so each iteration of the **while** loop adds another expression from the enumeration of $\mathcal{E}$ to the queue. (This also prevents the queue from being exhausted during the search for a replacement hypothesis.) At some time, therefore, an expression E_0 is placed on Q with the property that $\rho^n(E_0)$ contains a correct hypothesis for d_0, for some n. From this point on, the argument of the preceding theorem applies to show that convergence to a correct hypothesis is necessary.

□

3.5 Improvements

While the above algorithms for inference by refinement are not practical, they are (in a very weak sense, due to [Gold-67]) as good as possible in that no other deterministic inference algorithm is *uniformly faster.* (An algorithm is deterministic if its output is a function only of its input.)

Definition 3.27 *If A and A' are deterministic inference algorithms over the same domain and depend upon the same class of oracles EX for examples, we say that A is* uniformly faster *than A' if: (i) for any target d_0 and any presentation, A requires no more examples than A' before it converges to a correct hypothesis; and (ii) for some d_0 and some presentation, A requires strictly fewer examples than A' to converge to a correct hypothesis.*

Theorem 3.28 (Gold) *There is no deterministic algorithm for identification in the limit that is uniformly faster than Algorithm 3.9. (The same holds for Algorithms 3.23 and 3.25).*

Proof: Let A be an inference algorithm such that, for a target d_0 and a presentation $EX()$, A is faster than Algorithm 3.9. Then if A converges after the n'th example, Algorithm 3.9 converges after the $n+k$'th example, for some $k > 0$. Following the n'th example, Algorithm 3.9 outputs as its guess some expression e such that $h(e) = d_0$, and the presentation of d_0 agrees with that of $EX()$ for the first n examples. This is clearly possible since Algorithm 3.9 outputs hypotheses that are consistent with all examples it has seen. Since both algorithms are deterministic, their input/output behavior will be

identical. But Algorithm 3.9 will converge correctly after the n'th example, and A will converge only after more examples. □

The above result shows that we cannot improve the basic Identification-by-Enumeration algorithm in the worst case, insofar as the number of examples is concerned. Other quality factors, however, can be improved: the number of tests $ASK(e_1 \geq e_2?)$, the size of the queue, and so on. Complexity analysis is not within the scope of this work, but we wish to show how to take advantage of specific properties of the refinements and the ordering to simplify and improve the basic algorithms given above. Ultimately we shall obtain inference algorithms that look very much like the ones for regular expressions. The conclusion of all this, then, is that with a few simple assumptions about the structure of the syntactic representation language $\mathcal{E}$, we obtain both upward and downward inductive inference algorithms, similar to the ones for regular expressions.

In the following, where there is no need to distinguish upward and downward refinements, we shall simply use the term *refinement* generically and denote it by ρ. And where specific examples or algorithms are given, we shall use a specific direction, with the understanding that a similar discussion holds for the other direction.

3.5.1 Separating $\succ$ and $\approx$

The ordering $\succeq$ typically includes at least some elements of the relation $\approx$. Thus the refinement $\rho(e)$ will in general produce new expressions that are semantically equivalent to e. It would be desirable to recognize these, so that algorithms would not have to propose them as hypotheses. For example, suppose the regular expression $(10^* + 01^*)$ is too general, and its refinements include $((10^* + 01^*) + (10^* + 01^*))$ and 10^*. The former is equivalent to the original expression and, like its parent, is too general, so we need not consider it as a hypothesis. It *does* need to be enqueued, however, because its refinements are needed to maintain the completeness property of ρ. But the latter expression is strictly less general than its parent and is therefore a suitable hypothesis.

Definition 3.29 *A refinement ρ is said to be* completely separable *if it can be decomposed into two disjoint refinements $\rho_>$ and $\rho_=$ such that*

$$e_1 \in \rho_>(e_2) \Rightarrow h(e_2) > h(e_1)$$

$$e_1 \in \rho_=(e_2) \Rightarrow h(e_2) = h(e_1)$$

Similarly, for γ the corresponding decomposition is $\gamma_<$ and $\gamma_=$.

In practice, such a decomposition may be difficult, but an approximate decomposition is often nearly as good.

Definition 3.30 *A refinement ρ is said to be* partially separable *if it can be expressed as the union of two (not necessarily disjoint) refinements $\rho_{\geq}$ and $\rho_{=}$ such that*

$$e_1 \in \rho_{\geq}(e_2) \Rightarrow h(e_2) \geq h(e_1)$$

$$e_1 \in \rho_{=}(e_2) \Rightarrow h(e_2) = h(e_1)$$

Similarly, for γ the corresponding decomposition is $\gamma_{\leq}$ and $\gamma_{=}$.

Example 3.31 If $\approx$ is a decidable relation, then any refinement is completely separable: $e_1 \in \rho_{>}(e_2)$ if $e_1 \in \rho(e_2)$ and $e_1 \not\approx e_2$. Such is the case with both Boolean and regular expressions. Δ

Algorithm 3.32 (in the figure below) illustrates the use of the separability of the refinement. It follows the procedure of Algorithm 3.25 with these modifications:

1. Queue entries are triples $[Z, e, n]$, where Z is either "$>$" or "$\approx$", e is an expression, and n is a non-negative integer. $[\approx, e, n]$ represents the expression (if any) obtained in exactly n computation steps of $\rho_{\approx}(e)$. $[>, e, n]$ is similar for $\rho_{>}$.
2. Whenever H is found to be too general, add both $[\approx, H, 1]$ and $[>, H, 1]$ to the queue. This schedules the (dovetailed) enumeration of both $\rho_{\approx}$ and $\rho_{>}$.
3. The procedure *next_hypothesis* is replaced by one which checks the first component (Z) of the triple, and if it is $\approx$, it refines and requeues it without using it as a hypothesis.

3.5.2 Locally Finite Refinements

The complications of the algorithms in the previous sections have resulted in large part from the need to dovetail the calculations of $\rho(e)$ for expressions e with the rest of the inference procedure. If we can find a refinement $\rho(e)$ which yields a finite set for every expression e, and if we can construct a recursive procedure to enumerate that finite set (and halt), then the dovetailing can be eliminated.

Definition 3.33 *A refinement ρ is said to be* locally finite *if, for all $e \in \mathcal{E}$, the set $\rho(e) = \{e' \mid e\ \rho\ e'\}$ is finite.*

Example 3.34 The refinement for Boolean expressions shown in Example 3.22 is not locally finite. For instance, according to the axiom $e_1 \cap e_2 \approx e_2 \rightarrow e_1\ \rho\ e_2$, the infinite set $e, e \cap e, e \cap (e \cap e)$, etc. are all in $\rho(e)$. In Figure 4 we exhibit a different refinement ρ for Boolean expressions. By inspection, ρ is partially separable into $\stackrel{\geq}{\Rightarrow}$ and $\stackrel{=}{\Rightarrow}$, and $\stackrel{\approx}{\Rightarrow}$

Algorithm 3.32 *Downward Inference with a Separable Refinement*

Input: A recursively enumerable set $\mathcal{E} = e_1, \ldots$ of expressions.
A complete, downward, *separable* refinement, $\rho = \rho_{\approx} \cup \rho_{>}$.
An oracle $ASK(e_1 \geq e_2?)$ for $\geq$
An oracle $EX()$ for a sufficient set of examples of d_0

Output: A sequence of expressions $H_1, H_2, \ldots$, such that H_i is correct for the first i examples.

Procedure:
$Q \leftarrow$ *emptyqueue*(). (Queue elements are triples, $[Z, e, n]$. See text discussion. $[>, e, 0]$ represents e.)
examples $\leftarrow$ *emptyset*().
$i \leftarrow 1$
$H \leftarrow e_1$ (current hypothesis)
do forever
  Add an example $EX()$ to *examples*
  while H disagrees with some example
    $i \leftarrow i + 1$
    add $[>, e_i, 0]$ to Q (dovetail the enumeration)
    if *too_general*(H) and **not** *too_specific*(H)
      then add $[\approx, H, 1]$ and $[>, H, 1]$ to Q (refine H)
    $H \leftarrow$ *next_hypothesis*()
  Output H

where
next_hypothesis() is:
  repeat
    $[Z, e, n] \leftarrow$ *next*(Q)
    Add $[Z, e, n + 1]$ to Q (to continue dovetailing)
    if Z is "$\approx$"
    then add $[>, \rho_{\approx}(e, n), 1]$ and $[\approx, \rho_{\approx}(e, n), 1]$ to Q, provided $\rho_{\approx}(e, n)$ is an expression (continue refining)
    else (Z is "$>$") $H \leftarrow \rho_{>}(e, n)$
  until H has a new value
too_specific and *too_general* are as in Algorithm 3.25. ◇

Figure 3: Downward Inference with Separable Refinement

is a subset of $\approx$ shown in Figure 1. The relation $\stackrel{\geq}{\Rightarrow}$ includes some elements of $\approx$, since $x_i \cap \mathbf{false} \stackrel{\geq}{\Rightarrow} \mathbf{false} \cap \mathbf{false}$, even though these two expressions are equivalent ($\approx$).

It is tedious but not difficult to check that the two refinements in Figures 1 and 4 give rise to the same ordering $\succeq$. One must show that every rule in one figure can be proved in the other, and conversely.

The proof that ρ is locally finite consists in showing that both $\stackrel{\sim}{\Rightarrow}$ and $\stackrel{\geq}{\Rightarrow}$ are locally finite. This in turn is a mostly straightforward induction on the size of the expression being refined. The finiteness of $\stackrel{\geq}{\Rightarrow}$ for expressions of the form $\sim e$ presents the only difficulty. We assume that e is of size k and inductively that $\rho_>(e)$ is finite. ($\rho_>(e)$ is the set $\{e' \mid e \stackrel{\geq}{\Rightarrow} e'\}$.) From the axiom

$$e_1 \stackrel{\geq}{\Rightarrow} e_1' \rightarrow \sim e_1' \stackrel{\geq}{\Rightarrow} \sim e_1$$

the set $\rho_>(\sim e)$ consists of all expressions $\sim e'$ such that $e' \stackrel{\geq}{\Rightarrow} e$. But with the observation that each refinement rule for $\stackrel{\geq}{\Rightarrow}$ preserves the size of the expression, we know that e' and e are the same size. Thus the number of expressions $\sim e'$ such that $e \stackrel{\geq}{\Rightarrow} e'$ must be finite. Δ

Henceforth, when we say that a refinement is "locally finite," we shall implicitly include the condition that there is an algorithm to compute $\rho(e)$ for any expression e.

Algorithm 3.35 (below) follows the procedure of Algorithm 3.32 with these modifications:

1. Queue entries are simply expressions $e \in \mathcal{E}$.
2. Instead of adding $[e_i, 0]$ to Q, simply add e_i.
3. Instead of adding $[e_i, 1]$ to Q, add each expression in the (finite) set $\rho(e_i)$ to Q.
4. The procedure *next_hypothesis* is now simply: Remove the front expression from Q and make it the current hypothesis, H.

3.5.3 Bounding Expressions

As noted above, another major source of complication in the basic Algorithms 3.23 and 3.25 is the need to find an expression which bounds the target: $h(e) \leq d_0$ in the former case, $h(e) \geq d_0$ in the latter. Having done so, the algorithm proceeds to refine e and test the resulting expressions in sequence against the examples.

Oftentimes one can write down an expression which is most (least) general: for example, **true** or **false**, in the case of Boolean expressions. If we begin with such an expression

Note: The refinement ρ is the union of $\stackrel{\geq}{\Rightarrow}$ and $\stackrel{\sim}{\Rightarrow}$. $\rightarrow$ is a logical symbol of the axiomatization denoting implication. $\cap, \cup, \sim$ are the Boolean operations. $x_1, \ldots, x_n$, **true** and **false** are the Boolean constants. e_1, e_1', etc. are variables of the axiomatization representing arbitrary Boolean expressions.

