THE UNIVERSITY OF CALGARY

Inductive Theorem Generation

by

Brent J. Krawchuk

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

DECEMBER, 1991

THE UNIVERSITY OF CALGARY

Faculty of Graduate Studies

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Inductive Theorem Generation," submitted by Brent J. Krawchuk in partial fulfillment of the requirements for the degree of Master of Science.

Ian H. Witten
Supervisor
Department of Computer Science

Brian R. Gaines,
Department of Computer Science

Richard Cleve,
Department of Computer Science

John Heintz,
Department of Philosophy

Date December 2, 1991

Abstract

Machine learning and automated theorem proving are both important topics in computer science that have attracted much attention. With few exceptions, they have been investigated independently. This thesis explores their relationship by treating machine learning as a process of inductive theorem generation.

Machine learning is often said to be a process of induction, and some automated theorem proving systems are able to prove inductive theorems. However, we argue that the "concepts" or rules created by machine learning systems belong to broader classes of induction that we call "ampliative," "prudent," and "ignorative." While these classes are all inductivelike, they differ in how they model noise and unknown data. They are formalized in this thesis by developing a model theory of a four-valued non-monotonic equational logic.

The process of theorem generation is treated as a form of reverse theorem proving, in which theorem-proving operators are inverted to yield generation operators. Within an equational logic and term rewriting setting, antiunification is defined to be the inverse of unification, expansion the inverse of rewriting, and antinarrowing the inverse of narrowing. Specialization is viewed as an inverse to antiunification.

These ideas are implemented in the **Balog** system, a computer program that generates theorems and can emulate many well-known machine learning techniques within the framework of automated theorem proving.

Acknowledgements

The people of the Department of Computer Science have made my stay at the U of C immensely enjoyable. There are too many people that I'd like to thank so I'll just list the first ones that pop into my head:

Rosanna Heise, Debbie Leishman, David Maulsby, Dan Freedman, Maurice Sharp, Thong Phan, others in my office and the KSI, and all of the grad students I knew (for making the Computer Science environment far from boring); Craig Jackson (for lending an ear to my incoherent babbling, especially when I needed it the most); Lorraine Storey, Camille Sinanan, and the rest of the office staff (for their great help over the years); Paliath Narendran, Graham Birtwistle and John Cleary (for getting me onto the straight and narrow path of logic — maybe I should curse them); Bruce MacDonald (for being a cool resource for machine learning stuff); Brian Gaines (for his enthusiasm and for being a role model for my habit of devouring books); Dave Jevans and David Hankinson (for sev, glagliness and uncompromising excellence); Darse Billings (a friend who helped me escape from reality); Darrell Conklin (a friend who helped me escape from both normality and stupidity); Ian Witten (for being "The Filter" and for being an excellent supervisor); and several people outside the department (who made my life interesting during this period). I especially would like to thank Mom and Dad who have put up with far too much from me while I have been working (and not working!) on this thesis.

I also wish to thank the person who I could not have accomplished this dissertation without:

me.

Table of Contents

List of Figures

Chapter 1

Introduction

Machine learning is an important topic in Computer Science that has attracted a great deal of attention. This is not surprising since two research areas that have recently experienced tremendous growth are expert systems and robotics, both of which would benefit enormously from learning capabilities. The machine learning literature details many disparate methods including systems for intelligent question asking (Krawchuk & Witten, 1988; Sammut & Banerji, 1986), classification and concept learning (Quinlan, 1986; Quinlan, 1987; Gaines, 1991; Mitchell, 1982; Michalski, 1983), function induction (Phan, 1989), procedure and sequence learning (Heise, 1989; Maulsby & Witten, 1989; Dietterich & Michalski, 1986), and logic program synthesis (Muggleton & Buntine, 1988; Shapiro, 1983). This plethora of approaches indicates a need for descriptive and analytical theoretical frameworks, which should not only provide explanations of experimental research in the field but also should stimulate new ways of looking at problems. This thesis develops one such theory.

Traditional learning theories have focussed on learnability and other primarily *analytic* issues (Angluin & Smith, 1983). Unfortunately, they do not provide a simple *descriptive* language suitable for straightforward comparison of machine learning approaches. This thesis develops a logic-based theory used to explain some parts of machine learning and for generating new machine learning algorithms. Its main insight is that computer science logic, both semantics and proof theory, suggests a rich, well-understood basis for machine learning. In particular, it shows first, how a non-monotonic, multi-valued logical semantics can model induction of various types; and second, that methods of induction can be developed from theorem proving techniques.

The theory should help to provide a *lingua franca* for communication between machine learning researchers. For example, it has long been suspected that a common unit in machine learning systems is the process of universal generalization (see eg. Michalski, 1983; Popplestone, 1970). This process has often been alluded to by vague phrases such as *"replacing constants by variables."* Also, machine learning researchers will often say that their systems

are doing *"induction,"* a term which has been grossly abused as a catch-all to refer to very different things. These phrases undoubtedly need clarification — and this will follow directly from the logical framework developed in this thesis.

It is taken for granted here that the description languages used by machine learning systems can be encoded straightforwardly as logical formulae. It follows naturally from this assumption that the object of machine learning is to *generate theorems*. This topic is also of interest to automated theorem proving; Wos (1988) includes *theorem finding* in a list of crucial research problems in the field. Unfortunately, Wos's discussion refers only to deductive theorems. Machine learning strives to create other types of theorem, including inductive ones.

What type of logic should be chosen for this endeavor? For some applications (like logic programming), it would be most natural to use first order predicate calculus. The equally powerful equational logic, on the other hand, is much more in keeping with the spirit of functional programming. Functional programming often uses the computational mechanism of term rewriting, and equational logic provides a declarative semantics for term rewriting methods.

Everyday thinking and mathematical reasoning use rewriting quite regularly. Computer applications of term rewriting include:

- automated theorem proving
- parallel functional languages
- modelling nondeterministic computations
- program verification
- abstract data type specifications
- algebraic simplification
- semantics of programming languages (including Prolog).

This thesis uses the equational logic/term rewriting framework because of its wide application and because its use in machine learning has not yet been explored.

The next part of the chapter reviews previous work in this area. Then the thesis and objectives of this thesis are stated, followed by a brief preview of what is to come. The final portion of the chapter introduces the logic and the technical terms and symbols used in the remainder of this work.

1.1 Semantics of Induction

One problem addressed in this thesis concerns the meaning of induction:

Problem 1. *What is meant by the term "induction"? More precisely, how can induction be formalized in logic in a clear way, capturing the intended meaning of machine learning and automated theorem proving researchers?*

A full answer to the first part of the question is difficult. Indeed, it has been a primary concern of philosophers, from Aristotle (and before) to the present day, and is still far from resolved. If the question cannot be answered in general, it may still be possible to respond to it in the narrower context of a particular research interest. Thus, the second part of the problem trims the question down to the context of machine learning.

We might loosely define an induction as an *abstraction* or *emergent property* of a set of facts. However, this raises the obvious question of the meaning of "abstraction" and how it is somehow connected with the set of facts. Several definitions of induction have been attempted, yet few have adequately explained this connection between an induction and the data it is based upon.

One popular definition focuses on the assumptions that are necessary to turn induction into a deduction (see eg. Skyrms, 1975).

E – examples
A – hidden assumptions

I – inductive statement

For example, suppose that I is the statement "the sun will rise tomorrow" and E comprises all the examples of days when it rose, namely every day seen so far. Then the hidden assumption A could be "tomorrow will be just like today and previous days with respect to sun-rising." I clearly follows from E together with A.

This model of induction is weak in two ways. First, the hidden assumptions might be complex and unjustified. Generating A on a computer would be tricky without additional constraints on the nature of acceptable assumptions. For example, "tomorrow will be just like 25 trillion days ago with respect to sun rising" is a complex assumption that will not prove that the sun will rise tomorrow. A computer should not have to even consider hidden assumptions such as these. Second, almost anything follows from E and an *unspecified* A. Any I can be made into an induction unless it contradicts something in E. In other words, I must require some further justification for this to form a basis of a useful definition — it must be somehow based on the examples. This justification is the source of much philosophical controversy, and is usually couched in extralogical, statistical or simply imprecise terms. Thus this definition, without further *logical* machinery, is not appropriate for the present purposes.

Another popular description of induction defines it in terms of a deduction (see Genesereth and Nilsson, 1987). Suppose that B is a given background theory. Then I is an inductive statement when

$$B \wedge I \models E \text{ and } B \not\models I.^1$$

For example, in the sun-rising question let I be the statement "the sun rises every day" and B be empty. Then I clearly covers all of E (as well as tomorrow!).

Although I here is forced to cover part of E, it does not have to cover *all* of E. For example, suppose three people, Sandy, Chris and Jo, are examples of happy people. Suppose that Sandy and Chris are known to be happy because they just won a million dollars each in a lottery, and people with lots of money are known to be happy (B). Then, this definition surprisingly says that "Jo is happy" (I) is one possible inductive conclusion. That is not too bad; later we will propose the stronger view that deductions are also conveniently considered

[1] For now, read $A \models C$ as "C is a logical consequence of A." Section 2.2 defines this notion more precisely.

as inductions. Fortunately, the alternative, more appropriate, induction "everyone is happy" is allowed by this definition.

There are difficulties in basing the treatment of induction entirely on deduction. Perhaps the primary problem with the above definition is that it is unintuitive. We normally think of an induction emerging from given information rather than that information being explained *a posteriori* by the inductive statement. Instead, the definition should take the form

$$K \models_{ind} I$$

where K is the given information. Here induction is a particular rule of inference rather than an extralogical view of a particular pattern of deductive inferences.

Second, this deduction-based definition complicates the simple notion of induction by insisting on the artificial distinction between the background knowledge B and the example set E. While this may have some benefit in terms of efficiency, it seems to be merely an implementation issue that has crept its way into the definition of a basic logical process. The distinctions may be useful in evaluating inductions, but not necessarily for defining them. For simplicity's sake, it is better to think of inductions as emerging from a single base of information without regard to its sources. A further argument for removing the distinction between background theory and examples is that the former often plays a similar role to examples in producing inductions. In the lottery example, with B and E as before, another perfectly valid induction would be "people are happy if they are lottery winners." For this induction, the background information that Sandy and Chris are lottery winners is as important as the "example" information that they are happy. This induction is just as much about the background theory as it is about the examples.

A definition of $\models_{ind}$ that derives from the automated theorem proving community (see primarily Zhang, 1988) is based on the idea that a statement is an *inductive theorem* if and only if all of its ground instances are *deductive theorems*:

$$K \models_{ind} I \text{ iff } \forall I_i \; K \models I_i, \text{ where } I_i \text{ is a ground instance of } I.$$

This definition corresponds to what is known as *summative induction* in the philosophical literature (Von Wright, 1957). However, it does not address inductions that are based on

"missing information." For example, including an additional lottery player, Randy, into our simple example gambling world without including any information about Randy's happiness would invalidate the inductive conclusion "everybody is happy." However, it would be convenient to *assume* that Randy is happy so that the induction "everybody is happy" would be possible. Using such assumptions in order to form inductions is called *ampliative induction.* A complete theory should enable such inductions involving assumptions. Furthermore, one might allow the dismissal of some data in order to make an induction. For example, even if Randy was known to be unhappy it might still be useful to induce that everyone was happy despite this minor inconsistency. This type of induction we call *ignorative induction.* Also, we call a combination of both ampliative and ignorative induction *prudent induction.*

One way to capture ampliative and ignorative inductions in a general logical theory of induction stems from the realization that the former are simply special kinds of satisfiable formulae, while the latter are special kinds of inconsistent formulae. The restrictions on satisfiable formulae include the idea of explicitly stating the type of justification used to define a particular class of theorems. For example, in Chapter 2 we show how the definition of inductive theorems is based on "total justification" (all instances of the theorem are true), that of ampliative theorems on "biexemplar justification" (two instances are true) and others types on "minimal justification" (one instance is true).

Another approach to the problem is to base induction on non-monotonic rules of inference. A full treatment of this topic did not appear until recently (Helft, 1989). The main idea of this work is that nonmonotonistic preference can be used to bias the concept language during the generalization phase in a machine learning setting. It shows that nonmonotonic reasoning certainly has applications in induction and machine learning, but as yet can hardly be considered a general framework for induction.

This thesis uses nonmonotonicity as well, but from a very different angle. A more general, simple model is developed in Chapter 2, based on the insight that the summative inductive rule of inference used in automated theorem proving can be used to define ampliative and even noisy inductions.

This thesis deliberately ignores statistical approaches to defining induction (eg. Carnap, 1962; Hintikka, 1964). We are interested in what can be represented symbolically without

the use of numbers. It will transpire that *objective* probabilities and other statistical methods are likely needed to evaluate inductions. This study focuses on logical inference and does not entertain any notion of subjective probability.

1.2 Computational Induction

Having formalized the meaning of induction, in many of its guises, we proceed to explore how inductive theorems can be generated:

Problem 2. *How can inductive theorems be generated? Can some existing theorem proving techniques be adapted to generate theorems? In particular, how might such theorems be generated in an equational theory?*

As mentioned earlier, the aim of machine learning is to *generate* theorems rather than to prove them. It is not surprising that generation can be accomplished by methods that are roughly the opposite of theorem proving techniques. Reconsider the deduction-based definition of induction:

$$B \wedge I \models E \text{ and } B \not\models I.$$

E in this definition is produced from B and I using deductive rules of inference — resolution, for example. If the inverse of resolution were used on E and B we would expect to obtain I, the inductive theorems.

The basic idea is not new, although it appears never to have been fully developed. According to Plotkin (1971), Popplestone suggested that "since unification is useful in automatic deduction by the resolution method, its dual might prove useful in induction," and Plotkin, his student, responded in the same article with an early study of antiunification. Antiunification, like unification, attempts to make two terms look alike; unlike unification, it achieves this by replacing subterms with variables rather than variables with subterms, producing a term that is more general than the original ones. Since antiunification is the basic generalization method, it forms the foundation of the present work and will be described in great detail in Chapter 4.