ρ

$e_1 \stackrel{\geq}{\Rightarrow} e_2 \quad \rightarrow \quad e_1 \rho e_2$
$e_1 \stackrel{\sim}{\Rightarrow} e_2 \quad \rightarrow \quad e_1 \rho e_2$

$\stackrel{\geq}{\Rightarrow}$

$\mathbf{true} \stackrel{\geq}{\Rightarrow} x_i$ (for $1 \leq i \leq n$)
$x_i \stackrel{\geq}{\Rightarrow} \mathbf{false}$
$e_1 \stackrel{\geq}{\Rightarrow} e_1' \quad \rightarrow \quad e_1 \cap e_2 \stackrel{\geq}{\Rightarrow} e_1' \cap e_2$
$e_1 \stackrel{\geq}{\Rightarrow} e_1' \quad \rightarrow \quad e_1 \cup e_2 \stackrel{\geq}{\Rightarrow} e_1' \cup e_2$
$e_1 \stackrel{\geq}{\Rightarrow} e_1' \quad \rightarrow \quad \sim e_1' \stackrel{\geq}{\Rightarrow} \sim e_1$

$\stackrel{\sim}{\Rightarrow}$

$e_1 \cap e_2 \stackrel{\sim}{\Rightarrow} e_2 \cap e_1$ | $e_1 \cup e_2 \stackrel{\sim}{\Rightarrow} e_2 \cup e_1$

$e_1 \cap (e_1 \cup \mathbf{true}) \stackrel{\sim}{\Rightarrow} e_1$ | $e_1 \cup (e_1 \cap \mathbf{false}) \stackrel{\sim}{\Rightarrow} e_1$
$e_1 \cap (e_1 \cup x_i) \stackrel{\sim}{\Rightarrow} e_1$ | $e_1 \cup (e_1 \cap x_i) \stackrel{\sim}{\Rightarrow} e_1$
$e_1 \cap (e_1 \cup \mathbf{false}) \stackrel{\sim}{\Rightarrow} e_1$ | $e_1 \cup (e_1 \cap \mathbf{true}) \stackrel{\sim}{\Rightarrow} e_1$

$e_1 \cap (e_2 \cup e_3) \stackrel{\sim}{\Rightarrow} (e_1 \cap e_2) \cup (e_1 \cap e_3)$ | $e_1 \cup (e_2 \cap e_3) \stackrel{\sim}{\Rightarrow} (e_1 \cup e_2) \cap (e_1 \cup e_3)$

$\mathbf{true} \cap \mathbf{false} \stackrel{\sim}{\Rightarrow} \mathbf{false}$ | $\mathbf{true} \cup \mathbf{true} \stackrel{\sim}{\Rightarrow} \mathbf{true}$
$x_i \cap \mathbf{false} \stackrel{\sim}{\Rightarrow} \mathbf{false}$ | $x_i \cup \mathbf{true} \stackrel{\sim}{\Rightarrow} \mathbf{true}$
$\mathbf{false} \cap \mathbf{false} \stackrel{\sim}{\Rightarrow} \mathbf{false}$ | $\mathbf{false} \cup \mathbf{true} \stackrel{\sim}{\Rightarrow} \mathbf{true}$
$\mathbf{true} \cap \sim\mathbf{true} \stackrel{\sim}{\Rightarrow} \mathbf{false}$ | $\mathbf{true} \cup \sim\mathbf{true} \stackrel{\sim}{\Rightarrow} \mathbf{true}$
$x_i \cap \sim x_i \stackrel{\sim}{\Rightarrow} \mathbf{false}$ | $x_i \cup \sim x_i \stackrel{\sim}{\Rightarrow} \mathbf{true}$
$\mathbf{false} \cap \sim\mathbf{false} \stackrel{\sim}{\Rightarrow} \mathbf{false}$ | $\mathbf{false} \cup \sim\mathbf{false} \stackrel{\sim}{\Rightarrow} \mathbf{true}$

$e_1 \stackrel{\sim}{\Rightarrow} e_1' \quad \rightarrow \quad e_1' \stackrel{\sim}{\Rightarrow} e_1$
$e_1 \stackrel{\sim}{\Rightarrow} e_1' \quad \rightarrow \quad e_1 \cap e_2 \stackrel{\sim}{\Rightarrow} e_1' \cap e_2$
$e_1 \stackrel{\sim}{\Rightarrow} e_1' \quad \rightarrow \quad e_1 \cup e_2 \stackrel{\sim}{\Rightarrow} e_1' \cup e_2$
$e_1 \stackrel{\sim}{\Rightarrow} e_1' \quad \rightarrow \quad \sim e_1 \stackrel{\sim}{\Rightarrow} \sim e_1'$

Figure 4: Locally finite refinement for Boolean expressions

Algorithm 3.35 *Downward Inference with a Locally Finite, Separable Refinement*

Input: A recursively enumerable set $\mathcal{E} = e_1, \ldots$ of expressions.
A complete, downward, *locally finite* refinement, $\rho = \rho_{\approx} \cup \rho_{>}$.
An oracle $ASK(e_1 \geq e_2?)$ for $\geq$
An oracle $EX()$ for a sufficient set of examples of d_0

Output: A sequence of expressions $H_1, H_2, \ldots$, such that H_i is correct for the first i examples.

Procedure:
$Q \leftarrow$ *emptyqueue*(). (Queue elements are expressions.)
examples $\leftarrow$ *emptyset*().
$i \leftarrow 1$
$H \leftarrow e_1$ (current hypothesis)
do forever
  Add an example $EX()$ to *examples*
  while H disagrees with some example
    $i \leftarrow i + 1$
    add e_i to Q. (continue the enumeration)
    if *too_general*(H) and **not** *too_specific*(H)
      then add the expressions $\rho(H)$ to Q.
    $H \leftarrow$ *next*(Q).
  Output H

where
too_general and *too_specific* are as in Algorithm 3.25. ◇

Figure 5: Downward Inference with Locally Finite Refinement

as the current hypothesis, the algorithm can proceed without having to enumerate $\mathcal{E}$. The refinement can also be simplified, since it need be complete only for the starting expression, not all expressions.

Definition 3.36 *An expression $\bar{e}$ is said to be* maximal *if there is no $e \in \mathcal{E}$ for which $h(e) > h(\bar{e})$. $\bar{e}$ is said to be a* top element *of $\mathcal{E}$ if $h(\bar{e}) \geq h(e)$ for all $e \in \mathcal{E}$.*

Dual definitions apply to minimal *and* bottom *expressions.*

Example 3.37 Let Σ and V be finite, disjoint alphabets. V will be interpreted as a set of string variables. We define the class of *pattern languages* over (Σ, V) ([Angluin-80]) as follows. A *pattern* is a string in $(\Sigma \cup V)^+$. The language $L(p)$ generated by the pattern p is the set of strings derived from p by non-erasing homomorphisms from V to Σ^+. The class of pattern languages over (Σ, V) is the class of languages generated by patterns over those alphabets.

Let $\mathcal{E}$ be the set of patterns, and $\mathcal{D}$ the languages they generate, ordered by inclusion. Any single-variable pattern v is a top element for $\mathcal{D}$, since it generates Σ^+. By contrast there is no bottom element: indeed, there is no finite set of minimal elements, since all strings in Σ^+ are minimal. △

The following is a slight generalization of Definition 3.19.

Definition 3.38 *Let $M = \{\bar{e}_1, \ldots, \bar{e}_n\}$ be a finite set of maximal expressions, The refinement ρ is said to be* complete for M *if $h(\bigcup_i \rho^*(\bar{e}_i)) = \mathcal{D}$. That is, $\bigcup_i \rho^*(\bar{e}_i)$ includes at least one expression for every $d \in \mathcal{D}$.*

Algorithm 3.40 (below) follows the procedure of Algorithm 3.32 with these modifications:

1. Initially $H \leftarrow \bar{e}_1$ and $Q \leftarrow \{\bar{e}_2, \ldots, \bar{e}_s\}$.
2. Steps concerned with the enumeration of $\mathcal{E}$ are unnecessary.

Note that this algorithm is essentially that of Algorithm 2.3.

Example 3.39 The regular expression refinements used in Section 2 are both locally finite and complete (modulo equivalence) for ϕ upward and Σ^* downward. They can be partially separated if the rules $E^* \rightarrow E$ and $\sigma_i \rightarrow \phi$ are made into a separate relation $\stackrel{\succ}{\rightarrow}$. The algorithms of that section take advantage of all three of these properties and can be viewed as a fusion of Algorithms 3.32, 3.35, and 3.40. △

Algorithm 3.40 *Downward Inference, given a "Top" Expression*

Input: A recursively enumerable set $\mathcal{E} = e_1, \ldots$ of expressions.

A locally finite, downward refinement complete (modulo $\approx$) for a set $M = \bar{e}_1, \ldots, \bar{e}_s$ of maximal expressions.

An oracle $ASK(e_1 \geq e_2?)$ for $\geq$

An oracle $EX()$ for a sufficient set of examples of d_0

Output: A sequence of expressions $H_1, H_2, \ldots$, such that H_i is correct for the first i examples.

Procedure:

$Q \leftarrow \bar{e}_2, \ldots, \bar{e}_s$

examples $\leftarrow$ *emptyset*().

$H \leftarrow \bar{e}_1$ (current hypothesis)

do forever

Add an example $EX()$ to *examples*

while H disagrees with some example

if *too_general*(H) and **not** *too_specific*(H)

then add the expressions $\rho(H)$ to Q.

$H \leftarrow$ *next*(Q).

Output H

where

too_general and *too_specific* are as in Algorithm 3.25. ◇

Figure 6: Downward Inference with Bounding Expressions

Finally, let us take note of the kinship between refinements and the class of axiom schemes known as Term Rewriting Systems (TRS). ([Huet-80] presents a good overview of this topic.) TRS's have been applied to such problems as code optimization, automatic theorem proving, and the implementation of abstract data types.

A TRS consists of a set of rewrite rules of the form $\alpha \Rightarrow \beta$, representing the substitution of an occurrence of β for one of α in an expression ("term"). In its most general form, a TRS is a type-0 grammar ([Hopcroft-79]), but the interesting results occur when restrictions are imposed upon the reductions. Among the desirable properties of such systems are those of being *Noetherian* and *confluent*. A relation $\Rightarrow$ is Noetherian if there are no infinite chains $x_1 \Rightarrow \cdots \Rightarrow x_n \Rightarrow \cdots$, and confluent if, for all x, if y_1 and y_2 exist so that $x \stackrel{*}{\Rightarrow} y_1$ and $x \stackrel{*}{\Rightarrow} y_2$, then there is a term z such that $y_1 \stackrel{*}{\Rightarrow} z$ and $y_2 \stackrel{*}{\Rightarrow} z$. Relations with both properties have the *Church-Rosser* property: for every x there is unique $\hat{x}$ such that $x \stackrel{*}{\Rightarrow} \hat{x}$ and $\hat{x}$ is irreducible with respect to $\Rightarrow$. Among the interesting problems associated with TRS's is that of determining when a system has the Church-Rosser property, for then there is a decision procedure to decide whether two terms are interconvertible. Knuth and Bendix ([Knuth-70]) gave a partial decision procedure for deriving a confluent reduction system from equational axioms. When it converges, the procedure yields an effective decision procedure for equational theories.

A refinement ρ is also a form of rewrite system in which one expression is replaced by another in $\rho(e)$. Most refinements are not Noetherian, for most expressions have an infinite chain of reductions (e.g., $e \Rightarrow e{+}e \Rightarrow e{+}e{+}e \Rightarrow$ etc.). But when ρ^{-1} is Noetherian, there is a complete set of maximal expressions, from which any other expression can be derived. And when ρ^{-1} has the Church-Rosser property and the (graph of the) relation ρ is connected, it is easily shown that there is a unique top element.

Viewing ρ as a TRS raises a number of interesting questions. Given a complete, separable refinement $\rho = \rho_{>} \cup \rho_{\approx}$, can we use the Knuth-Bendix procedure on ρ to obtain a decision algorithm for $\succeq$? If so, then the oracle $ASK(e_1 \geq e_2?)$ can be replaced by such an algorithm. Even if not, the procedure might suggest ways to obtain partial refinements for maximal expressions which are more efficient. In practice, we find that $\rho_{>}^{-1} \cup \rho_{\approx}$ is an upward refinement (assuming ρ is a downward refinement), but it may not be complete, even if ρ is. An interesting problem would be to characterize the types of reductions for which completeness and other properties (e.g., local finiteness) are preserved by this transformation.

4 Applications to Logic

The definitions and the algorithms described heretofore apply generally to structures with a suitable relationship between the syntax and the semantics. We now specialize the development to the domain of first-order logic. In the bargain, we shall

- re-interpret and generalize Shapiro's model-inference algorithm.
- characterize all possible refinements over a given first-order domain.
- obtain an inference algorithm that takes advantage of normal-form properties.

4.1 Inferring Classes of Propositional Models

We adopt a standard notation for the symbols of our first-order languages; this usage supersedes that of preceding chapters. Predicate symbols will be denoted $p, p_1, p_2, \ldots$ and $q, q_1, q_2, \ldots$. Non-constant function symbols are drawn from $f, f_1, f_2, \ldots$ and $g, g_1, g_2, \ldots$. Constants will be indicated by $c, c_1, c_2, \ldots$. We consider only first-order languages with at least one constant and at least one predicate symbol. Predicate symbols of zero arity (propositions) are equivalent to the "Boolean variables" we used before. Logical variables will be drawn from $x, x_1, x_2, \ldots$ and $y, y_1, y_2, \ldots$. We use the symbols $\wedge, \vee, \sim, \rightarrow$, and $\leftrightarrow$ to denote the connectives for conjunction, disjunction, negation, implication, and equivalence, respectively. Sometimes the $\wedge$ will be implicit in a juxtaposition, especially in DNF expressions such as $p_1p_2 \vee q_1q_2$. The traditional rules for operator precedence are assumed. Finally, clauses of the form $\alpha_1 \vee \cdots \vee \alpha_p \vee \sim\beta_1 \vee \cdots \vee \sim\beta_n$ will usually be written $\alpha_1, \ldots, \alpha_p \leftarrow \beta_1, \ldots, \beta_n$.

Consider anew the problem of inferring a propositional formula (Boolean expression). Fix a language – in this case, a finite set of propositions $P = \{p_1, \ldots, p_t\}$ – and take as the semantic domain $\mathcal{D}$ the class of all *sets* of assignments $A : P \rightarrow \{true, false\}$, 2^{2^t} in number. The syntactic domain $\mathcal{E}$ is that of well-formed logical formulas using the propositions P and the connectives. The meaning $h(\varphi)$ of a formula φ is the set of assignments which satisfy it. $h^{-1}(\emptyset)$ is the set of contradictory (logically false) formulas, and if S is the set of all assignments to the variables, then $h^{-1}(S)$ is the set of tautologies. $\mathcal{D}$ is partially ordered by set inclusion. The refinement of Example 3.22, rewritten in the current notation, is: $e_1 \in \gamma(e)$ iff $\vdash (e_1 \wedge e) \leftrightarrow e$. Equivalently, $e_1 \in \gamma(e)$ iff $\vdash (e \rightarrow e_1)$. Logical implication is reflexive and transitive, so $\preceq$ is precisely the implication relation, and h is an order quasi-isomorphism.

Let d_0 be a set of assignments to P. Examples of d_0 usually take the form of minterms (conjunctions of t literals , one for each propositional symbol, such as $p_1 \wedge \sim p_2 \wedge \cdots \wedge p_t$), or maxterms (disjunctions of t literals). Minterms are precisely the formulas denoting a single assignment; maxterms are those satisfied by all but one assignment. Consequently the set of minterms (resp. maxterms) are a complete set of examples for any d_0. More formally:

Proposition 4.1 *Let e_1 and e_2 be propositional expressions such that, for any minterm m, $e_1 \rightarrow m$ is valid iff $e_2 \rightarrow m$ is valid. Then e_1 and e_2 are logically equivalent.*

Proof: Let $m_1, \ldots, m_k$ be the finite set of minterms implied by e_1 and e_2. Then e_1 and e_2 are both logically equivalent to the DNF expression $m_1 \vee \cdots \vee m_k$ and hence to each other. □

Similarly,

Proposition 4.2 *Let e_1 and e_2 be propositional expressions such that, for any maxterm M, $M \rightarrow e_1$ is valid iff $M \rightarrow e_2$ is valid. Then e_1 and e_2 are logically equivalent.*

4.2 Inferring Classes of First-Order Models

These ideas generalize to first-order languages. A sentence φ in a language $\mathcal{L}$ can be thought of as denoting a set of Herbrand models which satisfy it. If φ is contradictory, it has no models; if tautologically valid, it is satisfied in every (Herbrand) model [3]. If H is the set of ground (variable-free) atomic formulas constructible from the non-logical symbols of $\mathcal{L}$, then the semantic domain of inference $\mathcal{D}$ is the class 2^{2^H} of classes of subsets of H. For example, the sentence $\varphi = \forall x_1 p(x_1) \wedge \forall x_2 \sim q(f(x_2))$ denotes those subsets of H which, for every ground term t, contain $p(t)$ and do not contain $q(f(t))$. In general $\mathcal{D}$ may be uncountable, and most subsets of H have no first-order representation in $\mathcal{L}$. The mapping $h : \mathcal{E} \rightarrow \mathcal{D}$ carries a sentence to the set of models in which it is satisfied.

As with propositional logic, $\mathcal{D}$ is partially ordered by set inclusion. With the refinement $e_1 \in \gamma(e)$ iff $\vdash e \rightarrow e_1$, $\mathcal{E}$ is ordered by implication. (More useful refinements are considered below.) As in the propositional case, h is an order homomorphism.

Let d_0 be a class of models. How do we present examples of d_0? Following the propositional analogy, we could present sentences which have exactly one model; if that model is in d_0 the sentence is a positive example, and a negative example otherwise. Or the dual approach is possible: present sentences satisfied by all but one model, with the example being negative iff that model is in d_0. But the sentences with unique models are those which are minimal with respect to $\preceq$, i.e., , for which there is no inequivalent sentence ψ such that $\psi \rightarrow \varphi$. It can be shown, by reduction from Church's Theorem, that this set is not recursively enumerable. And whether this set constitutes a complete set of examples for a class of models is not clear. So this approach to examples for classes of models is not very promising.

Fortunately, there is another way. Following Shapiro, we assume that the language $\mathcal{L}$ is expressive enough that we need consider only sentences in *clause form*: conjunctions of

[3] Henceforth whenever we refer to a model, we shall implicitly mean a Herbrand model.

universally quantified clauses. [4] This restriction to clause-form sentences is convenient for the use of formal resolution proofs. The following theorem then shows that *the ground clauses are a sufficient set of examples for any (definable)* $d_0 \in \mathcal{D}$.

Definition 4.3 *Let* φ *be a sentence in clause form. The* ground implicands *of* φ *are the set* $C_0^+[\varphi]$ *of ground clauses implied by* φ. *We denote the complement of* $C_0^+[\varphi]$ *by* $C_0^-[\varphi]$.

Example 4.4 The ground implicands of $\forall x(p(c,x) \vee \sim p(x,c)) \wedge q(c)$ are $q(c)$ and clauses of the form $p(c,t) \vee \sim p(t,c)$, where t is a ground term in H. △

Theorem 4.5 *In a first-order language* $\mathcal{L}$, *let* φ_1 *and* φ_2 *be clause-form sentences. And for any sentence* φ *in* $\mathcal{L}$, *let* $h_m(\varphi)$ *be the class of models in which* φ *is satisfied. Then* $h_m(\varphi_1) \supseteq h_m(\varphi_2)$ *iff* $C_0^+[\varphi_1] \subseteq C_0^+[\varphi_2]$. *(Roughly, the number of models of a sentence has an inverse relationship with the number of ground implicands.)*

Proof: Assuming that any model of φ_2 is a model of φ_1, suppose $\varphi_1 \vdash c$ where $c \in C_0^+[\varphi_1]$. Let m be any model of φ_2. Then m is a model of φ_1 and hence verifies c. Thus $\varphi_2 \vdash c$.

In the other direction, suppose $C_0^+[\varphi_1] \supseteq C_0^+[\varphi_2]$. We show below that $\models \varphi_2 \rightarrow \varphi_1$. It then follows that any model of φ_2 is a model of φ_1, i.e., , $h_m(\varphi_2) \subseteq h_m(\varphi_1)$.

To see the claim, suppose that $\varphi_2 \not\vdash \varphi_1$. φ_1 is in clause form, so we can write $\varphi_1 = \forall \bar{x} c_1 \wedge \cdots \wedge \forall \bar{x} c_n$. Let m be a model in which φ_2 is true but not φ_1. Then some $\forall \bar{x} c_i$ is false in m. If c_i contains any of the variables $\bar{x}$, then there is a ground instance c_i^0 of c_i which is false in m; else, c_i is a ground clause, and we take $c_i^0 = c_i$. In either case, $\varphi_1 \vdash c_i^0$. Clearly $\varphi_2 \not\vdash c_i^0$, since then $m \models c_i^0$. But with c_i^0 we have a counterexample to the assumption that the ground implicands of φ_1 include those of φ_2. This proves the claim, and the lemma. □

Corollary 4.6 *A first-order clause-form formula is determined, up to equivalence, by its ground implicands. That is, for any two clause-form formulas* φ_1 *and* φ_2, $C_0^+[\varphi_1] = C_0^+[\varphi_1]$ *iff* $\models (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$.

This theorem and its corollary tell us that we can use either the class of models satisfying φ or the set of ground implicands as the semantic interpretation of the clause-form sentence φ. With the former, the ordering $\varphi_1 \preceq \varphi_2$ says that $\varphi_1 \rightarrow \varphi_2$; with the latter, $\varphi_2 \rightarrow \varphi_1$; so the two interpretations are duals. The class of models is perhaps more intuitive, but it suffers one disadvantage: by using ground clauses as examples, we are inferring sentences that are less general ($\preceq$) than the examples (ground clauses

[4] Recall that a *clause* is a formula $l_1 \vee \cdots \vee l_n$, where $n \geq 1$ and each l_i is an atomic formula or the negation of an atomic formula.

1. Define relatives	$relative(father(x), x)$ $relative(mother(x), x)$ $relative(x, y) \leftarrow relative(y, x)$ $relative(x, y) \leftarrow relative(x, z), relative(z, y)$ $relative(x, x)$
2. Define inlaws	$inlaw(spouse(x), x)$ $inlaw(x, y) \leftarrow inlaw(y, x)$ $inlaw(x, y) \leftarrow relative(x, z), inlaw(z, y)$
3. No inlaws are relatives	$\leftarrow inlaw(x, y), relative(x, y)$
4. Relatives are wrong if George is a loser	$wrong(x) \leftarrow loser(\text{George}), relative(x, \text{George})$
5. Inlaws are wrong if George is a loser	$wrong(x), loser(\text{George}) \leftarrow inlaw(x, \text{George})$
6. No relative is a loser except George	$x \doteq \text{George} \leftarrow relative(x, \text{George}), loser(x)$
7. No inlaw is a loser	$\leftarrow inlaw(x, \text{George}), loser(x)$

Figure 7: Example Axioms

have more models). In adapting the algorithms from the preceding sections, in which the examples are smaller than the expressions they represent, we would have to use the dual ordering relation ($x \succeq' y$ iff $x \preceq y$) and switch the roles of top and bottom, upward and downward, etc. To avoid this confusion, we shall instead adopt the interpretation $h(\varphi) = C_0^+[\varphi]$, so that $\boxed{\varphi_1 \succeq \varphi_2 \text{ iff } \varphi_1 \rightarrow \varphi_2.}$

Example 4.7 Consider the following situation. George has a problem in inter-familial politics. His inlaws think he is a born loser, while his relatives disagree. For our part, as neutral observers, we neither know nor care about George's inherent worth; we do, however, want to account axiomatically for the situation. So we define a sentence whose models include both the possibility that George is a born loser and that he is not; in the former, his relatives are wrong, and in the latter, his inlaws.