On its own, term generalization is far from adequate for generating inductive theorems. Commonly-used logics comprise not only single term statements, but implications, conjunctions, disjunctions, negations and quantification. Vere (1977) described a generalization method that generates the most specific conjunction from a set of terms. A further improvement to his algorithm allowed counterfactuals — statements of the form $A \wedge \neg B$ (Vere, 1980). Unfortunately, neither method allows full use of a background theory, even for its restricted hypothesis language. Furthermore, the algorithm is not based on any precise logical principles. While Mitchell's (1982) work on version spaces is more general, it nevertheless suffers the same flaws.

A major step in the right direction appeared in Shapiro's (1983) "model inference system," which introduced a sound, logic-based, general-to-specific refinement method. What is particularly interesting is the use of a logical semantics as a foundation for the induction methods. An interesting insight in this work is that machine learning is about describing, or summarizing, models using logical formulae. While commendable for its logical coherence, this work is restricted to the most commonly studied subset of first order logic, namely Horn clauses.

Most closely related to our ideas on generation are the techniques of Muggleton and Buntine (1988) They show how new Horn clauses can be generated from others using the inverse of a specific type of resolution (SLD), underscoring the present thesis that generation techniques are inversions of theorem proving methods. While this work may ultimately form the basis of a more complete, general resolution-based theory, it currently depends on many restrictions that do not allow the use of the full Horn clause logic. These restrictions are necessary for accomplishing *constructive induction*, the generation of new predicate symbols to aid in induction, a topic avoided by this thesis.

Other systems use techniques that should properly be thought of as inverse theorem proving. For example, least general generalization methods are the basis of some important induction algorithms (Vere, 1977; Vere, 1980; Buntine, 1987; Kodratoff, 1988). Most of these are extensions of Plotkin's algorithm (1970), a clause generalization method. None link these methods to a definition of induction. Some are incomplete in that they do not allow the use of background theories. The present research emphasizes that least general generalization algorithms should be thought of as inverse rules of inference in order to demonstrate their

soundness and completeness. Other algorithms, such as the ID3 series of classification techniques, are based on specializations, and are generally unrelated to the semantics of induction. We view specialization algorithms as reverse deductions as well. In a nutshell, theorem generation techniques should be based on inverse deductive theorem proving to provide a clear operational semantics. This thesis shows how this can be accomplished.

A distinguishing aspect of the present work is that it studies induction in equational logic rather than classical Horn clause logic, providing a bias toward functional rather than logic languages. One reason for preferring equational logic is its ease of extension. For example, continuous values, sets and simple conditionals require very little extension to the basic induction operators. Another reason is that machine learning is relatively less researched in functional languages than in logic languages. With a problem as difficult as theorem generation, it is worthwhile studying it in as many interesting subcases as possible.

1.3 Thesis

For pragmatic purposes, *induction* is an inadequately defined concept. This results in poorly designed systems for machine learning and leads to conceptual confusion. It can be addressed by formulating the varieties of induction precisely within a logical framework. As well as providing increased clarity, this suggests generating inductive theorems by inverting theorem proving techniques. Consideration of two particular generation methods, namely *antinarrowing* and *cover set specialization*, reveals their theoretical and practical power for generating several types of inductive theorems.

The objectives of this thesis are

1. To review methods and characterizations of induction, particularly with respect to machine learning.
2. To develop a logical framework for various types of inductive theorems that are useful for machine learning.
3. To review theorem proving techniques as a prelude to developing generation techniques.

4. To develop operators (rules of inference) that can be used to generate theorems and demonstrate their soundness and completeness.

5. To develop algorithms based on the generation operators and to implement them on a computer.

6. To describe some machine learning systems in terms of the generation operators developed.

7. To determine limitations of the algorithms and operators and to suggest additions or alternatives.

These objectives will be met in the following way: The first part of this chapter reviewed some ideas about induction (objective 1). The remainder gives mathematical preliminaries, and introduces the logic to be used. Chapter 2 defines several varieties of induction by identifying each type with a class of theorems. Some restrictions on these classes are discussed to show how they can be applied to machine learning (objective 2). Chapter 3 reviews theorem proving techniques for both deductive and inductive theorems (objective 3). These provide the basis for the theorem generation operators discussed in Chapter 4 (objective 4). Chapter 4 gives some propositions characterizing the power of these methods with respect to the theorem classes defined previously. Chapter 5 shows how the theory developed in the previous chapters can be used in a variety of ways to obtain very different styles of machine learning algorithm (objective 5). The algorithms of Chapter 5 are implemented in Balog, a system developed for equational logic theorem proving and generation (objective 5). Chapter 6 briefly considers other machine learning algorithms (objective 6) in light of the theory developed. Finally, Chapter 7 discusses some limitations of the theory and possible future directions (objective 7).

1.4 Mathematical Preliminaries

This thesis contains a wide variety of notational forms. This section reviews the language used, and can be used for reference while reading later chapters.

$S = \{\mathbf{num}\}$
$V = \{x, y, z, n, m\}$
$F = \{s/1, \mathbf{0}/0, pred/1\}$
$C = \{s, 0\}$

$$\begin{array}{lcl} Type(0) & = & \mathbf{num} \\ Type(s) & = & \mathbf{num} \\ Type(pred) & = & \mathbf{num} \end{array} \qquad \mathcal{Q} = \left[\begin{array}{l} 0 = pred(0) \\ n \doteq pred(s(n)) \end{array} \right]$$

Figure 1.1. A theory: predecessors of integers.

In the following, V denotes a set of variables and F a set of function symbols. The *arity* of a function is the number of arguments it takes. When specifying functions, the arity will be given along with the name. Here is a sample specification of a set F of functions:

$$F = \{f/2, g/1, c/0\}$$

A *constant* is a function symbol with arity 0. In F, for example, c is a constant while g and f are not.

A *term* is

(1) a variable
(2) a constant
(3) of the form $f(t_1, t_2, ..., t_n)$ where f is a function with arity n and each t_i is a term.

A *ground* term is one that contains no variables. We denote the terms built from a set of functions F and variables V as $Terms(F, V)$, and ground terms as $Terms(F)$. The set of variables which occur in a term is denoted $Vars(t)$. A term t can be *grounded* by replacing each variable in $Vars(t)$ with a constant. Assume, unless otherwise stated, that these constants are generated to be completely new to the context, that is, they are not in F if F is specified. The grounded version of a term t is denoted $Gnd(t)$. For example, if $F = \{f/2, g/1, c/0\}$, $Gnd(f(x, f(y, x))) = f(c1, f(c2, c1))$ where $c1$ and $c2$ are any constants other than c.

Terms may also be viewed as *trees* whose leaves are labelled with variable or constant symbols, and internal nodes with function symbols of arity k where k is the number of

children of the node. A *position* is specified by a sequence of natural numbers, each number separated by a ".". The root of the term tree is defined to be at position ϵ (the empty sequence), the leftmost child of the root is at position 1, and the third child of the second child of the root is at position 2.3. The set of all positions in a term is denoted $Pos(t)$. Consider the term $t = f(d, g(a, h(b)), c)$. The subterm a is at position 2.1 and f is at position ϵ. t/p denotes the subterm at position p. $t[p \leftarrow s]$ is the term resulting from the replacement of the subterm at position p with the term s. So $t/2.2 = h(b)$ and $t[2 \leftarrow e] = f(d, e, c)$. Also, $Pos(t) = \{\epsilon, 1, 2, 2.1, 2.2, 2.2.1, 3\}$.

An *equation* has the form $s = t$ where s and t are terms. The set of equations that can be built from $Terms(F, V)$ is denoted $Eqns(F, V)$. The set of all *ground equations*, that is, ones without variables, is denoted $Eqns(F)$. Note that it is possible for F to contain only constants. $s \equiv t$ denotes syntactic equality and $s =_{\mathcal{E}} t$ denotes semantic equivalence with respect to a background theory $\mathcal{E}$. In other words, $s =_{\mathcal{E}} t$ means that terms s and t can be transformed into each other by using the equations in $\mathcal{E}$. For example, $f(a, b) =_{\mathcal{E}} f(c, b)$ when $\mathcal{E}$ contains the equation $a = c$.

We are now able to define the logic that is to be used in the rest of this thesis. A *first order typed equational theory* $\mathcal{E}$ is a pair $(\Sigma, \mathcal{Q})$. $\Sigma = (L, S, V, F, C, Type)$ is an alphabet consisting of a set of logical symbols $L = \{\neg, \forall, \exists, (,), \wedge, \vee\}$, a set of sorts S, a set of variables V, a set of function symbols F, a set of constructor functions C, and a typing function *Type*: $F \rightarrow S$. The typing function assigns a type specification of the form $\sigma_1 \times \sigma_2 \times \ldots \times \sigma_n \rightarrow \sigma$ to every function symbol $f \in F$ of arity n. We say that f has *range* type of σ. For example, if $F = \{and/2, true/0, false/0\}$ and $S = \mathbf{bool}$, *Type* could be defined as: *Type*(*false*) = **bool**, *Type*(*true*) = **bool**, *Type*(*and*) = **bool** $\times$ **bool** $\rightarrow$ **bool**. We also define the typing function for terms as the range type of the function symbol at position ϵ. So $Type(and(true, false)) = Type(and) = \mathbf{bool}$.

A *well formed formula* (wff) is defined recursively:

- if s and t are terms then $s = t$ and $s \neq t$ are wffs.
- if t is a term, then $t = true$ and $t = false$ are wffs (these are sometimes abbreviated as t and $\neg t$ respectively).

- if F is a wff and x is a variable, then $\forall x\ F$ and $\exists x\ F$ are wffs.
- if A and B are wffs, then $\neg A$, $A \vee B$, and $A \wedge B$ are wffs.

The wffs generated by a set of terms $\mathit{Terms}(F, V)$ are denoted $\mathit{Wff}(F, V)$. The set of ground wffs — those without variables — is denoted $\mathit{Wff}(F)$. Furthermore, we identify the *first order typed equational language* of Σ as $\mathit{Wff}(F, V)$ itself.

$\mathcal{Q}$ is a set of equations and inequations called the *axioms* of the theory; these are a subset of $\mathit{Wff}(F, V)$. They are specified in a normal form; in other words existential variables will have been skolemized out and free variables are assumed to be universal. Unless otherwise stated, we will assume that the language is negationless: no use of the $\neg w$ or $a \neq b$ forms will be allowed in $\mathcal{Q}$.

An example of a completely specified first order typed equational theory is given in Figure 1.1. A theory is often just specified with a set of equations (or rules, see Section 3.2) when the types, functions and variables intended are obvious.

Finally, *constructor functions* are special types of functions that can be thought of as those used to build up the "data" in the theory. Any other function in F is called a non-constructor. The distinction between constructors and non-constructors is useful for many applications, and for induction in particular, as we will see in Section 2.4. A *constructive* term is one that is built solely out of constructor function symbols and variables. If c_1 and c_2 are two different constructive terms, then $c_1 \neq c_2$ is an *implicit axiom* found in $\mathcal{Q}$. Since these are too numerous to specify in practice, they are usually not listed as part of $\mathcal{Q}$.

Be aware that a few more important concepts, the application and composition of substitutions, will not be introduced until Section 3.1 even though they will be used in Chapter 2.

Chapter 2

Semantics for Induction

> It's one thing to be able to say "I've got a theory," quite another to say "I've got a semantic theory," but, ah, those who can claim "I've got a deep semantic theory," they are truly blessed. – Randy Davis

At first glance, there seem to be two types of induction. One is treated by the theorem proving community, and the other is of concern to machine learning researchers. They are known as *summative* and *ampliative* induction respectively (Von Wright, 1957). Summative induction was first mentioned by Aristotle (1928):

> "Induction proceeds through an enumeration of all the cases."

In other words, if all possible instances of a proposition hold, the proposition itself holds. Ampliative inductions, on the other hand, also hold for unknown cases. This gives an induced proposition predictive capability, which is valuable for machine learning applications. However, there is a third type of induction that allows the introduction of error, either accidentally or on purpose. These propositions, which we call *ignorative*, can still be quite useful in making inductive-like assertions. A fourth type of induction holds for unknown cases and allows errors, and we call this *prudent* induction.

This chapter will characterize four types of induction, summative, ampliative, ignorative and prudent, by showing how to capture these distinctions quite naturally in a model-theoretic logical setting. Figure 2.1 shows that the prudent theorems that we are about to develop will have (summative) inductive, ampliative, ignorative and deductive theorems as subclasses. These four types of inductive theorems are examples of *inductivelike theorems* (Definition 2.6) that are characterized by an *inductive modelling type* (Definition 2.5) and a *justification method* (Definitions 2.7, 2.9, 2.10, 2.11).

We begin with some basic model theory (Section 2.1) that enables us to define the deductive and (summative) inductive classes of theorems (Section 2.2). We next develop the

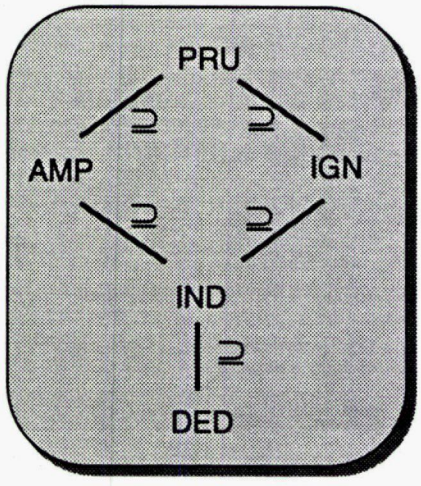


Figure 2.1. Theorem Classes in this Chapter

idea of inductivelike theorems and show that (summative) inductive theorems are a trivial special case (Section 2.3). We then show that standard methods are inadequate for dealing with unknown values, and with noise (Section 2.4.1). Next, following Rescher and Brandom (1979) and Langholm (1988), non-standard models are introduced by extending the possible truth values to include not only true and false but also *underdetermined* and *overdetermined* (Sections 2.4.2 – 2.4.4). The preference logic that Shoham (1988) introduces as a basis for non-monotonic reasoning is here adapted to obtain a natural definition of ampliative, ignorative, and prudent theorems (Sections 2.4.5 and 2.4.6). Finally, we argue that machine learning systems are inductivelike theorem generators (Section 2.5).