One set of axioms is given in Figure 7. The predicate $relative(x, y)$ is intended to mean that x is a relative of y. The predicate $inlaw$ is read similarly. The predicate $\doteq$ indicates equality. The other predicates and function symbols are easy to interpret. The sentence axiomatizing the class of models is, of course, the conjunction of the clauses

given.

Supposing that these axioms represent our hypothesis about George's predicament, we should seek some examples to test it. One confirming example would be the fact that either George's mother or his mother-in-law is wrong, since the ground clause

$$\mathit{wrong}(\mathit{mother}(\text{George})) \vee \mathit{wrong}(\mathit{mother}(\mathit{spouse}(\text{George})))$$

is a consequence of the axioms (as the reader can easily verify).

We would not expect to enounter $\mathit{loser}(\text{George})$ or $\sim\mathit{loser}(\text{George})$ as an example, since our axioms allow models in which the former is true and models in which the latter is true.

Another consequence of our axioms is that George does not believe he is a loser; specifically, the ground clause

$$\mathit{wrong}(\text{George}) \leftarrow \mathit{loser}(\text{George})$$

is provable from the axioms. Thus models in which George is a loser but not wrong (presumably because he is aware of his own limitations) are excluded. If we find we need to include such models, then the above clause is a negative example, and we would change the axioms accordingly. Possible changes include revising $\mathit{relative}(x, x)$ so that George is not a relative of himself, or adding the possible conclusion, $x \doteq \text{George}$, to the fourth axiom. △

4.3 Inferring Theories: Comparison with Shapiro's Approach

In the results to follow, we shall obtain refinements for clause-form sentences and combine them with the preceding results to obtain an algorithm that infers classes of models from examples of their ground implicands. Before jumping into this, however, let us compare the approach in [Shapiro-81], which differs in several significant ways. Most important is that Shapiro's domain of inference is the set 2^H of Herbrand models, whereas our domain is classes of models, 2^{2^H}.

In Shapiro's system, there is a single Herbrand model M; the task of the algorithm is to infer a clause-form sentence φ such that M is the unique smallest model of φ. Such a sentence has the property that the set of ground atoms α such that $\varphi \vdash \alpha$ is precisely the set of elements of the Herbrand base H that are in M. Thus the ground atoms form a complete set of examples. (Note that M is not necessarily the unique model of φ. For example, any model that includes the atom $p(a)$ is a model of the sentence $p(a)$. The model $\{p(a)\}$ is merely the smallest one.)

We can view this in another way. A *theory* is a set of sentences in $\mathcal{L}$ that is closed under logical deduction. A theory $\mathcal{T}$ is *complete* iff, for every sentence φ in $\mathcal{L}$, either $\varphi \in \mathcal{T}$ or $\sim\varphi \in \mathcal{T}$. If M is a model, the *theory of* M, denoted $\mathrm{Th}(M)$, is the set of

sentences true in M. More generally, the theory of a set of models happens to be the set of sentences true in all models in the set. The theory of a single model is obviously complete. The theory of a set of models may or may not be complete; but if the set of models is closed under intersection, its theory is the theory of the smallest model in the set, which is complete. This is the case for Shapiro's system: it infers axiomatizations for complete theories only.

The problem of inferring a more general class of models can also be viewed as a process of inferring an axiomatization for a theory, but the theory need not be complete. Positive examples are ground *clauses* in the theory, and negative examples are ground clauses not in the theory. Incomplete theories are useful when one does is unable or unwilling to specify the truth value of every ground atom in H.

The principal application of Shapiro's system is to the inference of programs. Often the meaning of a program is well specified, to the point that one can state whether the output of a program for a given input is right or wrong. So program inference is a good paradigm for the inference of complete theories.

But many situations call for predicates with "don't-care" outputs. Or, we may care about the output of a program only for a restricted range of inputs. In this situation, an incomplete theory may be more appropriate. Another situation where incomplete theories may be more useful is in incorporating *theoretical terms.* These are concepts based on predicates that are not directly observable. For example, the theory of elementary particles currently depends on the concept of quarks, particles which are not directly observable but whose assumed properties have implications for quantities which are observable. Ideally, theoretical physicists are content with any set of axioms that correctly implies all known observables; whether quarks or zorks are part of that axiomatization is immaterial. So the class of models which validate the examples is the class of interest. Generalizing Shapiro's algorithms to incomplete theories is perhaps a necessary step toward the incorporation of theoretical terms into the inference process, but it is not by itself sufficient because the first-order language must already contain the predicates to express these concepts, and the algorithms axiomatize a specific class of models, not any model in an admissible class.

4.4 A Refinement for Clause-Form Sentences

Definition 4.8 *For a given first-order language $\mathcal{L}$, the set of* most-general terms *consists of the constants and the terms of the form $f(x_1, \ldots, x_n)$, where f is an n-place function symbol. The set of* most-general literals *consists of literals of the form $p(x_1, \ldots, x_n)$ and $\sim p(x_1, \ldots, x_n)$, where p is an n-place predicate symbol (for $n \geq 0$).*

Definition 4.9 *Let $\kappa = \alpha_1, \cdots \alpha_p \leftarrow \beta_1, \cdots, \beta_n$ be a clause in a language $\mathcal{L}$, where the α's and β's are positive literals and the clause κ is, implicitly, universally quantified over its variables. Then $\rho_c(\kappa|\mathcal{L})$ is the set of clauses in $\mathcal{L}$ derived from κ by exactly one of the following operations:*

$\rho_c(1)$: unifying two distinct variables x and y occurring in κ (i.e., , replacing all occurrences of y in κ by x).

$\rho_c(2)$: substituting for all occurrences of a variable x occurring in κ a most-general term t such that no variable in t occurs elsewhere in κ.

$\rho_c(3)$: disjoining a most-general literal, α_{p+1} or $\sim\beta_{n+1}$, to κ, such that no variable in the new literal occurs elsewhere in κ.

Example 4.10 Let $\kappa = p(x_1, f(y_1)) \leftarrow p(f(x_1), y_1)$. In a language $\mathcal{L}$ which includes the two-place function symbol g and a one-place predicate symbol q, $\rho_c(\kappa|\mathcal{L})$ includes the clauses:

$$p(x_1, f(x_1)) \leftarrow p(f(x_1), x_1)$$
$$p(g(x_2, y_2), f(y_1)) \leftarrow p(f(g(x_2, y_2)), y_1)$$
$$p(x_1, f(y_1)), q(x_2) \leftarrow p(f(x_1), y_1)$$
$$p(x_1, f(y_1)) \leftarrow p(f(x_1), y_1), q(x_2).$$

△

Note that $\rho_c(\kappa|\mathcal{L})$ is finite if we do not distinguish clauses differing only in the order in which the literals occur, in having multiple occurrences of the same literal, or in bijectively renaming the variables. Indeed, we shall consider such *variants* as syntactically identical.

The relation $\rho_c(\kappa|\mathcal{L})$ is essentially the "most-general refinement operator ρ" as defined in [Shapiro-81].

Lemma 4.11 *Over the language $\mathcal{L}$, ρ_c is a downward refinement for clauses complete for $\square$, the empty clause. That is, for any clause κ in $\mathcal{L}$, there is a clause κ' in $\rho_c^*(\square|\mathcal{L})$ such that κ' is a variant of κ.*

Proof: See [Shapiro-81].

Definition 4.12 *Let $\varphi = \kappa_1 \wedge \cdots \wedge \kappa_r$ be a clause-form sentence in the language $\mathcal{L}$, such that no variable occurs in more than one clause. The set $\hat{\rho}(\varphi|\mathcal{L})$ is the set of clause-form sentences derived from φ by exactly one of the following operations:*

$\hat{\rho}(1)$: deleting a clause κ_i.

$\hat{\rho}(2)$: conjoining a new clause κ_{r+1} (with new variables) which is a most-general resolvent of two (not necessarily distinct) clauses in φ.

$\hat{\rho}(3)$: conjoining a new clause κ_{r+1} (with new variables) which is in $\rho_c(\kappa_i|\mathcal{L})$ for some $1 \leq i \leq r$.

Example 4.13 Let φ be the conjunction of the following two clauses:

$$\begin{aligned} \kappa_1 &: \quad p(f(x_1)) \leftarrow q(x_1) \\ \kappa_2 &: \quad \leftarrow p(f(f(c))), q(g(y_1)) \end{aligned}$$

Then $\hat{\rho}(\varphi|\mathcal{L})$ includes the sentences κ_1, κ_2, $\kappa_1 \wedge \kappa_2 \wedge \kappa_3$, and $\kappa_1 \wedge \kappa_2 \wedge \kappa_4$, where

$$\begin{aligned} \kappa_3 &: \quad \leftarrow q(f(c)), q(g(y_2)) \\ \kappa_4 &: \quad \leftarrow p(f(f(c))), q(g(c)). \end{aligned}$$

κ_3 is obtained by unifying $p(f(x_1))$ in κ_1 with $p(f(f(c)))$ in κ_2 and resolving. Note that we rename the variables so that each clause has a disjoint set of variable names. κ_4 is obtained from κ_2 by substituting the most-general term c for the variable y_1. Δ

When the first-order language $\mathcal{L}$ is clear from context, we shall use the simpler notations $\rho_c(\kappa)$ and $\hat{\rho}(\varphi)$. And as with ρ_c, $\hat{\rho}$ is locally finite, provided $\hat{\rho}(2)$ and $\hat{\rho}(3)$ are suitably restricted from producing two sentences with the same set of clauses, modulo variants. And as with clauses, we consider such sentences as variants of one another and syntactically equivalent.

The main result of this section is the following.

Theorem 4.14 *$\hat{\rho}$ is a complete downward refinement for the class of clause-form sentences in $\mathcal{L}$.*

Proof: The proof has two parts: to show that, if $\varphi_2 \in \hat{\rho}(\varphi_1)$, then $h(\varphi_2) \subseteq h(\varphi_1)$; and to show that, if $h(\varphi_2) \subseteq h(\varphi_1)$, then φ_2 (or some variant) is in $\hat{\rho}^*(\varphi 1)$.

The first part is easy: if $\varphi_2 \in \hat{\rho}(\varphi_1)$, then $\vdash \varphi_1 \rightarrow \varphi_2$. For, if we obtain φ_2 by deleting a clause in φ_1, then clearly, $\vdash \varphi_1 \rightarrow \varphi_2$. Likewise, by the resolution principle [Robinson-65], $\vdash \varphi_1 \rightarrow \varphi_2$ when φ_2 is obtained from φ_1 by conjoining a resolvent. Finally, it is shown in [Shapiro-81] that $\vdash \kappa_1 \rightarrow \kappa_2$ if $\kappa_2 \in \rho_c(\kappa_1)$; hence if κ_1 is a clause of φ_1, then $\vdash \varphi_1 \rightarrow \varphi_1 \wedge \kappa_2$. Finally, if $\vdash \varphi_1 \rightarrow \varphi_2$ then $h(\varphi_2) \subseteq h(\varphi_1)$.