2.1 Interpretations and Models

Semantics enables meaning to be attributed to statements made in a language. For a full semantics of a language, all of its constituent parts must be well defined.

An *interpretation* helps in this regard by fixing the meaning of each symbol in the language. For example, two different interpretations of the set of symbols $A = \{1, 2, 3, 4, 5\}$ are assignments of A to the *domains* of *the first five positive integers* and of *the fingers on a piano player's hand.* A *model* of a statement is an interpretation in which the statement

is true. The following definitions, which presuppose a first order equational theory $\mathcal{E} = ((L, S, V, F, C, Type), \mathcal{Q})$ as defined in Section 1.4, will make these concepts more precise.

Definition 2.1 *A* **standard interpretation** $\mathcal{I} = (D, \mathcal{K}, \mathcal{G})$ *is composed of a domain D, a domain mapping* $\mathcal{K}$*, and an equational truth mapping* $\mathcal{G}$*, such that*

1. $D = \{d \mid d \in D_T, T \in S\}$ *is a domain. A* **domain** *is any nonempty set of objects. A domain is partitioned into subdomains based on the types, S, of the language.* D_T *is called the* **domain of type** T.
2. $\mathcal{K}$ *:Terms*$(F) \rightarrow D$ *is a* **domain mapping** *where* $\mathcal{K}(f(t_1, \ldots, t_n)) = d$ *for some* $d \in D_{\text{Type}}(f)$.
3. $\mathcal{G} : Eqns(D) \rightarrow \{\mathbf{T}, \mathbf{F}\}$ *is an* **equational truth value mapping** *where there are no* e_1 *and* e_2 *in* $Eqns(D)$ *such that* $e_1 =_{\mathcal{E}} e_2$ *and both* $\mathcal{G}(e_1) = \mathbf{T}$ *and* $\mathcal{G}(e_2) = \mathbf{F}$.

The definition of truth value mappings in this chapter depend on the equational truth value mapping, $\mathcal{G}$, that is given as part of the interpretation. More generally, $\mathcal{G}$ could have been defined over all predicates, not just over the equality predicate. However, the language that is used in this thesis has only the equality predicate. The equation mapping will range over a nonstandard set of truth values later in this chapter. Note that $\mathcal{G}$ is defined only over ground sentences.

An important constraint on the definition of $\mathcal{G}$ is the **unique names assumption**. This requires that all constants (functions of arity zero) in the language are not equal unless otherwise specified by $\mathcal{Q}$. So, $\mathcal{G}(a = b) = \mathbf{T}$ iff $a = b \in \mathcal{Q}$, when a and b are constants. This condition can easily be formulated as a preference relation on models (see Section 2.4), but we will not do so here to avoid over-complexity.

Given an interpretation and any ground equation, we can determine the meaning of the equation. But to determine the meaning of more complex statements that include variables, quantifiers, negations, conjunctions and disjunctions, a *truth value mapping* is required. This is a function that assigns to each statement in a language a truth value in $\{\mathbf{T}, \mathbf{F}\}$.

Definition 2.2 *A* **standard truth value mapping** $\mathcal{T}_s$ *with respect to an interpretation I is a mapping where* $\mathcal{T}_s$ *: Wff(F,V)* $\rightarrow$ $\{\mathbf{T},\mathbf{F}\}$ *such that* $\mathcal{T}_s(w) = \mathcal{T}_2(K(w))$ *for* $w \in$ *Wff(F,V).* $\mathcal{T}_2$ *is*

defined in Figure 2.2.

$\mathcal{T}_s$ takes a well formed formula and converts its terms to symbols in the domain using the domain mapping K. It passes this partially interpreted formula to $\mathcal{T}_2$ which determines its truth value.

This truth value mapping relates *every* well formed formula to its truth. If a sentence evaluates to **T** over an interpretation I, it is said to be *true in* I, otherwise it is *false in* I. Note that one need not specify a new definition of the truth value mapping for each new theory; the definition remains constant across all (standard) theories. The definitions of the truth value of the *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$) connectives are given in Figure 2.3(a) and Figure 2.4 (the $\wr$ and $\times$ truth values in this figure will be described later).

An interpretation and a truth value mapping enables us to determine what a sentence in a theory means.

Definition 2.3 *A* **standard model** *of a sentence $w \in \mathcal{E}$ is an interpretation I such that w is true in I. A model of a set of sentences is a model of each sentence in the set.*

In Figure 2.5 $\mathcal{M}_1$ and $\mathcal{M}_2$ are both models of $\mathcal{E}$. Notice that all models are *consistent* in that the same equation is not assigned to both **T** and **F** ($\mathcal{G}$ is a *function*). Also notice that all the equational consequences of the axioms are assigned to **T**. This is the least that is required of a model. However, note that $\mathcal{M}_2$ includes an assumption, namely $mother(karla) = joe$ (ridiculous, but possible in a strange world!), and all of this assumption's consequences are also included in the model. The model $\mathcal{M}_1$ is called a **minimal model** because the truth value mapping maps as much as is possible to **F**. Note also that it is common practice to simply list the true equations and assume that the rest are false.

Two-valued Truth Mapping

$$T_2 : S \rightarrow \{T,F\}$$

Form of S	Conditions	Truth Value
$\in$ Eqns(D)		G(S)
$\exists x\, W$	$T_2(W\{x/d\}) = T$ for some $d \in D_{Type(x)}$	T
$\forall x\, W$	$T_2(W\{x/d\}) = T$ for all $d \in D_{Type(x)}$	T
A v B		$T_2(A)$ v $T_2(B)$
A & B		$T_2(A)$ & $T_2(B)$
$\neg$ A		$\neg\, T_2(A)$
	Otherwise	F

Figure 2.2. Standard Truth Mapping

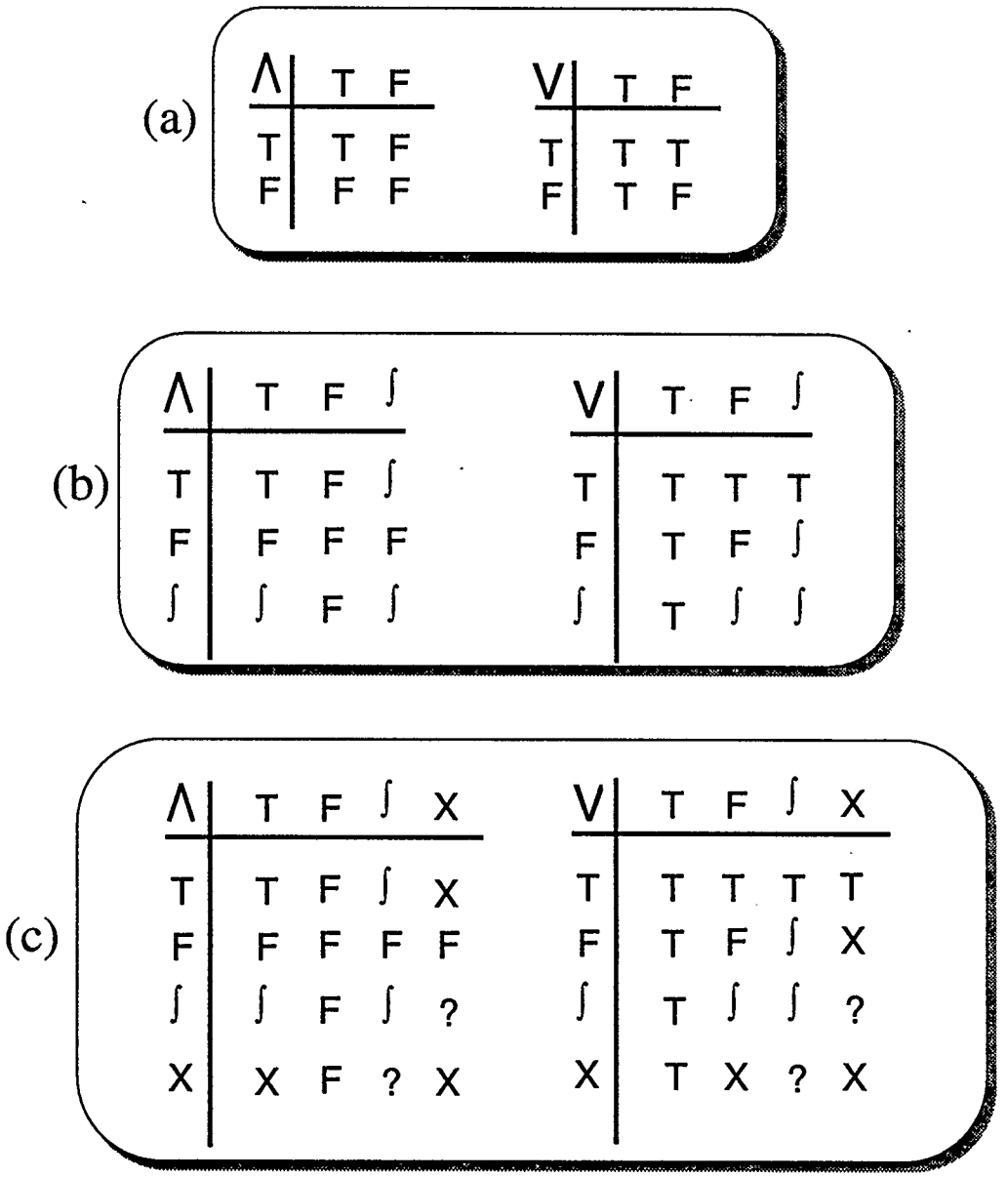


Figure 2.3. Truth Values for Truth Connectives

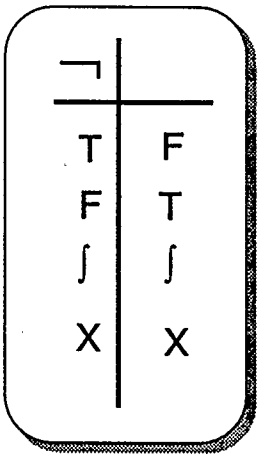


Figure 2.4. Truth Values for Negation

2.2 Deduction and Induction

Using standard models, deduction and induction can be defined straightforwardly.

2.2.1 Deductive theorems

The most common theorems are *deductive theorems*. Resolution theorem provers, conditional term rewriting systems and natural deduction systems prove such theorems. When the phrase *logic* is used, most people think of these theorems. However, there is much more to logic, including some non-deductive reasoning methods that this chapter will describe. First, the notion of *logical consequence* is defined in terms of the standard models of a theory.

Definition 2.4 *Suppose $\mathcal{E}$ is a theory with $\mathcal{Q}$ as its axioms and d is a well formed formula of $\mathcal{E}$. Then d is a* **logical consequence** *of $\mathcal{E}$ if all models of $\mathcal{Q}$ are models of d.*

From this definition, we see that a statement is a logical consequence if it is true no matter how the symbols of the theory are interpreted and no matter what truth values are assigned to statements whose truth is not determined by the theory's axioms.

The definition of deductive theorems follows directly from the definition of logical consequence.

$$\mathcal{E} = ((L, S, V, F, C, Type), \mathcal{Q}) \text{ where}$$
$$C = \{joe, betty, karla\}$$
$$F = C \cup \{mother/1, grandmother/1\}$$

$$\mathcal{Q} = \begin{bmatrix} mother(joe) = & betty \\ mother(betty) = & karla \\ grandmother(x) = & mother(mother(x)) \end{bmatrix}$$

$$\mathcal{M}_1 = \begin{pmatrix} mother(joe) = betty & \mathbf{T} \\ mother(betty) = karla & \mathbf{T} \\ grandmother(joe) = mother(mother(joe)) & \mathbf{T} \\ grandmother(joe) = mother(betty) & \mathbf{T} \\ grandmother(joe) = karla & \mathbf{T} \\ grandmother(betty) = mother(mother(betty)) & \mathbf{T} \\ grandmother(betty) = mother(karla) & \mathbf{T} \\ grandmother(karla) = mother(mother(karla)) & \mathbf{T} \\ otherwise & \mathbf{F} \end{pmatrix}$$

$$\mathcal{M}_2 = \begin{pmatrix} mother(joe) = betty & \mathbf{T} \\ mother(betty) = karla & \mathbf{T} \\ grandmother(joe) = mother(mother(joe)) & \mathbf{T} \\ grandmother(joe) = mother(betty) & \mathbf{T} \\ grandmother(joe) = karla & \mathbf{T} \\ grandmother(betty) = mother(mother(betty)) & \mathbf{T} \\ grandmother(betty) = mother(karla) & \mathbf{T} \\ grandmother(karla) = mother(mother(karla)) & \mathbf{T} \\ joe = mother(karla) & \mathbf{T} \\ grandmother(karla) = betty & \mathbf{T} \\ grandmother(karla) = mother(joe) & \mathbf{T} \\ grandmother(betty) = joe & \mathbf{T} \\ otherwise & \mathbf{F} \end{pmatrix}$$

Figure 2.5. Models of a simple theory

Deductive Theorems (DED)

$d \in DED(\mathcal{E})$ iff d is a logical consequence of $\mathcal{E}$.

If d is a deductive theorem of $\mathcal{E}$ we write $d \in DED(\mathcal{E})$. Consider the theory $\mathcal{E}$:

$$\mathcal{Q} = \left[\begin{array}{ll} \mathit{feline}(x) & = \mathit{cat}(x) \\ \mathit{canine}(x) & = \mathit{dog}(x) \\ \mathit{cat}(\mathit{felix}) & = \mathit{true} \\ \mathit{cat}(\mathit{morris}) & = \mathit{true} \\ \mathit{nice}(\mathit{felix}) & = \mathit{true} \\ \mathit{nice}(\mathit{morris}) & = \mathit{true} \end{array}\right] \qquad \begin{array}{l} \mathrm{V} = \{x\} \\ \mathrm{S} = \{\ \text{bool, cattype, dogtype}\} \\ \mathrm{F} = \{\mathit{morris}/0, \mathit{felix}/0, \mathit{cat}/1, \\ \qquad \mathit{dog}/1, \mathit{canine}/1, \mathit{feline}/1, \\ \qquad \mathit{nice}/1\} \end{array}$$

Then $\mathit{feline}(\mathit{felix}) \in DED(\mathcal{E})$[1] since all models of $\mathcal{Q}$ must contain *cat(felix)* and, by the first equation, are forced to contain *feline(felix)* as well. Similarly, $\neg \mathit{canine}(\mathit{felix}) \notin DED(\mathcal{E})$ since there are models of $\mathcal{Q}$ which have $\mathrm{T}_s(\mathit{canine}(\mathit{felix})) = \mathrm{T}$ and others with $\mathcal{T}_s(\mathit{canine}(\mathit{felix})) = \mathrm{F}$. Note that *dog(felix)* is not a deductive theorem while *dog(felix) = canine(felix)* is.

2.2.2 Inductive theorems

There are many formulae which seem reasonable, but are nevertheless not deductive theorems. In the theory above, for example, $\forall x\ \mathit{cat}(x)$, that everything is a cat, seems reasonable, since *felix* and *morris* are the only beings known in the theory. Further, $\forall x\ \mathit{cat}(x) = \mathit{nice}(x)$, that all cats are nice and all nice beings are cats, seems even more reasonable. However, neither of these are deductive theorems since there are many models in which *cat(lassie)* is false and others in which *cat(mccavity)* is true but *nice(mccavity)* is false[2].

But why should we be talking about Lassie and McCavity Cat when they are not even mentioned in our theory? Recall that a deductive theorem in a theory is supposed to be true independent of the context or the interpretations of the symbols in the theory. That is,

[1] If P is not an equation, $P \in DED(\mathcal{E})$ means $(P = \mathit{true}) \in DED(\mathcal{E})$.

[2] Lassie the dog and the not-so-nice McCavity Cat are famous.

statements like *feline(felix)* or $\exists y\ cat(y) = nice(y)$ will always be deductive theorems even if we do add statements about Lassie or McCavity to the theory. On the other hand, statements like $\forall x\ cat$(x) are not always true.

One approach is to define such theorems by considering only minimal models rather than all models. A **minimal (initial) model** is a model that assigns statements to **F** when they are not implied to be either true or false by the axioms (rules/equations) of the theory. We could say that a statement is an inductive theorem if it is true in all minimal models rather than in *all* models. This approach is known as *initial model induction* and forms the basis for the *inductionless induction* theorem proving technique (Lankford, 1981). In Section 2.4.1, we will see that initial model induction is inadequate for our purposes.

Another natural way to define inductive theorems is to rely on the types and function symbols in the specification of the theory. Then we define inductive theorems to be statements whose instances are all deductive theorems, rather than a type of deductive theorem over a subset of models:

Inductive Theorems (IND)

$\mathcal{I} \in IND(\mathcal{E})$ iff $\mathcal{I}\theta \in DED(\mathcal{E})$ for all ground substitutions θ
where $\mathcal{I}\theta \in Wff(\mathcal{E})$

$\mathcal{I} \in IND(\mathcal{E})$ means that $\mathcal{I}$ is an inductive theorem of a theory $\mathcal{E}$. Using this definition, it follows that all deductive theorems are inductive theorems as well since all instances of a deductive theorem are deductive theorems. The final clause ensures that substitutions do not violate typing conventions and do not use function symbols that are not in the specification of the theory.

This definition of induction corresponds to Aristotle's notion of summative induction. By enumerating all possible instances of a statement and showing that each one is a deductive theorem, the inductive theoremhood of the sentence is established.

This definition does not handle ampliative induction. Suppose it is known that *bowser*, *rolf* and *pluto* are dog-names and that the following are axioms:

$$\mathcal{Q} = \begin{bmatrix} nice(bowser) \\ nice(rolf) \\ dog(bowser) \\ dog(rolf) \\ dog(pluto) \end{bmatrix}$$

There is no longer a proof that $\forall\ x\ dog(x) = nice(x)$ is an inductive theorem. However, it seems that it would be a good assumption to make that *nice(pluto)* is true as well. With this assumption, the theorem can easily be shown to be an inductive theorem. This example illustrates that there is another class of theorems that includes these types of formulae.

2.3 Inductivelike Theorems and Justification

We will shortly be developing three classes of theorems that differ from inductive theorems in the types of interpretations and the method of justification that they use. In this section, we describe what makes these classes similar. We then describe some forms of justification other than the one used for the *IND* class of theorems.

2.3.1 Inductivelike theorems and total justification

All the classes of theorems described in this thesis share two properties: none of their instances are false[3] and they are somehow *justified* by their axioms. This can be defined formally as follows.

Definition 2.5 *A model $\mathcal{M}$ of interpretation type IT is an* **inductivelike modelling** *of a statement S of theory $\mathcal{E}$ with justification J iff*

1. *S is justified in $\mathcal{E}$ by J.*
2. *S is* not false *in $\mathcal{M}$.*

[3]They may be true, underdetermined or overdetermined. See Section 2.4.2.

Possible values for IT in this definition are *standard, overdetermined, underdetermined* or *full*. So far only standard models have been introduced.

Definition 2.6 *Consider a statement S in some equational theory $\mathcal{Q}$. If all $\mathcal{M}$ that are models of interpretation type IT that model $\mathcal{Q}$ are also inductivelike modellings of S with justification J, then S is an* **inductivelike theorem** *of $\mathcal{Q}$.*

Normal inductive theorems, $IND(\mathcal{E})$, are inductivelike theorems over standard interpretations. Their justification (J) is given by the definition of inductive theorems itself: all ground instances must be true. This all-or-nothing type of justification is called *total justification*:

Definition 2.7 *A statement S is* **totally justified** *if for all θ, where θ is ground and constructive, $S\theta$ is* true *in all models.*

The second condition for inductivelike modelling theorems follows from total justification and the *coherence property* of standard models which says that everything *true* must be *not false*. Thus, inductive theorems are inductivelike theorems over standard interpretations with total justification.

Using Definition 2.3.1, with standard models and total justification, we obtain the definition of inductive modelling, the type used to determine IND-theoremhood:

Definition 2.8 *A model $\mathcal{M}$ is an* **inductive modelling** *of S if it is an inductivelike modelling of S using standard interpretations and total justification. We write $\mathcal{M} \models\!\!\!\models S$ if $\mathcal{M}$ is an inductive modelling of S.*

We may now redefine inductive theorems using this definition (assuming total justification):

Inductive Theorems (IND)

$$\mathcal{I} \in IND(\mathcal{E}) \text{ iff } \forall \mathcal{M}\ \mathcal{M} \models \mathcal{Q} \Rightarrow \mathcal{M} \models\!\!\!\models \mathcal{I}$$

Note that the definition of deductive theorems can also be written in a similar form:

Deductive Theorems (DED)

$$S \in DED(\mathcal{E}) \text{ iff } \forall \mathcal{M}\ \mathcal{M} \models \mathcal{Q} \Rightarrow \mathcal{M} \models S$$

2.3.2 Non-total justification

Inductive and deductive theorems have total justification. In our new theorem classes, theorems that have less justification will be allowed. Describing the amount of justification could be done probabilistically. However, this work aims to restrict the use of probabilities to *searching* the theorem space, not to defining it. Instead, the idea of a minimal amount of justification is used.

Definition 2.9 *A sentence S is* **minimally justified** *in a model $\mathcal{M}$ if there is some θ such that $S\theta$ is* true *in $\mathcal{M}$. ($\mathcal{M}$ may be standard, overdetermined, underdetermined or full).*

A statement is minimally justified if it has at least one instance for which it is true. For example, the theory $\{f(a) = c\}$ has $\forall x\ f(x) = c$ minimally justified. Notice that this constitutes a large inductive leap. However, also notice that $\forall x\ f(c) = x$ is not minimally justified. It might be possible to define weaker justifications that lie somewhere between no justification and minimal justification. Note that inductive theorems are inductivelike theorems with standard interpretations and *minimal* justification. This follows from the fact that inductivelike theorems with standard interpretations always have total justification, which implies that they are at least minimally justified.

2.3.3 Biexemplar justification

Many machine learning systems look at similarities between parts of a database in order to generate their theorems. This implies that there are at least two examples of the theorem that is to be generated. However, the minimal justifiability criterion requires only that at least *one* instance of the theorem must be in $DED(\mathcal{E})$. A stronger definition of justification is possible, to look for two unrelated instances for which the theorem is true.

Definition 2.10 *A sentence S is* **biexemplar justified** *in a model $\mathcal{M}$ if there are* at least *two substitutions $\theta_1 = \{v_1/s_1, \ldots, v_n/s_n\}$ and $\theta_2 = \{v_1/t_1, \ldots, v_n/t_n\}$ such that $S\theta_1$ and $S\theta_2$*

are true *or biexemplar justified in* $\mathcal{M}$ *and if* $s_i \equiv f(\ldots)$ *and* $t_i \equiv g(\ldots)$ *then* $f \neq g$.

A sentence is biexemplar justified if it has at least two instances that are true or biexemplar justified themselves. They must be suitably "different" — namely, each pair of subterms that replace a variable in the sentence S to obtain $S\theta_1$ and $S\theta_1$ must not be rooted with the same function symbol. The recursive nature of this definition allows cases where a sentence requires three or more true instances for its justification.

This is still a rather weak form of justification. Those theorems captured by biexemplar justification but not by stronger forms of justification (eg, tri-exemplar) are said to be strictly biexemplar justified:

Definition 2.11 *A sentence* S *is* **strictly biexemplar justified** *in a model* $\mathcal{M}$ *if there are* exactly *two substitutions* $\theta_1 = \{v_1/s_1, \ldots, v_n/t_1\}$ *and* $\theta_2 = \{v_1/t_1, \ldots, v_n/t_n\}$ *such that* $S\theta_1$ *and* $S\theta_2$ *are* true *in* $\mathcal{M}$ *and if* $s_i \equiv f(\ldots)$ *and* $t_i \equiv g(\ldots)$ *then* $f \neq g$.

Consider the equations $\{f(g(a)) = g(a), f(g(b)) = g(b)\}$. Then $\forall x\ f(x) = x$ is not biexemplar justified and $\forall x\ f(g(x)) = g(x)$ is strictly biexemplar justified. If the equation base was $\{f(g(a)) = g(a), f(g(b)) = g(b), f(c) = c\}$, $\forall x\ f(x) = x$ would become biexemplar justified though not strictly biexemplar justified.

2.4 Ampliation and Ignoration

The terms ampliation and ignoration are opposing terms. Ampliation refers to the process of amplifying the amount of knowledge in a database. Ignoration refers to the process of ignoring things in a database. Both are useful in machine learning. Ampliative theorems, which were alluded to at the end of Section 2.2, are seemingly reasonable statements that make assumptions about the truth of what is not explictly stated in a database. To see why ignoration can be useful, consider the problem of noise: ignoration can be used to ignore errors in a database.

In this section, we formulate ampliation and ignoration in a logical setting. First we describe the insufficiency of some simple types of nonmonotonic reasoning. Then we show that a simple four-valued logic can meet the needs of ampliation and ignoration. Finally

we introduce the idea of choosing preferred subsets of four-valued models and use this to give a precise definition of ampliative, ignorative and full theorems (theorems that are both ampliative and ignorative).

2.4.1 The insufficiency of minimal and final models

It seems that summative induction, ampliation, and ignoration could be captured by non-monotonic reasoning, a formalized common sense type of reasoning. However, the minimal model approach of methods such as circumscription (McCarthy, 1980) are not directly usable to describe ampliation.

Minimal models are models in which all things unconstrained by the set of equations are assumed to be false. Consider the set of equations $\{p(s(0)) = \mathit{true},\ p(s(s(0))) = \mathit{true},\ p(s(s(s(0)))) = \mathit{true}\}$.[4] The minimal model of this includes $\{p(0) = \mathit{false},\ p(s(s(s(s(0))))) = \mathit{false}\}$. This results in two unintuitive conclusions. First, it shows that $p(0) = \mathit{false}$ is a theorem. That would certainly be strange considering that the rules offer absolutely no justification for it. Although this unjustified inference is useful for some applications (such as default reasoning), it does not help in ampliation. Second, $\forall x\, p(s(x)) = \mathit{true}$ is false as a deductive or inductive theorem, since there is an instance for which it is false. However, we would expect it to be an ampliative theorem.

Using final models instead of minimal models is another possibility. Final models assume everything unspecified to be true. However, this merely shifts the problem from false things to true things. $p(0) = \mathit{true}$ would be true without justification and $\forall x\ q(x) = \mathit{true}$ would also be inductively true.

This minimal model approach provides a semantics for the inductionless induction style of inductive theorem proving. However, we opt to investigate the instance based approaches for proof and generation, because a minimal model semantics for ampliation is difficult, or at least turbid, and it is unclear whether generation operators can be defined that are similar to inductionless induction methods. Also, minimal model approaches seem to be unable to give a semantics for noise. Other reasons for avoiding this approach are given by Zhang

[4] Do not confuse *true* and *false*, which are function symbols, with **T** and **F**, which are truth values.