For the second part, assume that $h(\varphi_2) \subseteq h(\varphi_1)$, so that $\varphi_1 \rightarrow \varphi_2$ is provable. We claim that φ_2 (or some equivalent) belongs to $\hat{\rho}^*(\varphi_1)$. Assume that $\varphi_2 = \kappa_1 \wedge \cdots \wedge \kappa_r$ and that φ_2 is a logical consequence of φ_1. There is a resolution proof $\varphi_1 \vdash \varphi_2$, and hence $\varphi_1 \vdash \kappa_i$ for each $1 \leq i \leq r$. Our approach is to use the resolution proof of κ_i from φ_1 to construct a $\hat{\rho}$-refinement path from φ_1 to $\varphi_1 \wedge \kappa_i$. We can repeat this construction for each i and thereby show inductively that $\varphi_1 \wedge \varphi_2 \in \hat{\rho}^*(\varphi_1)$. By deleting in succession the clauses of φ_1 (via rule $\hat{\rho}(1)$) we can then refine $\varphi_1 \wedge \varphi_2$ to φ_2, demonstrating that $\varphi_2 \in \hat{\rho}^*(\varphi_1)$. So the proof reduces to the case that φ_2 is a clause κ.

We shall retain full generality if we assume that no clause of φ_2 is a tautology, and that no two clauses of φ_2 are variants of each other (for we can always replace φ_2 by an

equivalent sentence such that these conditions hold). Suppose that φ_1 is inconsistent; then, for any clause κ it is easy to derive $\varphi_1 \wedge \kappa$ from φ_1 by $\hat{\rho}$-refinements: derive $\varphi_1 \wedge \square$ by resolution steps, then successively add literals to $\square$ and refine them using ρ_c until $\square$ becomes κ.

We assume, therefore, that φ_1 is consistent. We also assume that every clause in φ_1 and φ_2 has its own set of variables not found in any other clause. Let φ_2 consist of a single clause κ. There is a resolution proof $\varphi_1 \vdash \kappa$; equivalently, there is a derivation of the empty clause $\square$ from the clauses of φ_1 and $\sim\kappa$. Suppose $\kappa = \forall \bar{x}(\alpha_1(\bar{x}), \ldots, \alpha_p(\bar{x}) \leftarrow \beta_1(\bar{x}), \ldots, \beta_n(\bar{x}))$, where $\bar{x}$ denotes the entire set of variables bound in κ. Then

$$\begin{aligned} \sim\kappa = & \sim\forall \bar{x}\,(\alpha_1(\bar{x}) \vee \cdots \vee \alpha_p(\bar{x}) \vee \sim\beta_1(\bar{x}) \vee \cdots \vee \sim\beta_n(\bar{x})) \\ = & \exists \bar{x}\,(\sim\alpha_1(\bar{x}) \wedge \cdots \wedge \sim\alpha_p(\bar{x}) \wedge \beta_1(\bar{x}) \wedge \cdots \wedge \beta_n(\bar{x})) \end{aligned}$$

This sentence is not in clause form. To eliminate the $\exists$-quantifiers, we extend the language $\mathcal{L}$, for purposes of this proof, by introducing new "Skolem constants" $\bar{s}$, one for each variable in $\bar{x}$. Thus

$$\sim\kappa[\bar{s}] = \sim\alpha_1(\bar{s}) \wedge \cdots \wedge \sim\alpha_p(\bar{s}) \wedge \beta_1(\bar{s}) \wedge \cdots \wedge \beta_n(\bar{s}).$$

Each literal of $\sim\kappa[\bar{s}]$ is variable-free and functions as an independent clause in the derivation of $\square$ from $\varphi_1 \wedge \sim\kappa$. Note that there is a bijection between the variables $\bar{x}$ occurring in $\kappa[\bar{x}]$ and the corresponding constants $\bar{s} \in \sim\kappa[\bar{s}]$. We shall utilize this correspondence later to restore uniquely the variables $\bar{x}$ to a formula with constants $\bar{s}$.

Example 4.15 To illustrate the steps of this proof, we shall develop an extended example. Let

$$\begin{aligned} \varphi_1 = & \begin{cases} p_1(f_1(x_1)) \leftarrow p_2(x_1) \wedge \\ p_2(f_2(x_2)) \wedge \\ p_3(f_3(x_3)) \leftarrow p_1(x_3), p_2(f_2(x_3)) \end{cases} \\ \kappa[x] = & \; p_3(f_3(f_1(f_1(x)))) \leftarrow p_2(f_1(x)), p_4(x, x) \end{aligned}$$

To construct the resolution proof $\varphi_1 \vdash \kappa$, we determine

$$\sim\kappa[x] = \exists x(\sim p_3(f_3(f_1(f_1(x)))) \wedge p_2(f_1(x)) \wedge p_4(x, x)).$$

To convert this to clause form, let s be a constant value for x (which exists, according to the $\exists x$). Then in the expanded language which contains the new constant s,

$$\sim\kappa[s] = \begin{cases} \sim p_3(f_3(f_1(f_1(s)))) \wedge \\ p_2(f_1(s)) \wedge \\ p_4(s, s) \end{cases}$$

The resolution proof tree is shown in Figure 8. Notice that the clause $p_4(s, s)$ of $\sim\kappa[s]$ does not appear in the tree. Δ

The proof proceeds by converting the resolution tree into a sequence of refinement steps. We shall label each node of the tree with an additional clause. To distinguish the two labels, let us call the label in the original resolution tree the r-clause, and denote the r-clause at node N by $r(N)$. The extra clause will be termed the x-clause and denoted $x(N)$. First, we give an algorithm to compute the x-clauses for the nodes of a resolution tree. Then we prove some properties of the x-clause labels that imply that these labels are clauses obtained from φ_1 by $\hat{\rho}$-refinements from the leaves. In particular, the x-clause at the root node will be a subclause of κ, and can be further refined to κ exactly by a sequence of ρ_c-refinements.

In the resolution proof tree, we say that a node N_1 is a *predecessor* of a non-leaf node N_2 if $N_1 \neq N_2$ and there is a path from a leaf to N_2 that includes N_1. N_1 is an *immediate predecessor* of N_2 if it is a predecessor and there is an edge joining N_1 and N_2. (This terminology is inverted from the usual tree terminology because the resolution tree, and our modifications to it, are constructed from the leaves toward the root.)

Algorithm for constructing x-clause labels:

The algorithm proceeds in two phases. First, clauses $x(N)[\bar{s}]$ are computed for each node N in the tree. Then the constants $\bar{s}$ are changed back to variables $\bar{x}$ using the bijection noted earlier, yielding $x(N)$.

Phase 1: The following rules describing the computation of the x-clauses proceed from the leaves to the root.

1. If N is a leaf node, then $x(N) = r(N)$.
   For example, in Figure 8 the node marked (4) is such a node, so $x(4)[s] = p_1(f_1(x_1)) \leftarrow p_2(x_1)$. (The fact that s does not appear explicitly means that $x(4)[s] = x(4)$ for this node.)
2. If N is a node such that *exactly one* of its immediate predecessors, N_1, is a leaf whose r-label is a literal from $\sim\kappa[\bar{s}]$, and its other immediate predecessor N_2 is not a clause from $\sim\kappa[\bar{s}]$, then $x(N)[s] = \theta[x(N_2)]$, where θ is the substitution used to resolve $r(N_1)$ and $r(N_2)$.
   For example, in Figure 8, the node marked (3) is such a node. The substitution used to obtain $r(3)$ is $\theta = \{x_3 \leftarrow f_1(f_1(s))\}$. Applying this substitution to $x(2)$ yields
   $$x(3)[s] = p_3(f_3(f_1(f_1(s)))) \leftarrow p_1(f_1(f_1(s))), p_2(f_2(f_1(f_1(s)))).$$
3. If N is a node *neither* of whose immediate predecessors N_1 or N_2 is a literal from $\sim\kappa[\bar{s}]$, then $x(N)[\bar{s}]$ is obtained by resolution from $x(N_1)[\bar{s}]$ and $x(N_2)[\bar{s}]$ via the same resolution operation used to obtain $r(N)$ from $r(N_1)$ and $r(N_2)$.

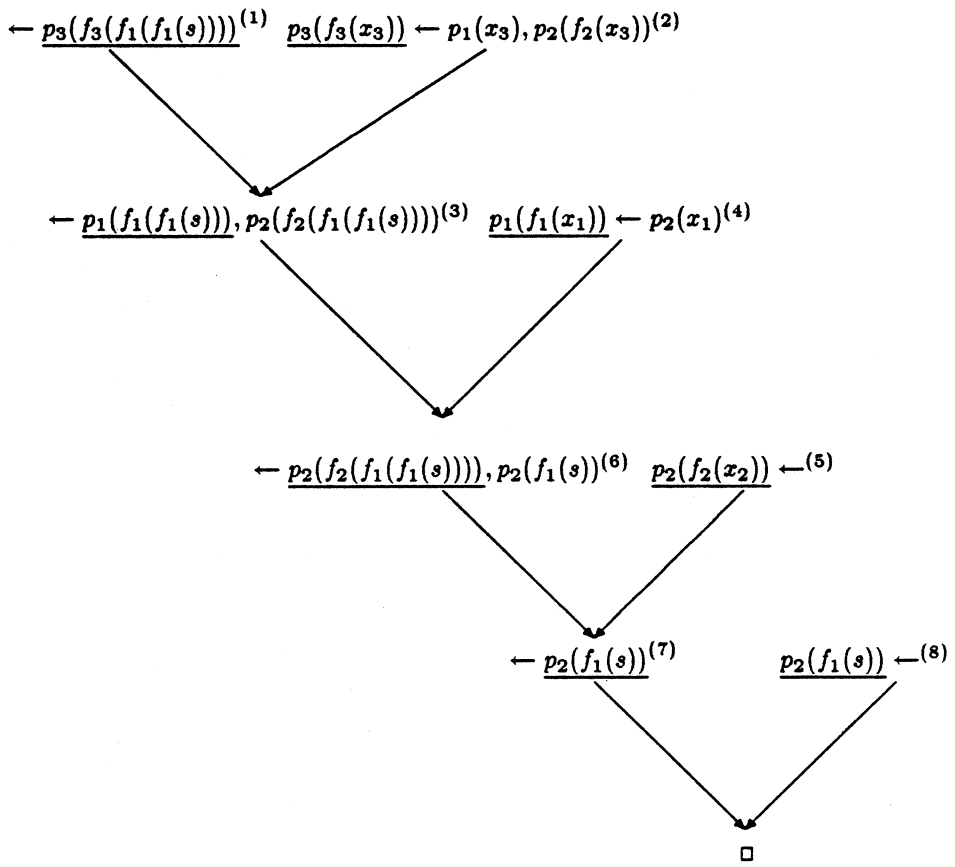

Figure 8: Resolution Proof Tree. Atoms resolved upon are underscored.

This is always possible because, as we subsequently prove, every literal in $r(N)$ occurs in $x(N)[\bar{s}]$.

For example, the node labeled (6) in Figure 8 is such a node. Literals $p_1(f_1(f_1(s)))$ and $p_1(f_1(x_1))$ were unified by the substitution $\{x_1 \leftarrow f_1(s)\}$ in the r-clauses. The same resolution operation applied to $x(3)[s]$ (above) and $x(4)[s]$ yields

$$x(6)[s] = p_3(f_3(f_1(f_1(s)))) \leftarrow p_2(f_2(f_1(f_1(s)))), p_2(f_1(s)).$$

4. If N is a node *both* of whose immediate predecessors N_1 and N_2 are leaves whose r-clause labels are literals from $\sim\kappa[\bar{s}]$, then $r(N) = \square$ (since $r(N_1)$ and $r(N_2)$ are ground literals). But this implies the κ is a tautology since $\square$ has been derived by resolution from $\sim\kappa[\bar{s}]$ alone. By assumption, φ_2 contains no tautological clauses, so *this case will not arise.*

Since every node of the resolution tree falls into one of the first three categories above, we can apply this procedure to obtain an extra label for every node in the tree.