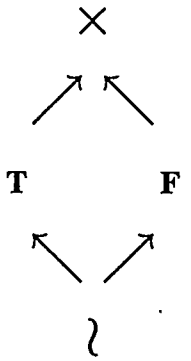


Figure 2.6. Truth Value Lattice

(1988). Nevertheless, there will still be a use for a kind of nonmonotonism, in Section 2.4.5. Ampliation and ignoration will be based on the straightforward definition of inductivelike theorems, rather than taking the minimal model approach.

2.4.2 A new ontology

The simple answer to the problem of defining ampliative and ignorative induction lies in extending the number of truth states that a proposition could possibly be in. In doing so, a four-valued logic is obtained. The extra truth values include both *underdetermined* and *overdetermined* in addition to the standard *true* and *false*. The symbols used for underdetermined and overdetermined are ≀ and × respectively. For a proposition to be underdetermined means that there is no knowledge, or blurred knowledge, that bears on whether a proposition is true or false. An overdetermined proposition is overcommitted, and has both positive and negative (complete) justification — it could be thought of as being both true and false at the same time. All together, the four truth values form the simple lattice (using the specificity relation) seen in Figure 2.6. Belnap (1975) developed a multivalued logic based on a similar set of four values.

The treatment here of propositions concerns their ontological status, not merely some epistemological status. In other words, it is assumed that there is just one world and things in the world — at least the computer's world — actually are unspecifiable and even contradictory. An epistemological approach would assume that there were many worlds

corresponding to knowers within the real world. If it were indeed an epistemological problem a system of modalities might work well; there are many logics of knowledge that have been developed. However, it is unclear who the contradictory and unsure 'knowers' would be inside of a database. Also, modifying our ontology avoids the problem of multiple worlds and the large amount of machinery needed to handle them. This approach to induction simply looks at a single database and describes it in useful ways. If more interesting epistemological problems came up, logics of knowledge could be built upon this four-valued logic.

2.4.3 Underdetermined and overdetermined (three-valued) models

In order to define ampliative and ignorative theorems, underdeterminations and overdeterminations must be defined, along with the truth valuation methods that they induce. We first need to define some non-standard interpretations.

Definition 2.12 *An* **underdetermined interpretation** *is the same as a standard interpretation (Definition 2.1) except that* $\mathcal{G} : Eqns(D) \rightarrow \{\mathbf{T}, \mathbf{F}, \wr\}$ *with the further restriction that when* $e_1 =_\mathcal{E} e_2$ *then* $\mathcal{G}(e_1) = \wr$ *implies* $\mathcal{G}(e_2) = \wr$ *for* $e_1, e_2 \in Eqns(D)$.

The extra condition on $\mathcal{G}$ ensures that an equation is assigned to $\wr$ only when its truth value is not forced to be either **T** or **F**.

Definition 2.13 *An* **overdetermined interpretation** *is the same as a standard interpretation except that* $\mathcal{G} : Eqns(D) \rightarrow \{\mathbf{T}, \mathbf{F}, \times\}$ *with the further restriction that when* $e_1 =_\mathcal{E} e_2$ *then* $\mathcal{G}(e_1) = \times$ *implies* $\mathcal{G}(e_2) = \times$ *for* $e_1, e_2 \in Eqns(D)$.

Here the extra condition on $\mathcal{G}$ ensures that an equation is assigned to $\times$ when all equations equivalent to it are also assigned to $\times$.

The meaning of 'truth' becomes muddled since it is unclear how universal and existential formulae are affected by the introduction of $\wr$ and $\times$. Consequently we extend the notion of truth mapping:

Definition 2.14 *A* **three valued truth mapping** $\mathcal{T}_\wr$ *with respect to an interpretation I is defined where* $\mathcal{T}_\wr : Wff(F, V) \rightarrow \{\mathbf{T}, \mathbf{F}, \wr\}$ *such that* $\mathcal{T}_\wr(w) = \mathcal{T}_3(\mathcal{K}(w))$ *for* $w \in Wff(F, V)$. $\mathcal{T}_3$ *is defined in Figure 2.7.*

Three-Valued Truth Mapping

$$T_3 : S \rightarrow \{T, F, \int\}$$

Form of S	Conditions	Truth Value
$\in$ Wff(F,V)		G(S)
$\exists x\, W$	$T_3(W\{x/d\}) = \mathbf{T}$ for some $d \in D_{Type(x)}$	T
	$T_3(W\{x/d\}) = \int$ for some $d \in D_{Type(x)}$ and $T_3(W\{x/d\}) \in \{\mathbf{F}, \int\}$ for all $d \in D_{Type(x)}$	$\int$
$\forall x\, W$	$T_3(W\{x/d\}) = \mathbf{T}$ for all $d \in D_{Type(x)}$	T
	$T_3(W\{x/d\}) = \int$ for some $d \in D_{Type(x)}$ and $T_3(W\{x/d\}) \in \{\mathbf{T}, \int\}$ for all $d \in D_{Type(x)}$	$\int$
A v B		$T_3(A)$ v $T_3(B)$
A & B		$T_3(A)$ & $T_3(B)$
$\neg$ A		$\neg\, T_3(A)$
	Otherwise	F

Figure 2.7. Three Valued Truth Mapping

The definition of $\mathcal{T}_\times$ is equivalent with $\wr$ replaced with $\times$. This is not surprising since a sentence's "real" truth value is just as unclear when it is overdetermined as when it is underdetermined.

Given this new definition of truth, the appropriate non-standard models can be defined.

Definition 2.15 *An* **underdetermined model** *of a sentence $e \in \mathcal{E}$ is an underdetermined interpretation $\mathcal{M}$ such that e is* true *in $\mathcal{M}$. An underdetermined model of a set of sentences is an underdetermined model of each sentence in that set.*

Notice that in the following definition *true* has been replaced with *not false*, allowing $\times$ to aid in the modelling. This is the only place where the distinction between $\wr$ and $\times$ appears.

Definition 2.16 *An* **overdetermined model** *of a sentence $e \in \mathcal{E}$ is an overdetermined interpretation $\mathcal{M}$ such that e is* not false *in $\mathcal{M}$. An overdetermined model of a set of sentences is an overdetermined model of each sentence in that set.*

2.4.4 Full (four-valued) models

It is also possible, and useful, to combine underdetermined and overdetermined models allowing all four truth values in interpretations.

Definition 2.17 *A* **full interpretation** *is the same as an interpretation except that $\mathcal{G} : Eqns(D) \rightarrow \{\mathbf{T}, \mathbf{F}, \wr, \times\}$. with the further restriction that when $e_1 =_\mathcal{E} e_2$ then $\mathcal{G}(e_1) = \times$ implies $\mathcal{G}(e_2) = \times$ and $\mathcal{G}(e_1) = \wr$ implies $\mathcal{G}(e_2) = \wr$ for $e_1, e_2 \in Eqns(D)$.*

There is a little difficulty in defining a four valued truth mapping. It is unclear how $\wr$ and $\times$ interact with respect to quantifiers. Suppose that S is a universally quantified sentence and that one instance of S is overdetermined and all the other instances are underdetermined. Then S can only be one of $\wr$ and $\times$, since choosing one of **T** and **F** is a jump to a conclusion. However, we choose $\times$.

Definition 2.18 *A* **four valued truth mapping** *$\mathcal{T}_{\wr\times}$ with respect to an interpretation I is defined where $\mathcal{T}_{\wr\times} : Wff(F,V) \rightarrow \{\mathbf{T},\mathbf{F}\}$ such that $\mathcal{T}_{\wr\times}(w) = \mathcal{T}_4(\mathcal{K}(w))$ for $w \in Wff(F,V)$. $\mathcal{T}_4$ is defined in Figure 2.8.*

Four-Valued Truth Mapping

$$T_2 : S \rightarrow \{T,F,\int,\times\}$$

Form of S	Conditions	Truth Value
$\cdot \in$ Eqns(D)		G(S)
$\exists x\, W$	$T_4(W\{x/d\}) = \mathbf{T}$ for some $d \in D_{Type(x)}$	T
	$T_4(W\{x/d\}) = \times$ for some $d \in D_{Type(x)}$ and $T_4(W\{x/d\}) \in \{\mathbf{F},\times,\int\}$ for all $d \in D_{Type(x)}$	$\times$
	$T_4(W\{x/d\}) = \int$ for some $d \in D_{Type(x)}$ and $T_4(W\{x/d\}) \in \{\mathbf{F},\int\}$ for all $d \in D_{Type(x)}$	$\int$
$\forall x\, W$	$T_4(W\{x/d\}) = \mathbf{T}$ for all $d \in D_{Type(x)}$	T
	$T_4(W\{x/d\}) = \times$ for some $d \in D_{Type(x)}$ and $T_4(W\{x/d\}) \in \{\mathbf{T},\int,\times\}$ for all $d \in D_{Type(x)}$	$\times$
	$T_4(W\{x/d\}) = \int$ for some $d \in D_{Type(x)}$ and $T_4(W\{x/d\}) \in \{\mathbf{T},\int\}$ for all $d \in D_{Type(x)}$	$\int$
A v B		$T_4(A)$ v $T_4(B)$
A & B		$T_4(A)$ & $T_4(B)$
$\neg$ A		$\neg\, T_4(A)$
	Otherwise	F

Figure 2.8. Four Value Truth Mapping

Definition 2.19 *A* **full model** *of a sentence $S \in \mathcal{E}$ is a full interpretation $\mathcal{M}$ such that S is true or overdetermined in $\mathcal{M}$. A full model of a set of sentences is a full model of each sentence in that set. This is denoted as $\mathcal{M} \models_{\mathsf{I}\times}$.*

The following proposition shows that the set of standard models of a particular theory is a subset of the set of nonstandard models. We say that all standard models are trivial examples of nonstandard models:

Proposition 2.1 *Let S be the set of all standard models, let U be all underdetermined models, let O be all overdetermined models, and let F be all full models of a negationless theory $\mathcal{E}$. Then $S \subseteq U \subseteq F$ and $S \subseteq O \subseteq F$.*

All proofs of propositions are found in Appendix A. This proposition depends on negationless theories. It is possible to define negation such that the negation of values $\mathbf{F}$, $\times$, and $\wr$ are all $\mathbf{T}$ instead of those defined in Figure 2.4. However, this would mean that sometimes $\neg\neg A \neq A$, for example, $\mathbf{F} = \neg\neg\wr \neq \wr$. Other mechanisms are required to avoid this problem.

2.4.5 Preferred models

There are statements that have inductive qualities that are nevertheless not inductive theorems. In the introduction to this chapter we introduced the term "ampliative theorem" for a statement that is not invalidated by data with unknown truth values, "ignorative theorem" for one that is not invalidated by data with contradictory truth values, and "prudent theorem" for one that is not invalidated by data with unknown or contradictory truth values. Expanding the ontology to include underdetermined and overdetermined truth values enables us to work with sentences whose truth values are not determined by a set of axioms, leading to definitions of nonstandard models. At first, it seems that nonstandard theorems can be characterized precisely using these new ways of interpreting sentences. For example, ampliative theorems could be defined as statements "not false" (and with some justification) in all underdetermined models of a set of axioms. But since all standard models are nonstandard models, all nonstandard theorems defined in this way will be inductive theorems, since they are "not false" in all standard models as well. In other words, only inductive theorems can be described using the machinery developed so far.

To make use of nonstandard models, it is necessary to develop a method of ignoring standard models when checking for nonstandard theoremhood (non-inductive ampliative, ignorative, or prudent theoremhood). In particular, underdetermined models that have "unknown" things assigned to **T** or **F** instead of ≀ are ignored. Also, we ignore overdetermined models that have consistent things assigned to × instead of the truth value that the axioms suggest (**T**, **F**, or ≀). Strictly speaking, these ignored models are not wrong; they simply do not follow the intuition behind using ≀ and × as truth values.

For example, suppose the axioms $\mathcal{Q}$ of a theory $\mathcal{E}$ only contains the equation $f(a) = b$ (where $Type(a) \not\equiv Type(b)$). $\mathcal{Q}$ has one overdetermined model that assigns this equation to **T** and another that assigns it to ×. (No model assigns it to **F** since an overdetermined interpretation must have every axiom assigned to something "not false"). Since there is nothing in $\mathcal{Q}$ to suggest that it is also **F**, its truth status is not overdetermined and we intuitively think of it as true. The second model does not capture the expected behaviour of the × truth value — so it should be ignored, leaving only the preferred first model.

Selection of desired models can be accomplished by using a type of nonmonotonic minimization called *preferencing*. Recall that minimizing **T** and **F** in standard models had not matched the intuition behind ampliation or ignoration (see Section 2.3.2). However, now the extra truth values ≀ and × have been introduced. These truth values can be minimized, in several different ways, allowing attention to be restricted to strictly nonstandard models when checking for theoremhood.

Preferences have been formalized by Shoham (1988) who used them to explain nonmonotonic reasoning. The following definition modifies his definition to include inductive modelling:

Definition 2.20 *Given a partial order on interpretations* $\sqsubset$ *called a* **preference relation**, *a* **preferred model** *of a set of sentences* $S \in \mathcal{E}$ *is an interpretation* $\mathcal{M}$ *such that each sentence in* S *is true in* $\mathcal{M}$ *and* $\neg\exists\mathcal{M}_1\ \mathcal{M}_1 \models S$ *and* $\mathcal{M} \sqsubset \mathcal{M}_1$. *This is denoted as* $\mathcal{M} \models_{\sqsubset}$.

The $\sqsubset$ relation must be defined based on the application. The next definitions give three preference relations that are appropriate for defining ampliative, ignorative and prudent theorems. The subscript of the equational truth mapping $\mathcal{G}$ in the following definitions refers to the interpretation (model) that defines it.

Definition 2.21 *An* **uncommitted preference relation** *is a partial order* $\sqsubset_l$ *on underdetermined interpretations.* $\mathcal{M}_0 \sqsubset_l \mathcal{M}_1$ *iff* $\forall e\ \mathcal{G}_1(e) = l \Rightarrow \mathcal{G}_0(e) = l$ *and* $\exists e\ \mathcal{G}_1(e) \in \mathbf{T},\mathbf{F} \Rightarrow \mathcal{G}_0(e) = l$. *Preferred underdetermined models under the uncommitted preference relation are called* **uncommitted-preferred models**. *We write* $M \models_l S$ *if M is an uncommitted-preferred model of S.*

The uncommitted preference relation says that a model $\mathcal{M}_0$ is preferred to another model $\mathcal{M}_1$ when everything underdetermined in $\mathcal{M}_1$ is also underdetermined in $\mathcal{M}_0$ and when something that is true or false in $\mathcal{M}_1$ is underdetermined in $\mathcal{M}_0$. This relation helps to prefer models with the least amount of commitment to **T** or **F**. An uncommitted-preferred model has no models preferred to it under the uncommitted preference relation.

Definition 2.22 *An* **avoidant preference relation** *is a partial order* $\sqsubset_\times$ *on overdetermined interpretations.* $\mathcal{M}_0 \sqsubset_\times \mathcal{M}_1$ *iff* $\forall e\ \mathcal{G}_1(e) = \mathbf{T} \Rightarrow \mathcal{G}_0(e) = \mathbf{T}$ *and* $\forall e\ \mathcal{G}_1(e) = \times \Rightarrow \mathcal{G}_0(e) \in \{\mathbf{T}, \times\}$ *and* $\exists e\ \mathcal{G}_1(e) = \times \Rightarrow \mathcal{G}_0(e) = \mathbf{T}$. *Preferred overdetermined models under the avoidant preference relation are called* **avoidant-preferred models**. *We write* $M \models_\times S$ *if M is an avoidant-preferred model of S.*

The avoidant preference relation says that a model is preferred when it has less $\times$ points (and has them replaced by **T**). By doing so, it helps to prefer models that avoid inconsistency. An avoidant-preferred model has no models preferred to it under the avoidant preference relation.

A forgetful preference could also be defined, maximizing the inconsistency; it is not clear what purpose it would serve. Also, a committed preference could also be defined, yet would be useless: it would leave only the standard models. But the uncommitted and avoidant relations can be combined for use with full models:

Definition 2.23 *An* **uncommitted avoidant preference relation** *is a partial order* $\sqsubset_{l\times}$ *on full interpretations.* $\mathcal{M}_0 \sqsubset_\times \mathcal{M}_1$ *iff* $\forall e\ \mathcal{G}_1(e) = \mathbf{T} \Rightarrow \mathcal{G}_0(e) \in \{\mathbf{T}, l\}$ *and* $\forall e\ \mathcal{G}_1(e) = \times \Rightarrow \mathcal{G}_0(e) \in \{\mathbf{T}, \times\}$ *and* $\forall e\ \mathcal{G}_1(e) = l \Rightarrow \mathcal{G}_0(e) = l$ *and* ($\exists e\ \mathcal{G}_1(e) \in \{\mathbf{T},\mathbf{F},\times\} \Rightarrow \mathcal{G}_0(e) = l$ *or* $\exists e\ \mathcal{G}_1(e) = \times \Rightarrow \mathcal{G}_0(e) = \mathbf{T}$). *Preferred full models under the avoidant preference relation are called* **uncommitted avoidant preferred models**. *We write* $M \models_{l\times} S$ *if M is an uncommitted-avoidant-preferred model of S.*

$$\mathcal{Q} = \begin{bmatrix} f(a) = & a \\ f(b) = & b \\ f(c) = & d \\ f(c) = & c \end{bmatrix}$$

$$\mathcal{M}_f : \left\{ \begin{array}{lll} f(a) = a & \times \\ f(b) = b & \mathbf{T} \\ f(c) = d & \times \\ f(c) = c & \times \\ f(d) = d & \mathbf{F} \end{array} \right\} \qquad \mathcal{M}_p : \left\{ \begin{array}{lll} f(a) = a & \mathbf{T} \\ f(b) = b & \mathbf{T} \\ f(c) = d & \times \\ f(c) = c & \times \\ f(d) = d & \wr \end{array} \right\}$$

Figure 2.9. A full model and an uncommitted avoidant preferred model

The uncommitted avoidant preference relation says that a model $\mathcal{M}_0$ is preferred to another model $\mathcal{M}_1$ when everything true in $\mathcal{M}_1$ is also true in $\mathcal{M}_0$, everything overdetermined in $\mathcal{M}_1$ is overdetermined or true in $\mathcal{M}_0$, everything underdetermined in $\mathcal{M}_1$ is also underdetermined in $\mathcal{M}_0$, and when $\mathcal{M}_0$ and $\mathcal{M}_1$ differ. In particular, they must differ in that either something that is not underdetermined in $\mathcal{M}_1$ is underdetermined in $\mathcal{M}_0$ or something that is overdetermined in $\mathcal{M}_1$ is true in $\mathcal{M}_0$. The preferred models under this relation, that is, those that have no models that are preferred to them, have the least amount of inconsistency and avoid committing to **T** or **F**.

Figure 2.9 gives a set of axioms $\mathcal{Q}$ and two models of it, $\mathcal{M}_f$ and $\mathcal{M}_p$. There are other full models but no other uncommitted avoidant preferred models. Note that all models must have $f(c) = d$ and $f(c) = c$ both overdetermined since they are inconsistent. $\mathcal{M}_p$ is preferred to $\mathcal{M}_f$ because $f(d) = d$ is underdetermined instead of anything else, $f(a) = a$ is true rather than overdetermined, and no other illegal changes to $\mathcal{M}_f$ are required to obtain $\mathcal{M}_p$.

2.4.6 Ampliative, ignorative and prudent theorems

This subsection defines ampliative, ignorative and prudent theorems, which embody the three other forms of induction besides summative induction. These are defined as particular inductivelike theorem classes, and vary only in the type of inductivelike modelling used. First, the particular types of inductive modelling that discriminate these theorem classes are defined.

Recall the concept of inductivelike modelling in Section 2.3.1. Each of the three new types of models developed in the last section, uncommitted, avoidant, and uncommitted-avoidant, when used instead of standard models in the definition of inductive modelling, along with biexemplar justification, introduce three new types of inductivelike modelling:

Definition 2.24 *An uncommitted-preferred model $\mathcal{M}$* **ampliatively models** *a sentence S if $\mathcal{M}$ inductively models $\mathcal{M}$ with biexemplar justification. $\mathcal{M} \mid\!\models_{\imath} S$ means $\mathcal{M}$ ampliatively models S.*

Definition 2.25 *An avoidant-preferred model $\mathcal{M}$* **ignoratively models** *a sentence S if $\mathcal{M}$ inductively models $\mathcal{M}$ with biexemplar justification. $\mathcal{M} \mid\!\models_{\times} S$ means $\mathcal{M}$ ignoratively models S.*

Definition 2.26 *An uncommitted-avoidant-preferred model $\mathcal{M}$* **prudently models** *a sentence S if $\mathcal{M}$ inductively models $\mathcal{M}$ with biexemplar justification. $\mathcal{M} \mid\!\models_{\imath\times} S$ means $\mathcal{M}$ prudently models S.*

Using these new modelling types, three major classes of theorems can be defined directly. They are summarized in Figure 2.12, along with the other types of theorems developed in this chapter.

Ampliative Theorems (AMP)

$$\mathcal{S} \in AMP(\mathcal{E}) \text{ iff } \forall \mathcal{M}\ \mathcal{M} \models_{\imath} \mathcal{Q} \Rightarrow \mathcal{M} \mid\!\models_{\imath} \mathcal{S}$$

In words, S is an ampliative theorem of $\mathcal{E}$ if and only if all uncommitted-preferred models of the rules of $\mathcal{E}$ ($\mathcal{Q}$) ampliatively model S.

Ignorative Theorems (IGN)

$$S \in IGN(\mathcal{E}) \text{ iff } \forall \mathcal{M}\ \mathcal{M} \models_{\times} \mathcal{Q} \Rightarrow \mathcal{M} \mid\!\models_{\times} S$$

In words, S is an ignorative theorem of $\mathcal{E}$ if and only if all avoidant-preferred models of $\mathcal{Q}$ ignoratively model S.

Prudent Theorems (PRU)

$$S \in PRU(\mathcal{E}) \text{ iff } \forall \mathcal{M}\ \mathcal{M} \models_{!\times} \mathcal{Q} \Rightarrow \mathcal{M} \mid\!\models_{!\times} S$$

In words, S is a prudent theorem of $\mathcal{E}$ if and only if all uncommitted-avoidant-preferred models of the rules $\mathcal{Q}$ prudently model S.

Each inductive theorem is an ampliative, ignorative, and prudent theorem. Also, each ampliative and ignorative theorem is a type of prudent theorem. These relationships between classes are formalized in the following proposition. Figure 2.1 illustrates the subset relations of this proposition, also noting that DED is a subclass of IND. First, a lemma used in the proposition's proof is presented:

Lemma 2.1 *Let S be the set of all standard models, let A be all avoidant preferred models, let K be all uncommitted preferred models, and let P be all uncommitted avoidant preferred models of a negationless theory $\mathcal{E}$. Then $S \subseteq A \subseteq P$ and $S \subseteq K \subseteq P$.*

Proposition 2.2 $IND(\mathcal{E}) \subseteq AMP(\mathcal{E}) \subseteq PRU(\mathcal{E})$ *and* $IND(\mathcal{E}) \subseteq IGN(\mathcal{E}) \subseteq PRU(\mathcal{E})$ *for a negationless theory $\mathcal{E}$.*

Consider the equation sets in Figure 2.10 and the formula $I \equiv \forall x$ *grandmother*$(x) =$ *mother*(*mother*(x)). It is clear that I is an inductive theorem of $\mathcal{Q}_s$, $I \in IND(\mathcal{Q}_s)$. By Proposition 2.2, $I \in AMP(\mathcal{Q}_s)$, $I \in IGN(\mathcal{Q}_s)$ and $I \in PRU(\mathcal{Q}_s)$ as well. For the other axiom sets, I is not an inductive theorem. $I \notin IND(\mathcal{Q}_l)$ because *grandmother*(*karla*) = *mother*(*mother*(*karla*)) is not true in all standard models (consider model $\mathcal{M}_1$ of Figure 2.11). $I \notin IND(\mathcal{Q}_{\times})$ and $I \notin IND(\mathcal{Q}_{l\times})$, because there are no standard models since *mother*(*joe*) = *karla* = *betty* is inconsistent.

$F = \{joe, betty, karla, mother/1, grandmother/1\}$

$$\mathcal{Q}_s = \begin{bmatrix} mother(joe) = & betty \\ mother(betty) = & karla \\ grandmother(joe) = & mother(betty) \\ grandmother(betty) = & mother(karla) \\ grandmother(karla) = & mother(mother(karla)) \end{bmatrix}$$

$$\mathcal{Q}_l = \begin{bmatrix} mother(joe) = & betty \\ mother(betty) = & karla \\ grandmother(joe) = & mother(betty) \\ grandmother(betty) = & mother(karla) \\ grandmother(karla) = & mother(mother(karla)) \end{bmatrix}$$

$$\mathcal{Q}_\times = \begin{bmatrix} mother(joe) = & karla \\ mother(joe) = & betty \\ mother(betty) = & karla \\ grandmother(joe) = & mother(betty) \\ grandmother(betty) = & mother(karla) \\ grandmother(karla) = & mother(mother(karla)) \end{bmatrix}$$

$$\mathcal{Q}_{l\times} = \begin{bmatrix} mother(joe) = & karla \\ mother(joe) = & betty \\ mother(betty) = & karla \\ grandmother(joe) = & mother(betty) \\ grandmother(betty) = & mother(karla) \end{bmatrix}$$

Figure 2.10. Equation sets that induce $\forall x$ *grandmother*(x) = *mother*(*mother*(x))

However, $I \in AMP(\mathcal{Q}_l)$, $I \in IGN(\mathcal{Q}_\times)$, and $I \in PRU(\mathcal{Q}_{l\times})$. For example, consider all uncommitted-preferred models of $\mathcal{Q}_l$. In fact there is only one, namely model $\mathcal{M}_1$ of Figure 2.11. In this model, no instances of I are assigned to false. Also, it is biexemplar justified by the instances $I\{joe/x\}$ and $I\{betty/x\}$. Thus $I \in AMP(\mathcal{Q}_l)$. Similar arguments can be made to show that $I \in IGN(\mathcal{Q}_\times)$ and $I \in PRU(\mathcal{Q}_{l\times})$.

$$\mathcal{M}_1 = \left(\begin{array}{ll} mother(joe) = betty & \mathbf{T} \\ mother(betty) = karla & \mathbf{T} \\ grandmother(joe) = mother(mother(joe)) & \mathbf{T} \\ grandmother(joe) = mother(betty) & \mathbf{T} \\ grandmother(joe) = karla & \mathbf{T} \\ grandmother(betty) = mother(mother(betty)) & \mathbf{T} \\ grandmother(betty) = mother(karla) & \mathbf{T} \\ grandmother(karla) = mother(mother(karla)) & \wr \\ otherwise & \wr \end{array} \right)$$

$$\mathcal{M}_2 = \left(\begin{array}{ll} mother(joe) = betty & \mathbf{T} \\ mother(betty) = karla & \mathbf{T} \\ grandmother(joe) = mother(mother(joe)) & \mathbf{T} \\ grandmother(joe) = mother(betty) & \mathbf{T} \\ grandmother(joe) = karla & \mathbf{T} \\ grandmother(betty) = mother(mother(betty)) & \mathbf{T} \\ grandmother(betty) = mother(karla) & \mathbf{T} \\ grandmother(karla) = mother(mother(karla)) & \mathbf{T} \\ joe = mother(karla) & \mathbf{T} \\ grandmother(karla) = betty & \mathbf{T} \\ grandmother(karla) = mother(joe) & \mathbf{T} \\ grandmother(betty) = joe & \mathbf{T} \\ otherwise & \mathbf{F} \end{array} \right)$$

Figure 2.11. Some sample models

2.5 Theorems for Machine Learning

The theorem classes IND, AMP, IGN, and PRU, can be used to characterize the theorems generated by machine learning systems. IND theorems are conservative and are used mainly to compress data reversibly. They do have predictive ability when the language is extended to include new function symbols of the sorts referred to in the theorems. IGN theorems are rather conservative, but can deal with noise. AMP theorems are risky in their inductive leaps, but cannot deal with noisy data. PRU theorems are also strongly predictive, but can also deal with noise. These four sets of theorems are comprehensive — the theorems discussed in the remainder of this thesis will be syntactically restricted types of them. In particular, we will

Name	*Modelling Type*	*Model Type*	*Justification*
DED (deductive)	$M \models s$	standard	total
IND (inductive)	$M \mid\!\models s$	standard	total
AMP (ampliative)	$M \mid\!\models_{\mathrm{l}} s$	uncommitted	biexemplar
IGN (ignorative)	$M \mid\!\models_{\times} s$	avoidant	biexemplar
PRU (prudent)	$M \mid\!\models_{\mathrm{l}\times} s$	uncommitted-avoidant	biexemplar

Figure 2.12. List of theorem types

concentrate on negationless conjunctive universal theorems.