Phase 2: To complete the derivation of x-clauses, apply the substitution $\theta_{x \leftarrow s} = \{\bar{x} \leftarrow \bar{s}\}$, substituting x's for the corresponding s's. See Figure 9 for the complete set of labels in the tree of Example 4.15. ◇

We now make the following three claims:

CLAIM 1: Every literal occurring in $r(N)$ also occurs in $x(N)[\bar{s}]$.

CLAIM 2: The x-clause at the root node is a non-empty subclause of κ.

CLAIM 3: Each rule for deriving $x(N)[\bar{s}]$ from the x-clauses of its immediate predecessors corresponds to a $\hat{\rho}$-refinement operation that results in conjoining the clause $x(N)$ to the original clause.

Given these results, we complete the proof as follows. Starting with the sentence φ_1, we can (using the third claim) add to the sentence each x-clause occurring on the internal nodes , using $\hat{\rho}$-refinement operations; we do so by adding first the x-clauses at nodes whose immediate predecessors are leaves, then their successors, and so on, until the x-clause for the root node $x(R)$ has been added. By the second claim, this clause is a non-empty subset of the literals of κ; the remaining literals of κ can be obtained by applying rule $\hat{\rho}(3)$, by Lemma 4.11. Finally, any extraneous clauses not in $\varphi_1 \wedge \kappa$ can be removed using rule $\hat{\rho}(1)$.

Example 4.15 (Continued) The x-clause at the root is κ except for the missing literal $\sim p_4(x, x)$. To obtain κ , we add the most general literal $\sim p_4(y_1, y_2)$ to the clause (rule $\rho_c(3)$). Applying rule $\rho_c(1)$, we add the same clause, but with the two variables unified: $\sim p_4(y, y)$. Finally we apply the same rule to unify the variable y with the variable x occurring elsewhere in the clause, and obtain the clause κ. △

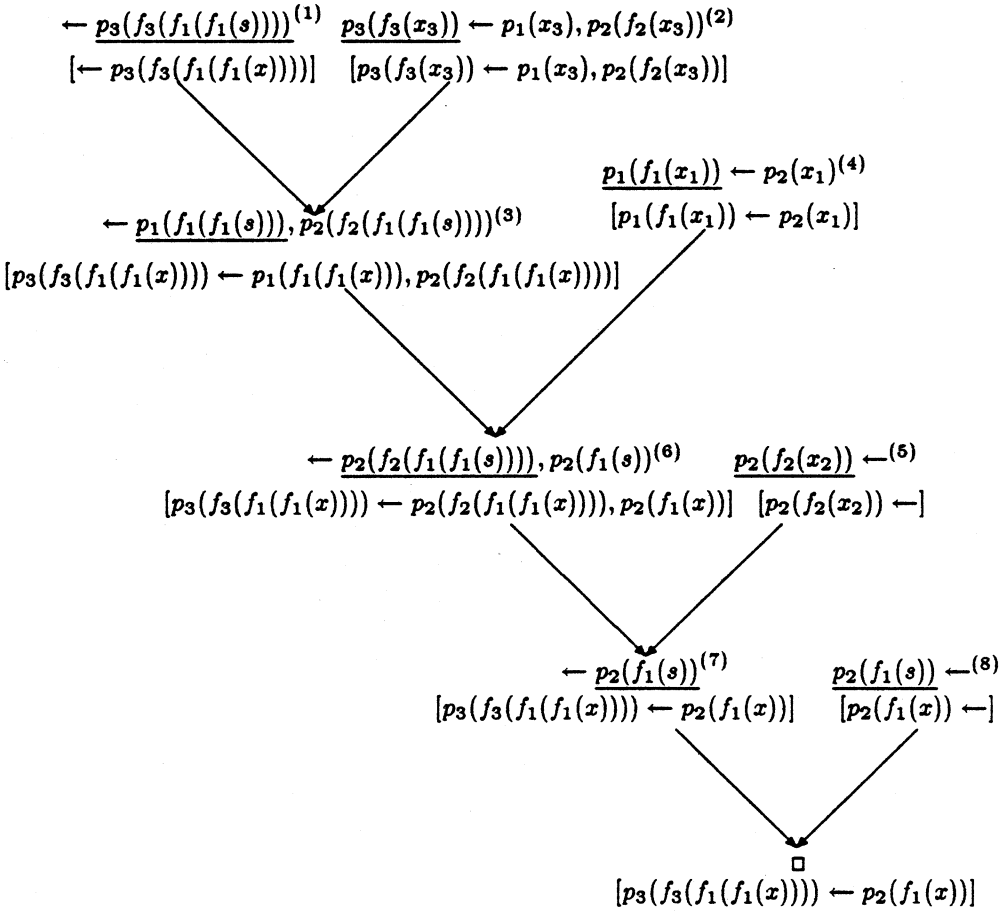

Figure 9: Resolution Proof Tree. The x-clauses are in square brackets.

Proof of Claim 1: by induction on the height of the node. For leave nodes N, $r(N)$ and $x(N)[\bar{s}]$ are identical, by Rule 1. Suppose that $x(N)[\bar{s}]$ is computed via the second rule, with predecessor node N_1 a ground literal from $\sim\kappa[\bar{s}]$ and the other node N_2 inductively satisfying the claim. The resolution step that results in $r(N)$ consists in unifying at least one literal in each of $r(N_1)$ and $r(N_2)$ via a substitution θ, combining the clauses $\theta[r(N_1)]$ and $\theta[r(N_2)]$, and removing the unified literals. But $r(N_2)$ is a single ground literal, so $r(N)$ is $\theta[r(N_2)]$ with at least one literal removed. On the other hand, $x(N)[\bar{s}] = \theta[x(N_2)[\bar{s}]]$ (no literals are removed), hence $r(N) \subseteq x(N)[\bar{s}]$.

Suppose that $x(N)[\bar{s}]$ is computed by Rule 3, and the two predecessors N_1 and N_2 of N satisfy the inductive hypothesis. The same literals removed from $r(N_1)$ and $r(N_2)$ by the resolution step are unified and removed from $x(N_1)[s]$ and $x(N_2)[s]$ and the same substitution is applied to the remaining literals. Hence the property is preserved at N. This concludes the proof of the first claim.

Proof of Claim 2: We observe first that only literals from κ will "survive" to reach the x-clause at the root node, since the other literals, originating from the clauses of φ_1, are resolved away by Rule 3. Also note that, once a literal from κ is "added to " (more precisely, unified with a literal of) the x-clause of a node, it is present in the x-clauses of all successor nodes in the tree, since no rules entail removing literals except those occurring in r-clauses. Thus if $x(R)[\bar{s}]$ at the root R were empty, then the resolution tree would represent a derivation of $\square$ using only clauses from φ_1. But then φ_1 would be inconsistent, contrary to hypothesis. This proves the second claim.

Proof of Claim 3: Rule 2 results in the substitution of a term for variables, since one of the two predecessors is a single literal from $\sim\kappa[s]$. Clearly, the same result can be obtained using one or more applications of $\rho_c(2)$ (substituting a most-general term for variables). For example, the substitution $\{x \leftarrow f(g(y))\}$ is effected by the ρ_c-refinements: $x \Rightarrow f(z) \Rightarrow f(g(y))$.

Rule 3 is a resolution operation, which corresponds directly to $\hat{\rho}(2)$.

This completes the proof of the claim, and the theorem. □

Summarizing, we have defined an order homomorphism from clause-form sentences to classes of models, or equivalently, to sets of ground implicands. The class of ground clauses can be used to provide a complete set of examples. And we have obtained, in $\hat{\rho}$, a locally finite downward refinement for clause-form sentences. $\hat{\rho}^*(\varphi)$ includes all clause-form sentences (modulo variants) less general than φ, with respect to the ordering $\varphi \succeq \psi$ iff $\varphi \rightarrow \psi$. Consequently $\hat{\rho}$ is complete for any most-general sentence, including the sentence $\{\square\}$ consisting of only the empty clause. *Consequently we can use Algorithm 3.40 to infer classes of first-order models.*

4.5 Meta-Refinements

Even though we have not presented $\hat{\rho}$ in axiomatic form, we could have, since it is clearly computable. The (meta-)language $\hat{\mathcal{L}}$ in which the axioms for $\hat{\rho}$ would be expressed is different from $\mathcal{L}$. $\hat{\mathcal{L}}$ needs predicate symbols denoting such properties as "x is a clause-form sentence in $\mathcal{L}$", "sentence $x \succeq$ sentence y", "w is the clause x with term y substituted for variable z", and so on. The logical and non-logical symbols of the language $\mathcal{L}$ would typically be represented by function symbols in the language $\hat{\mathcal{L}}$. For example, if $p(\cdot)$ is a one-place predicate symbol in $\mathcal{L}$ and "*is-a-literal*" and "*is-an-atom*" are predicates in $\hat{\mathcal{L}}$, then $\hat{\mathcal{L}}$ might include the axioms:

$$\textit{is-a-literal}(p(x)) \leftarrow \textit{is-an-atom}(p(x)) \tag{1}$$

$$\textit{is-a-literal}(\textit{not}(p(x))) \leftarrow \textit{is-an-atom}(p(x)) \tag{2}$$

We are considering this "implementation detail" in order to make the following observation: ***since $\hat{\rho}$ is a sentence in a first-order language, it can in turn be refined.*** In fact, every sub-refinement of $\hat{\rho}$ is obtainable by refining $\hat{\rho}$. A refinement which operates on axioms to produce other refinements can be termed a *meta-refinement.*

To see what we mean, we have to be more careful about the languages in which the various sentences are expressed. $\mathcal{L}$ is the language in which the target class of models is to be denoted. $\hat{\rho}(\cdot, \mathcal{L})$ is a downward refinement for clause-form sentences in $\mathcal{L}$; the language $\hat{\mathcal{L}}$ in which $\hat{\rho}(\cdot, \mathcal{L})$ is axiomatized might well include a two-place predicate "*refines*" such that $\hat{\rho} \vdash \textit{refines}(\varphi_2, \varphi_1)$ (in $\hat{\mathcal{L}}$) iff $\varphi_2 \in \hat{\rho}(\varphi_1, \mathcal{L})$. We may, of course, assume that the axioms for $\hat{\rho}$ are in clause form. $\hat{\rho}$ is a complete refinement (up to variants), so the transitive closure of the *refines* relation is the $\succeq$ relation.

Let $\hat{\alpha}(\cdot, \mathcal{L})$ be a partial downward refinement with a clause-form axiomatization in $\hat{\mathcal{L}}$ like $\hat{\rho}$. If $\hat{\alpha} \vdash \textit{refines}(\varphi_2, \varphi_1)$, then $\hat{\rho} \vdash \textit{refines}(\varphi_2, \varphi_1)$ (since $\hat{\rho}$ is complete). Again, all these proofs are in the language $\hat{\mathcal{L}}$.

Now consider $\hat{\hat{\mathcal{L}}}$ to be the language with the same relationship to $\hat{\mathcal{L}}$ as $\hat{\mathcal{L}}$ has to $\mathcal{L}$. $\hat{\hat{\mathcal{L}}}$ has the same predicates as $\hat{\mathcal{L}}$ but the function symbols specifying the allowable non-logical symbols will reflect $\hat{\mathcal{L}}$ instead of $\mathcal{L}$. This is the language in which the meta-refinement $\hat{\hat{\rho}}(\cdot, \hat{\mathcal{L}})$ is axiomatized. If $\hat{\hat{\rho}} \vdash \textit{refines}(\hat{\alpha}_2, \hat{\alpha}_1)$ in $\hat{\hat{\mathcal{L}}}$, then as a first-order sentence (in $\hat{\mathcal{L}}$), $\hat{\alpha}_1 \succeq \hat{\alpha}_2$, and $\hat{\alpha}_1$ logically implies $\hat{\alpha}_2$.

Conversely, $\hat{\hat{\rho}}$ is complete (for $\hat{\mathcal{L}}$-sentences), so if $\hat{\alpha}_1 \succeq \hat{\alpha}_2$, then $\hat{\alpha}_2 \in \hat{\hat{\rho}}^*(\hat{\alpha}_1, \hat{\mathcal{L}})$ – that is, there is a chain of sentences in $\hat{\mathcal{L}}$, starting with $\hat{\alpha}_1$ and ending with $\hat{\alpha}_2$ (or variant thereof), such that each sentence in the chain is a $\hat{\rho}$-refinement of the one before it. Note, however, that not every clause-form sentence in $\hat{\mathcal{L}}$ is an $\mathcal{L}$-refinement; for example, $\{\square\}$ is not, because it proves $\textit{refines}(x, y)$ for any x and y. But the set $\hat{\hat{\rho}}^*(\hat{\rho}, \hat{\mathcal{L}})$ includes (modulo variants) all partial $\mathcal{L}$-refinements.

In summary, we have shown the

Corollary 4.16 *Let $\mathcal{L}$ be a first-order language, and $\hat{\rho}$ an axiomatization (in a language $\hat{\mathcal{L}}$) for the complete downward $\mathcal{L}$-refinement used in Theorem 4.14. Let $\hat{\hat{\rho}}$ be an axiomatization (in language $\hat{\hat{\mathcal{L}}}$) for the complete $\hat{\mathcal{L}}$-refinement. Then every downward $\mathcal{L}$-refinement, up to variants, is included in the set $\hat{\hat{\rho}}^*(\hat{\rho}, \hat{\mathcal{L}})$ of refinements of $\hat{\rho}$.*

Here is a simpler way to obtain sub-refinements in such a way that we can visualize more easily what is happening, without appealing to the details of the axiomatic representation. A refinement ρ of $\mathcal{E}$ is a recursively enumerable binary relation on $\mathcal{E}$ which preserves the ordering: $\alpha \rho \beta$ only if $h(\alpha) \geq h(\beta)$. Another r.e. binary relation ρ' is a *subrefinement* of ρ iff it is also order preserving and $\rho' \subseteq \rho^*$.

We observe that ρ' is a sub-refinement of ρ iff $\rho' = r \cap \rho^*$, where r is any r.e. binary relation, representing a restriction on ρ. To see this, note that any such ρ' is r.e. (since r.e. sets are closed under $\cap$ and $*$); that any sub-refinement ρ' is of this form (let $r = \rho'$); and that ρ' is order preserving (since, if $\alpha \rho' \beta$ then $\alpha \rho^* \beta$, hence $\alpha \succeq \beta$).

The meta-refinement generates sub-refinements in an order-preserving way, so that if ρ_2' is obtained by meta-refinement from ρ_1', then ρ_2' is no more general a refinement than ρ_1' (i.e., it will not be a larger subset of ρ^*). But for sub-refinements obtained as $r \cap \rho^*$, this will be the case only if restrictions are produced in an order preserving manner.

With the advantage of generating less general refinements, the meta-refinement has the disadvantage of deriving relations which are not refinements. One way to avoid this is to maintain the form $r \cap \rho^*$ for sub-refinements ρ' by axiomatizing them in the following way:

$$\mathit{subrefines}(\varphi, \psi) \leftarrow \varphi \, \rho^* \, \psi, \ \mathit{restricts}(\varphi, \psi)$$

where ρ^* is axiomatized by the clauses:

$$\varphi \, \rho^* \, \varphi$$

$$\varphi \, \rho^* \, \psi \leftarrow \varphi \, \rho \, \varphi', \ \varphi' \, \rho^* \, \psi$$

and initially "*restricts*" is the relation $\mathcal{E} \times \mathcal{E}$:

$$\mathit{restricts}(\varphi, \psi) \leftarrow \mathbf{true}.$$

We then "meta-refine" only the axiomatization of *restricts*, leaving the axiomatization of ρ^* and of *subrefines* unchanged. This approach does not preserve the locally finite property of the refinement ρ, whereas the pure meta-refinement of ρ does.

4.6 Bottom-Up Inference

Unlike the propositional case, where the upward and downward directions are duals, the upward direction in inferring classes of first-order models is less satisfying than the downward. The asymmetry arises during the construction of the terms: the sequence x,

$f(x)$, $f(f(x))$, ...is unbounded and increasingly restrictive for purposes of unification. Whereas the downward refinement replaces a variable x by a more restrictive term, the upward refinement goes the other way, and the absence of a most restrictive term makes the upward direction inherently different.

Definition 4.17 *Let κ be a clause in the language $\mathcal{L}$ as in Definition 4.9. $\gamma_c(\kappa, \mathcal{L})$ is the set of clauses in $\mathcal{L}$ derived from κ by exactly one of the following operations:*

$\gamma_c(1)$: Replace some, *but not all*, occurrences of a variable x by a fresh variable y. (This operation is called *anti-unification of variables.*)

$\gamma_c(2)$: Let t be a most general term, occurring at least once, such that no variable of t occurs in the sentence except in t. Replace *all* occurrences of t by a fresh variable y. (This operation is called *anti-substitution* of variables for terms.)

$\gamma_c(3)$: Delete a literal.

$\gamma_c(4)$: Disjoin a copy of a literal α which already occurs in the clause.

Example 4.18 Let $\kappa = p_1(x_1, f(x_2)) \leftarrow p_2(f(x_1))$. Then $\gamma_c(\kappa, \mathcal{L})$ includes the clauses:

$$p_1(x_1, f(x_2)) \leftarrow p_2(f(x_3))$$

$$p_1(x_1, x_4) \leftarrow p_2(f(x_1))$$

$$p_1(x_1, f(x_2))$$

$$p_1(x_1, f(x_2)), p_1(x_1, f(x_2)) \leftarrow p_2(f(x_1))$$

obtained from κ using $\gamma_c(1)$ through $\gamma_c(4)$, respectively. △

Lemma 4.19 *If $\kappa' \in \rho_c{}^*(\kappa)$, then $\kappa \in \gamma_c{}^*(\kappa')$.*

Proof: By inspection, $\gamma_c(i)$ is $\rho_c{}^{-1}(i)$ for $1 \leq i \leq 3$. □

Observe, in passing, that γ_c is locally finite and that $\gamma_c(4)$ is the only rule which always results in an equivalent clause. In general γ_c is not complete for any finite set of clauses since no rule creates new literals or instances of literals already present.

Definition 4.20 *Let $\varphi = \kappa_1 \wedge \cdots \wedge \kappa_r$ be a clause-form sentence over $\mathcal{L}$, with no variable occurring in more than one clause. The set $\hat{\gamma}(\varphi|\mathcal{L})$ is the set of clause-form sentences derived from φ by exactly one of the following operations:*

$\hat{\gamma}(1)$: Conjoin a new ground clause κ_{r+1}.

$\hat{\gamma}(2)$: Replace a clause κ_i by a pair of clauses $\kappa_i \vee \alpha$ and $\kappa_i \vee \sim\alpha$, where α is any atom. (This operation is called *anti-resolution.*)

$\hat{\gamma}(3)$: Replace a clause κ_i by a clause $\kappa_i' \in \gamma_c(\kappa_i|\mathcal{L})$.

Note that $\hat{\gamma}$ is not locally finite, since there are countably many clauses that could in general be added in steps $\hat{\gamma}(1)$ and (2).

Theorem 4.21 *$\hat{\gamma}$ is an upward refinement for the class of clause-form sentences in $\mathcal{L}$, complete for the empty sentence $\emptyset$.*

Proof: This is not a big result, since $\hat{\gamma}$ is, in essence, the inverse of $\hat{\rho}$. We need only check that, if $\varphi \in \hat{\rho}(\varphi')$, then $\varphi' \in \hat{\gamma}^*(\varphi)$ for every sentence φ. The result then follows, since the empty sentence $\emptyset$ is in $\hat{\rho}^*(\varphi)$ for every sentence φ.

CASE: φ is derived from φ' by rule $\hat{\rho}(1)$, that is, by deleting a clause. Then φ' is derived from φ by conjoining a ground-instance of the deleted clause and applying $\hat{\gamma}(3)$ as many times as necessary. Since all ground instances of a clause are derivable by ρ_c-refinements of the clause, Lemma 4.19 says we can reverse the process.

CASE: φ is derived from φ' by rule $\hat{\rho}(2)$ (resolution). The resolution operation has two parts: applying a substitution, and combining two clauses into one while eliminating complementary literals. Anti-resolution is almost the inverse of the second part, differeing only in that the unifying substitution can unify more than one literal in each clause. But when more than one literal is eliminated from one clause, the literals are always copies of each other. Thus anti-resolution and $\gamma_c(4)$ together can reverse the second part of the resolution step. The substitution part can (appealing again to Lemma 4.19) be reversed by γ_c steps.

CASE: φ is derived from φ' by rule $\hat{\rho}(3)$. Then Lemma 4.19 applies once more to derive φ' from φ using γ_c. □

A corollary of this theorem is that the bottom-up equivalent of Algorithm 3.40 can be applied to the inference of model classes. Since $\hat{\gamma}$ is not locally finite, a dovetailed version must be used.

Is there a locally-finite upward refinement for clause-form sentences? Yes, but it is somewhat artificial. Following [Reynolds-70] we can define a size-complexity measure on atomic formulas as follows: the size of an atom A is the number of symbol occurrences in A – including the predicate, variable, and function symbols but excluding punctuation and parentheses – minus the number of distinct variables occurring in A. Thus $p(f(x), y)$ has size 2, while $p(f(x), x)$ has size 3. We extend this to literals by defining the size of the literal to be the size of its atom, regardless of sign. The principal properties of this measure are that a most-general literal has size 1, and that instantiating a literal by

unifying two variables or replacing a singly-occurring variable by a most-general term increases the size by 1.

Let φ be a sentence. Define the *size of* φ to be the size of the largest literal in φ, or zero if φ is the empty sentence. For any fixed size $S > 0$, there is a finite number of ground clauses of size S or less. If we modify rule $\hat{\gamma}(2)$ to conjoin a new ground clause to φ only if its size is at most one more than the size of φ, then we impose a finite limit on any single application of the rule. Similarly we can modify rule $\hat{\gamma}(3)$ to introduce only literals of size at most one greater than that of φ.

4.7 Normal Forms and Monotonic Operations

General Properties. The existence of conjunctive and disjunctive normal forms is a property of the algebra of logical expressions which leads to many important simplifications (among them, resolution). In this section we give an abstract inference algorithm which takes advantage of normal form representations for expressions. We then consider the propositional and first order implementations of this algorithm, leading to results similar in flavor to those of Shapiro's Model Inference System ([Shapiro-82]).

Definition 4.22 *Let $\circ$ be a binary operation, part of the algebra of the syntactic domain $\mathcal{E}$ of expressions. We say that $\circ$ is* h-monotonic upward *if, for any expressions e_1 and e_2, $h(e_1 \circ e_2) \geq h(e_i)$ (for $i = 1, 2$). Similarly, we say that $\circ$ is* h-monotonic downward *if, for any expressions e_1 and e_2, $h(e_1 \circ e_2) \leq h(e_i)$ (for $i = 1, 2$).*

Example 4.23 Let $\mathcal{E}$ be the domain of propositional logic, and h the map from propositional expressions to the assignments which satisfy them. Then $\vee$ is h-monotonic upward and $\wedge$ is h-monotonic downward. If the implication operation were part of the algebra, it would not be monotonic.