Machine learning systems seem to emphasize universal theorems rather than existence theorems. This is not surprising, since existential theorems are less specific and contain less information than the data from which they are derived, while universal theorems describe specific properties that range over a complete database. However, existence theorems can be useful for zooming in on interesting sub-properties of a large scene, and it would sometimes be useful for a machine learning program to discover them. In general, all possible existence theorems can be generated from all deductive theorems. Universal theorems are the most often discussed of the two types of theorems. Kodratoff (1988) gives a good discussion of the use of existential and universal theorems in machine learning.

The type of description language most often used in machine learning systems is conjunctive, in which a set of rules is produced from a set of examples. These sets can be thought of as a conjunction of equations. Though some more advanced systems generate disjunctive descriptions, we disregard these.

Some machine learning systems, including a few developed in Chapter 5, first create subconcepts and use these subconcepts to create more complex concepts. A particularly clear example of this is the Marvin system (Sammut & Banerji, 1986). These higher level concepts can be seen as ampliative or prudent theorems, but not of the original theory:

Definition 2.27 *An* **extension** *to a theory $\mathcal{E}$ is a theory created by adding theorems of $\mathcal{E}$ to $\mathcal{E}$.* **Ampliative extensions** *augment $\mathcal{E}$ with ampliative theorems of $\mathcal{E}$ and* **prudent extensions**

$$\mathcal{Q} = \begin{bmatrix} mother(joe) = & betty \\ mother(betty) = & karla \\ grandmother(joe) = & mother(betty) \\ grandmother(betty) = & mother(karla) \\ grandmother(karla) = & mother(mother(karla)) \\ grandfather(bob) = & husband(mother(mother(bob))) \\ grandfather(karla) = & husband(mother(mother(karla))) \end{bmatrix}$$

$$\mathcal{Q}' = \mathcal{Q} \cup\ grandmother(x) = mother(mother(x))$$

Figure 2.13. An ampliative extension of a theory

augment $\mathcal{E}$ with prudent theorems of $\mathcal{E}$.

Using this definition, machine learning systems that work this way generate theorems of extensions of a given theory.

For example, consider the rules in Figure 2.13. $\mathcal{Q}'$ is an ampliative extension of $\mathcal{Q}$ since $grandmother(x) = mother(mother(x))$ is an ampliative theorem of $\mathcal{Q}$. It is biexemplar justified by instances with *joe* and *karla* and is underdetermined for all other instances. But then, $\forall x grandfather(x) = husband(grandmother(x))$ is an ampliative theorem of $\mathcal{Q}'$, but not of $\mathcal{Q}$ (since it is only justified by its *karla* instance). Some machine learning systems would generate such theorems of extensions as well as theorems of the original axioms.

In summary, this chapter has given a semantics for induction as used in machine learning. There were two common factors in all types of induction, a justification criterion and the avoidance of refutation. The varying factors were the particular justification used, the ontological status of propositions and the minimization technique. The status of propositions was extended from the standard true and false to include underdetermined (unsure) and overdetermined (inconsistent). The mechanics of managing the two new truth values to behave in concert with new truth definitions involved minimization techniques. These minimization techniques could be varied as well. This framework shows how to distinguish different types of induction. The remainder of the thesis will demonstrate some of its descriptive power.

Chapter 3

Theorem Proving Techniques

That which needs to be proved cannot be worth much. – Nietzsche

There are several methods of proving deductive and inductive theorems in first order logic. This chapter discusses the techniques that are commonly used for equational logic proofs. The first section describes unification and the next introduces term rewriting systems and reduction. The next section describes narrowing, a method of equation solving. The last section introduces a method of proving inductive theorems. All of these proof techniques will be reversed in Chapter 4 for the purpose of generating theorems. Figure 3.1 summarizes the important methods that are developed here.

3.1 Term Unification

In this section, the most basic concepts involved in theorem proving, namely substitution, application, and unification, are defined.

Definition 3.1 *A* **binding** *is a pair written v/t where t is a term, and v is a variable, and $v \not\equiv t$. Two bindings v_1/t_1 and v_2/t_2 are* disjoint *if $v_1 \not\equiv v_2$. A* **substitution** *is a set of disjoint bindings.*

Definition 3.2 *Suppose $\theta = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$. Then $Domain(\theta) = \{v_1, v_2, \ldots, v_n\}$ and $Range(\theta) = \{t_1, t_2, \ldots, t_n\}$. A* **ground substitution** *is a substitution θ such that $Vars(Range(\theta)) = \emptyset$. A ground substitution θ is a* **ground substitution of term** *t if $Vars(t) \subseteq Domain(\theta)$. A* **proper substitution** *is a substitution θ where $\exists t\ t \in Range(\theta)$ and v is a non-variable. Otherwise, θ is called a* **variable renaming**.

Some authors define $Range(\theta)$ to be the set of variables in $\{t_1, t_2, \ldots, t_n\}$ rather than the set of terms itself.

	Application	
tθ	Substitution θ applied to term *t*	*Section 3.1*
	Unification	
Mgu(s,t)	Most general unifier of terms *s* and *t*	*Section 3.1*
Mguterm(s,t)	Most general unificand of terms *s* and *t*	*Section 3.1*
Match(s,t)	Matching substitution of term *s* with term *t*	*Section 3.1*
	Reduction	
Nf(t,R)	Normal form of term *t* with respect to term rewriting system *R*	*Section 3.2*
s --> t	Term *s* reduces in one step to term *t*	*Section 3.2*
s --> t*	Term *s* reduces in any number of steps to term *t*	*Section 3.2*
	Equational Unification	
Narrow(e,R)	The set of all equations with a narrowing derivation from equation *e* with rules in *R*	*Section 3.3*
Csi(I,R)	Statement *I* is a cover set induction of *R*	*Section 3.4*

Figure 3.1. Generation operators discussed in this chapter

Definition 3.3 *A substitution* $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ *can be* **applied** *to a term* t *to produce a term* $t\theta$ *by replacing each* v_i *in* t *with the term* t_i*, for each binding* $(1 \leq i \leq n)$ *in* θ.

Definition 3.4 *Suppose* $\theta = \{v_1/s_1, v_2/s_2, ..., v_n/s_n\}$ *and* $\gamma = \{w_1/t_1, w_2/t_2, ..., w_m/t_m\}$. *Then the* **composition** *of substitutions is defined as:* $\theta \circ \gamma = \{v_1/s_1\gamma, v_2/s_2\gamma, ...v_n/s_n\gamma\} \cup \gamma - \{w_i/t_i \mid w_i \in Domain(\theta)\}$. For example, if $\theta_1 = \{v/f(x), w/f(z)\}$ and $\theta_2 = \{x/c, w/d\}$, then $\theta_1 \circ \theta_2 = \{v/f(c), w/f(z), x/c\}$.

Definition 3.5 *A substitution* θ *is said to be* **unifier** *of two terms* s *and* t *iff* $s\theta \equiv t\theta$. *We say that* s *and* t *are unifiable when a unifier exists for them. A* **most general unifier** *of* s *and* t*, denoted* $Mgu(s,t)$*, is a unifier* θ *of* s *and* t *such that for each unifier* σ *of* s *and* t*, there exists a* δ *such that* $\theta \circ \delta = \sigma$. *If* $\theta = Mgu(s,t)$*, then the term* $s\theta$ *is called the* **unificand** *of* s *and* t *and is denoted* $Mguterm(s,t)$. *A* **matching unifier** *of* s *with* t*, written* $Match(s,t)$*, is the most general unifier of* $Gnd(s)$ *and* t. *In other words,* $Match(s,t) = Mgu(Gnd(s),t)$. *We say* s **matches** t *when a matching unifier exists.*

For example, $Mgu(f(x,b), f(a,y)) = \{x/a, y/c\}$ and $Mguterm(f(x,b), f(a,y)) = f(a,b)$. Also, $Match(f(z,b), f(x,y)) = \{x/z, y/b\}$ and $Match(f(a,b), f(a,y)) = \{y/b\}$, but $f(z,b)$ and $f(x,x)$ do not match.

Unification is a central component of all first order theorem proving systems. Algorithms to compute most general unifiers are straightforward, and can be found in any book on basic logic in computer science (eg. Fitting, 1990; Genesereth and Nilsson, 1987) along with descriptions of many interesting properties of unification.

3.2 Term Rewriting Systems

The generation methods and semantics presented here are based on equational logic. Since *term rewriting systems* are very useful for proving theorems in this logic, they are prime candidates on which to build theorem generation methods. Huet and Oppen (1980) provide a good survey of term rewriting system theory and Avenhaus and Madlener (1990) have written an extensive introduction to the area. A brief introduction to relevant aspects of the subject is presented here.

First, recall that equations are pairs of terms, written as $s = t$. The equation $t = s$ is the same equation as $s = t$. **Rewrite rules** are oriented equations, that is, equations that have one term labelled as the left hand side and the other as the right hand side. For example, the equation $s = t$ can be oriented into the rewrite rule $s \rightarrow t$ where s is the left hand side, $Lhs(s \rightarrow t) = s$, and t is the right hand side, $Rhs(s \rightarrow t) = t$. Note that each (non-trivial, ie. $s \neq t$) equation can be oriented in two ways: $s = t$ can be oriented into both $s \rightarrow t$ and $t \rightarrow s$. Rewrite rules have the further restriction that $s \rightarrow t$ must satisfy $Vars(t) \subseteq Vars(s)$. So the equation $g(x) = f(x, y)$ can only be oriented into the rewrite rule $f(x, y) \rightarrow g(x)$.

A **term rewriting system** is a triple (F, V, R) where F is a set of functions, V a set of variables, and R a set of rewrite rules constructed from $Terms(F, V)$. A *first order typed term rewriting system* is a quadruple (T, F, V, R) where T is a typing function over terms (see Section 1.4). All rules $s \rightarrow t$ in a typed term rewriting system must satisfy $Type(s) = Type(t)$. These are called "first order" term rewriting systems, since variables are not allowed in type specifications. They provide no extra power over untyped term rewriting systems since a derivation (see below) in a typed term rewriting system is equivalent to one without the types. However, they will be used in this study since they enhance representational clarity and theorem proving efficiency.

Term rewriting systems are used to reduce terms to simpler forms. This is accomplished by a series of *reduction steps* using the *reduction operator*:

Reduction

$$\frac{t \qquad L_i \rightarrow R_i \in R}{t[u \leftarrow R_i]\theta}$$

where $u \in Pos(t)$ and
$\theta = Match(t/u, L_i))$

In this definition, some subterm of the term to be reduced is matched with the left hand side of some rule in the term rewriting system. Note that position u will never refer to a variable position unless L_i is a variable (this is allowable but rare in practice). The symbol

$$V = \{x, y\}$$

$$R = \begin{bmatrix} (1) & likes(x, supervisor(x)) \rightarrow true \\ (2) & supervisor(brent) \rightarrow ian \\ (3) & supervisor(debbie) \rightarrow brian \\ (4) & supervisor(brian) \rightarrow god \\ (5) & likes(god, ian) \rightarrow unknown \\ (6) & likes(brian, ian) \rightarrow true \\ (7) & likes(brent, supervisor(x)) \rightarrow true \end{bmatrix}$$

$$likes(brian, supervisor(brent)) \rightarrow^* true$$
since
$$likes(brian, supervisor(brent)) \rightarrow likes(brian, ian) \rightarrow true$$

Figure 3.2. Example of reduction.