Likewise, $\vee$ and $\wedge$ are, respectively, h-monotonic upward and downward in the domain of first-order sentences where $h(e)$ is the class of models satisfying e. We have chosen, however, the dual semantics where $h(e)$ is the class of ground implicands of e; in this case, $\vee$ is downward and $\wedge$ non-decreasing. Δ

Definition 4.24 *Let $\circ$ be a binary, associative, and commutative operation in $\mathcal{E}$. $\mathcal{E}$ is said to have a* $\circ$-normal form property *if, for every semantic object $d \in \mathcal{D}$, there exists an expression e in $h^{-1}(d)$ in the form $e = e_1 \circ e_2 \circ \cdots \circ e_n$, where $e_i \in \mathcal{E}$ and the operation $\circ$ does not occur in any e_i, for $1 \leq i \leq n$. An expression in which $\circ$ does not occur is called a* component *expression.*

The algorithm to follow uses a $\oplus$-normal form property, where $\oplus$ is h-monotonic upward, to infer an expression in $\oplus$-normal form. It assumes an additional oracle, *DIAGNOSE*, which takes an expression in $\oplus$-normal form and, if it is too general, identifies a component e_i of e which is too general for the target object d_0. That there exists at least one such expression is a consequence of the following.

Definition 4.25 *An expression e is said to be* too general *with respect to a target object d_0 if, for any expression e_0 denoting d_0, $h(e_0 \oplus e) > d_0$. The expression is said to be* correct *with respect to d_0 if $h(e_0 \oplus e) = d_0$.*

Lemma 4.26 *Let $H = e_1 \oplus \cdots \oplus e_n$ be a $\oplus$-normal form expression and $-x$ a negative example of the target object d_0. If $h(H) \geq h(x)$, then there exists a component e_i of H that is too general with respect to d_0.*

Proof: Let e_0 be a $\oplus$-normal form expression for d_0. If no component of H is too general, then $e_0 \oplus e_1$ is a normal form expression such that either $h(e_0 \oplus e_1) \leq h(e_0)$ or $(e_0 \oplus e_1)$ and e_0 are incomparable. But monotonicity of $\oplus$ requires that $h(e_0 \oplus e_1) \geq h(e_0)$; consequently, they cannot be incomparable, and it must follow that $e_0 \oplus e_1$ is equivalent to e_0, modulo h. The same argument holds for e_2, using $e_0 \oplus e_1$ as an expression for d_0, leading to the conclusion that $e_0 \oplus e_1 \oplus e_2$ is equivalent to d_0. And so on, until we have shown that $e_0 \oplus H \approx e_0$. But now we have $e_0 \approx H \oplus e_0 \geq H \geq -x$, which is clearly contradictory. □

The problem of identifying a component that is too general evidently depends upon the nature of the semantic function h, for we may have the situation where $h(e_1 \oplus e_2) \geq h(x)$, e_1 is correct, e_2 is too general, and yet $h(e_2) \not\geq h(x)$.

Example 4.27 If we are inferring a set of ground atoms, the target might be the set of ground instances of $p(x)$. The hypothesis $H = p(f(a)) \wedge (q(x) \leftarrow p(x))$ denotes the set $\{p(f(a)), q(f(a))\}$, which covers the negative example $q(f(a))$. The first clause of H is correct, while the second is too general. But individually the clauses denote the sets $\{p(f(a))\}$ and $\emptyset$, respectively. So neither is, by itself, too general. △

For domains like that of this example, Shapiro gives a procedure that, with the aid of the oracle *ASK*, identifies a clause which is too general. The net effect is that, for this particular domain, *DIAGNOSE* is reducible to *ASK*. But this does not appear to hold in general.

Suppose H is not general enough to cover a positive example $+x$. We need to add ($\oplus$) a component or components to H. Let L be an enumeration of all component expressions. One approach is to search L for a component e such that $h(H \oplus e) \geq h(x)$. But this may not work. For example, x may be covered only by expressions with two or more components, and none of those components are currently among those in H. Or, even if there is a single component which covers e, that component might be too general for the target, and hence could have been eliminated earlier by a negative example. So, in the general setting, with no information about the properties of the relation $\geq$, we can reject a component expression only if it is too general, but we cannot pass over a component expression simply because it currently does not cover any positive examples.

Let us now give an algorithm for inferring normal form expressions. As in the previous sections, many different algorithms are possible, depending on the properties of the refinement, the direction of inference, and the various dualities. We shall be content with only one example after which others may be easily patterned.

Definition 4.28 *Suppose $\mathcal{E}$ enjoys a $\circ$-normal form property. ρ is said to be a* downward refinement for $\circ$-components *if, for any two components e_1 and e_2, $e_1 \rho e_2$ implies that $h(e_1) \geq h(e_2)$. In addition, we say that ρ is* complete for M *if: (i) M is a finite set of maximal components, $\{\bar{e}_1, \ldots, \bar{e}_s\}$; and (ii) $h(\bigcup_{i=1}^{s} \rho^*(\bar{e}_i))$ includes $h(e)$ for every component e in $\mathcal{E}$. As before, $\succeq$ on components is defined to be ρ^*.*

Algorithm 4.29 for inferring expressions in normal form is given in Figure 10. It assumes, among other things, that ρ is locally finite and complete for a finite set M of maximal expressions. The hypothesis H is maintained as a set of components which are implicitly joined by $\oplus$-operations. Initially H contains only the set M. A new example starts up a **while**-loop that keeps modifying H as long as it disagrees with some example. If H is too general (implies a negative example), an oracle is invoked to pinpoint one of the components of H having excess generality. That component is evicted, and its refinements are added to a queue. If H is too specific, we make it more general by adding components to it. From the preceding discussion, we cannot in general be very discriminating about what we add; in fact, the algorithm just grabs components as they occur on the queue until its requirement to cover a particular positive example has been satisfied. It appears there might be some risk that the queue could be exhausted before the example has been covered, but in the proof we show that this cannot happen. Eventually this process of grabbing new components, discarding and refining old ones ends with the hypothesis correct for all the examples, whereupon a new example is summoned, and the process begins anew.

Theorem 4.30 *Algorithm 4.29 identifies d_0 in the limit.*

Proof: Assume that the algorithm eventually reads in every example and converges to some hypothesis. Then it converges to a correct hypothesis, since the **while**-loop ensures that every positive example and no negative examples are covered. Thus we need to show that every example is eventually read – i.e., , that the **while**-loop terminates for every finite set of examples – and that the algorithm will not keep modifying the hypothesis forever.

Let d_0 be the target object, and let $e_0 = e_{01} \oplus \cdots \oplus e_{0k}$ be a normal-form expression for d_0. We argue first that, without loss of generality, we can assume that for each e_{0i} there is a refinement path from an element of M to e_{0i} along which e_{0i} is the only component that is *not* too general. Consider any component e_{0i} of e_0. Since ρ is complete for M, e_{0i} (or some equivalent) is derivable from some component $\bar{e}$ in M by ρ-refinement. Take any such sequence $\bar{e} \stackrel{\rho}{\Rightarrow} f_1 \stackrel{\rho}{\Rightarrow} \cdots \stackrel{\rho}{\Rightarrow} f_n \stackrel{\rho}{\Rightarrow} e_{0i}$ and determine the first component c_i in the chain which is *not* too general for e_0. This is clearly possible since e_{0i} is not too general. If c_i is not e_{0i} then we can replace e_{0i} by c_i in the expression $e_0 = e_{01} \oplus \cdots \oplus e_{0i} \oplus \oplus \cdots \oplus e_{0k}$ and obtain $e_0' = e_{01} \oplus \cdots \oplus c_i \cdots \oplus e_{0k}$ as an equivalent expression for d_0. And by repeating this construction for each of the components of e_0, we obtain an expression $c_1 \oplus \cdots \oplus c_k$ for d_0 with the property that each component has a refinement path from a maximal element and that c_i is the first component on the path not too general for d_0. Also, replacing c_i by any other component along the path yields an expression too general for d_0.

Now let $e_0 = c_1 \oplus \cdots \oplus c_k$ be such an expression. Make the following observations:

1. Once a component c_i of e_0 is added to H, it will never be refined (only components that are too general are refined).

Algorithm 4.29 *Downward Inference of Normal Form Expressions*

Input:
- A r.e. set of expressions $\mathcal{E}$ such that $\oplus$ is h-monotonic upward and a $\oplus$-normal form theorem applies.
- A finite set $M = \{\bar{e}_1, \ldots, \bar{e}_s\}$ of maximal $\oplus$-components.
- A locally finite, downward component refinement ρ complete for M.
- An oracle EX for a sufficient set of examples.
- An oracle $ASK(e_1 \geq e_2?)$, where e_1 is a $\oplus$-normal form expression and e_2 a component expression.
- An oracle $DIAGNOSE(e_1, e_2)$ that finds a component in the $\oplus$-normal form expression e_1 that is too general. (Applies when e_2 is a negative example and $h(e_1) \geq h(e_2)$.)

Output: A sequence H_1, H_2, ... of $\oplus$-normal form expressions such that H_i is correct for the first i examples.

Procedure:

$H \leftarrow$ *emptyset*(). (The hypothesis is represented as a set of components implicitly connected by $\oplus$.)

$Q \leftarrow (\bar{e}_1, \ldots, \bar{e}_s)$.

examples $\leftarrow$ *emptyset*().

do forever:

  Get the next example from $EX()$ and add it to the set *examples*.

  while H disagrees with some example:

    if $ASK(H \geq x?) = 1$ for some negative example $-x$:

      Identify a component $e = DIAGNOSE(H, x)$ of H that is too general.

      Remove e from H.

      Add $\rho(e)$ to (the tail of) Q.

    if $ASK(H \geq x?) = 0$ for some positive example $+x$

      Remove a component e from the front of Q. (The algorithm fails if Q is empty).

      $H \leftarrow H \cup \{e\}$.

  Output H as the $\oplus$ of each component in H.

Figure 10: Downward Inference of Normal Form Expressions

[Pitt-84] Pitt, L.
A characterization of probabilistic inference.
In *Proc. 25th Annual Symposium on Foundations of Computer Science*, pp. 484-494, IEEE, 1984.

[Red'ko-64] Red'ko, V. N.
Relational definitions for the algebra of regular events.
Ukrain. Math. Zh. 16: 120–126, 1964. (In Russian)

[Reynolds-70] Reynolds, J. C.
Transformational systems and the algebraic structure of atomic formula.
In *Machine Intelligence 5*, B. Meltzer and D. Mitchie, Eds.
Edinburgh University Press, Edinburgh, 1970.

[Robinson-65] Robinson, J. A.
A machine-oriented logic based on the resolution principle.
J. ACM 12: 23–41, 1965.

[Salomaa-66] Salomaa, A.
Two complete axiom systems for the algebra of regular events.
J.ACM 13: 158–169, 1966.

[Shapiro-81] Shapiro, E. Y.
Inductive inference of theories from facts.
Tech. Rep. 192, Department of Computer Science, Yale University, New Haven, Ct. 1981

[Shapiro-82] Shapiro, E. Y.
Algorithmic program debugging.
Ph. D. dissertation, Computer Science Department, Yale University, New Haven, Ct. 1982.
Published by M.I.T. Press, 1983.

[Summers-77] Summers, P. D.
A methodology for LISP program construction from examples.
J.ACM 24: 161–175, January, 1977.

[Valiant-84] Valiant, L. G.
A theory of the learnable.
C. ACM 27: 1134–1142, November, 1984.

Rev: May 29, 1986