$\rightarrow$ denotes a single reduction step, and the symbol $\rightarrow^*$ denotes the transitive closure of all known reduction steps. $s \rightarrow^* t$ means that s reduces to t after some number of reductions, or t is *derived from* s. A sequence of reduction steps $s \rightarrow s_1 \rightarrow s_2 ... \rightarrow t$ is a **derivation.**

Consider the term rewriting system R in Figure 3.2. The only possible reduction of the term *likes(supervisor(y),supervisor(brent))* is via rule (2) resulting in the term *likes(supervisor(y),ian).* Any other attempts at using a rule in R are mismatches. However, *likes(debbie,supervisor(debbie))* reduces via rule (1) with $\theta = \{x/debbie\}$ to $true$ and via rule (3) to *likes(debbie,brian).* The figure notes that *true* is derived from *likes(brian,supervisor(brent))* with a two step derivation *likes(brian,supervisor(brent))* $\rightarrow$ *likes(brian,ian)* $\rightarrow$ *true.*

A few more definitions end our introduction to term rewriting systems. A term t is in **normal form** (modulo R) if there is no rule in R that reduces t. A normal form of a term t modulo R is written $Nf(t, R)$. R is *confluent* if $s \rightarrow^* t_1$ and $s \rightarrow^* t_2$ implies that $\exists t\ t_1 \rightarrow^* t\ \wedge\ t_2 \rightarrow^* t$. In other words, all terms must have a unique normal form modulo R. R is *noetherian* (or terminating) if $\forall s \exists t\ t = Nf(s, R)$. For example, the rules $g \rightarrow g$ and $f(x) \rightarrow f(f(x))$ would produce infinite sequences of rewritings on the terms g and $f(a)$ respectively, and their inclusion in R would prevent it from being noetherian. However,

$f(f(x)) \rightarrow f(x)$ could be included. R is *interreduced* if $\forall\,(s \rightarrow t) \in R, t = Nf(t, R) \wedge s = Nf(s, R - \{s \rightarrow t\})$. In other words, all rules in R are composed of reduced terms. If R is confluent and noetherian, then it is *convergent.* If it is also interreduced, then it is **complete.**[1]

Consider the rewrite system R in Figure 3.2. It is not confluent and thus not complete. The completion of R would include:

$$\left[\begin{array}{l} likes(debbie, brian) \rightarrow true \\ likes(brent, ian) \rightarrow true \\ likes(brian, god) \rightarrow true \\ likes(brent, brian) \rightarrow true \\ likes(brent, god) \rightarrow true \end{array}\right]$$

Methods called *completion procedures* have been developed that attempt to make term rewriting systems complete (Knuth & Bendix, 1970). In this thesis, the rewrite systems that are used and generated are assumed to be complete. All results will be given under this assumption. To ensure that a term rewriting system is noetherian, completion procedures check to see that each rule is reducing by referring to a *reduction ordering*, a partial ordering on terms (Dershowitz, 1982).

Given a complete term rewriting system for $\mathcal{E}$, theorems in $DED(\mathcal{E})$ can be proven very simply. Using this method, $s = t$ (where s and t have no existential variables) is a theorem when $Nf(s, R) \equiv Nf(t, R)$. In other words, simply rewrite both sides of the equation to their normal forms, and if they are identical the equation is a theorem.

Because a universal Turing Machine can be represented as a set of equations, term rewriting systems are *Church-Turing equivalent.* This implies that they are representationally powerful but are sometimes undecidable. Fortunately, term rewriting systems that are confluent and noetherian are known to be decidable.

Term rewriting systems also can be used to prove theorems in the first order predicate calculus with equality, not just in the restricted equational logic that is used here. To do this, general formulae in the first order predicate calculus are reduced to a set of equations using

[1] Many authors use the term *canonical* to avoid overloading the word *complete.*

a deterministic procedure (Hsiang, 1986; Kapur & Narendran, 1985). This transformation suggests that equational logic is at least as powerful as the full first order predicate calculus with equality. Unfortunately, a drawback with this method is that the translation can be computationally intensive.

3.3 Equational unification by narrowing

Equational unification, or equation solving, is required for proving theorems in equational logic that involve existential variables. Given an equation $s = t$, equational unification finds a θ such that $s\theta =_{\mathcal{E}} t\theta$. In other words, it is unification after background theory is used to rewrite terms. *Narrowing* is a method of equation solving that requires a complete rewrite system for the equational theory. Other methods of equation solving are presented by Dershowitz and Sivakumar (1988). The narrowing operator, the main component of the narrowing method, is just a slight mutilation of the reduction operator:

Narrowing

$$\frac{t : L_e = R_e \qquad L_i \rightarrow R_i \in R}{t[u \leftarrow R_i]\theta}$$

where $u \in Pos(t)$ and
$\theta = Mgu(t/u, L_i)$ (ie. $t/u\theta = L_i\theta$)
t/u is a variable.

A *narrowing derivation* is a sequence of narrowing steps. Also, an equation is called a **narrowing** of another equation e with a set of rules R if it is produced by some narrowing derivation from e. We write the set of all narrowings of e with R as $Narrow(e, R)$.

The study of narrowing involves attempting to choose u intelligently. An overview of different narrowing strategies is given by Rety (1987).

To find solutions to an equation $Q_1x_1, ..., Q_nx_n\ s = t$ where the Q_i are either $\exists$ or $\forall$,

the universal variables are skolemized out (Lloyd, 1984).[2] This involves replacing universal variables with new, unique functions called skolem functions. In particular, a universal variable x_j is replaced with $sk_j(v_1, ..., v_m)$ where $v_1, ..., v_m$ are existential variables from among $x_1, ..., x_{j-1}$. Second, all quantifiers are dropped. Next, the resulting equation $s' = t'$ is made into a term $f(s', t')$ where f is a new function symbol. Then the narrowing operator is applied. If narrowing results in a term $f(r, r)$ where r is any term, then the equation is proven, and the substitution required to obtain $f(r, r)$ from $f(s', t')$ is a solution.

Consider the complete term rewriting system $\{f(x) \rightarrow c, g(a, x) \rightarrow x, g(b, x) \rightarrow c\}$. To prove the theorem $\forall y \exists z f(y) = g(z, c)$, the narrowing procedure is applied to its skolemized form: $f(sk) = g(z)$. It returns two solutions for z, namely z *is* a and z *is* b. This not only shows that the formula is a theorem but also computes possible values for the existential variables it contains.

Lemma 3.1 (Hullot, 1980) *Let $\mathcal{E}$ be an equational theory $\mathcal{E}$, R be a term rewriting system that is complete in $\mathcal{E}$, and s and t be two terms. The set of all solutions of a narrowing derivation of $s = t$ is complete.*

This lemma asserts that if there is a θ such that $s\theta = t\theta$ via R, then θ will be computed by using narrowing on $s = t$. It enables the proof of the following useful proposition:

Proposition 3.1 *Let $\mathcal{E}$ be an equational theory and R be a term rewriting system that is complete in $\mathcal{E}$. Then narrowing on R is a complete procedure for proving theorems in $DED(\mathcal{E})$.*

A corollary of this is:

Proposition 3.2 *Completion and narrowing together are complete for proving theorems in DED($\mathcal{E}$), for completable theories $\mathcal{E}$.*

This is true since a completion procedure produces a complete term rewriting system for a completable theory. These propositions will be used later to show the power of the generation operators.

[2] In proofs by refutation, existential variables are skolemized out. In proofs of equality, the universal variables are skolemized out.

3.4 Inductive Theorem Proving

Inductive theorem proving is difficult. Inductive theorems were defined in Chapter 2 as statements whose instances are deductive theorems. Thus, proving an inductive theorem amounts to enumerating all its instances and testing each for a deductive proof (perhaps using narrowing and completion). The problem with this is that there is often an infinite number of instances to test for deductive proof. Also, even if the number of instances is finite, it is likely to be large in reasonable applications.

Two approaches have been developed for this problem. *Inductionless induction* (Lankford, 1981) uses proof by consistency (Kapur & Musser, 1987) rather than proof by contradiction. It is a complex topic, and will not be discussed in this thesis. The remainder of this chapter describes *cover set induction*, the other principal approach.

One early method of making inductive theorem proving practical was to generate finite induction schemas made up of only a few premisses to be established. Mathematical induction is an example where induction schemas are used to prove theorems over infinite well-ordered domains. Its schemas are made up of base cases and generative cases. Burstall's structural induction method (Burstall, 1969) and the Boyer-Moore induction method (Boyer & Moore, 1979) are the most famous of these methods. Zhang, Kapur and Krishnamoorthy (1988) have recently developed the *cover set induction* approach, a comprehensive and yet usable induction method. The following definition of cover sets is paraphrased from them:

Definition 3.6 *A* **cover set** *of a sort T is the domain of an onto mapping Ψ from a finite set of terms of sort T to the power set of the ground constructor terms of sort T. Ψ is defined such that $\Psi(s) = T$ iff $\forall t \in T\ \exists \theta\ s\theta =_{\mathcal{E}} t$, $\Psi(t) \neq \emptyset$ and $\Psi(s) \cap \Psi(t) \neq \emptyset$ iff $s \equiv t$.*

A cover set of positive integers, for example, is $\{0, s(x)\}$, while one for lists is $\{nil, cons(x, y)\}$. Using cover sets simplifies the proof of inductive assertions. Instead of considering each instance of the assertion for a deductive proof, we need only consider each element of the cover set:

Definition 3.7 Cover Set Induction *Suppose that x is a (universal) variable of sort T in an inductive theorem $\mathcal{I}$ and M is a cover set of sort T. Then $\mathcal{I} \in IND(\mathcal{E})$ if for all $m \in M$, there exists a subterm t of sort T in m such that $\mathcal{I}\{t/x\} \Rightarrow \mathcal{I}\{m/x\} \in DED(\mathcal{E})$. $\mathcal{I}\{t/x\}$*

S = {integer}
V = {x}
F = {neg/1, 0, s/1, p/1}
F_c(integer) = {0,s,p}

$$Q = \begin{bmatrix} neg(0) = 0 \\ neg(p(x)) = s(neg(x)) \\ neg(s(x)) = p(neg(x)) \\ p(s(x)) = x \\ s(p(x)) = x \end{bmatrix}$$

Inductive Theorem: $\forall x\ neg(neg(x)) = x$

Figure 3.3. Negation theory

is called the inductive hypothesis.

If statement $\mathcal{I}$ is a cover set induction of R, we write $Csi(\mathcal{I}, R) = \mathrm{T}$. Although this definition is sufficient for our purposes, it is incomplete since it requires for soundness a more careful choice of the subterms of m (Zhang, 1988). Cover set induction is at least as powerful as Burstall's structural induction. Structural induction only uses a cover set developed from the constructors of a sort, whereas cover set induction can use any cover set.

For an example of cover set induction in action, consider the theory in Figure 3.3. The inductive theorem $\forall x\ neg(neg(x)) = x$ can be proven using cover set induction. The cover set $\{0, s(x), p(x)\}$ can be used. First $neg(neg(0)) = 0$ is shown to be a deductive theorem. $neg(neg(0)) = neg(0) = 0$ by the first axiom. Next, the inductive hypothesis $neg(neg(g)) = g$ is assumed (by adding it temporarily to the theory). Then $neg(neg(p(g))) = p(g)$ is shown to be true. Using the axioms at each step, $neg(neg(p(g))) = neg(s(neg(g))) = p(neg(neg(g)))$, and using the inductive hypothesis, $p(neg(neg(g))) = p(g)$. A similar argument is used for the successor (s) case. Using this cover set method avoids the infinite enumeration of $\{0, p(0), s(0), p(s(0)), p(p(0)), s(s(0)), ...\}$ that would be needed if the definition of induction from Chapter 2 were used directly.

It is sometimes tricky to choose cover sets that will work. Zhang (1988) shows that the definitions of function symbols in the conjecture can be used to generate cover sets automatically, under certain conditions.

Chapter 4

Theorem Generation Techniques

Computers are useless. They can only give you answers.
– Pablo Picasso

This chapter argues that (despite Picasso) computers can generate questions. Theorems, the central entities of this study, are the "questions" of logical languages and their generation is the principal operation of many machine learning programs. This chapter details methods that can be used to generate theorems. In doing so, it will help to answer the second problem of this thesis, the problem of computational induction. Recall that Chapter 3 investigated unification, term rewriting, equational unification and cover set induction. These proof methods are reversed in the first four sections of this chapter in order to obtain generation methods. Methods that involve specialization are discussed in Section 4.5. Among other benefits, this clarifies the relationship between generalization and specialization. Figure 4.1 summarizes most of the operators that are developed here.

4.1 Reverse Unification

Unification is an important operation in theorem proving due to its ability to detect commonalities between terms, and in a wider sense, between clauses and rules. The unificand of two terms is the most general description of what they have in common. Theorem provers use this commonality to choose appropriate rules and clauses to apply next in the chain of reasoning.

For inductive theorem generation and machine learning in general, a system must detect and resolve *differences* between examples, rather than just finding commonalities. Examples that unify, and thus have common subcases, convey redundant and thus important information. Examples that do not unify suggest that the respects in which they differ may be unimportant. The antiunification of terms locates and resolves differences between them.

	Antiapplication	
$t \uparrow \alpha$	Antisubstitution α antiapplied to term *t*	*Section 4.1.1*
	Antiunification	
Msa(s,t)	Most specific antiunifier of terms *s* and *t*	*Section 4.1.1*
Au(s,t)	Antiunificand of terms *s* and *t*	*Section 4.1.1*
Rau(r1,r2)	Antiunificand of rules *r1* and *r2*	*Section 4.1.2*
Irau(S,R)	Inductive rule antiunifications of a set of rules with respect to theory *R*	*Section 4.4.1*
	Expansion	
Exterms(t,R)	Set of expansions of a term *t*, or rule *r*, or its expansions, with rules in *R*.	*Section 4.2.1*
Exrules(r,R)		*Section 4.2.2*
Wids(r,R)	Set of widenings of rules *r1* and *r2* with theory *R*	*Section 4.3.2*
	Equational Antiunification	
Eau(r1,r2,R)	Equational antiunification of rules *r1* and *r2* with theory *R*	*Section 4.3.1*
An(r1,r2,R)	Antinarrowing of rules *r1* and *r2* with theory *R*	*Section 4.3.2*
Can(r1,r2,R)	Complete antinarrowing of rules *r1* and *r2* with theory *R*	*Section 4.3.3*
$Can^{i}(r1,r2,R)$	Inconsistent complete antinarrowing of rules *r1* and *r2* with theory *R*	*Section 4.3.3*
	Specialization	
Sp(t,V)	Cover set specialization of a term *t* with respect to variables *V*	*Section 4.5.1*
Nspec(S,R)	Naive cover set specialization of set of rules *S* with respect to theory *R*	*Section 4.5.2*
Spec(S,R)	Cover set specialization of set of rules *S* with respect to theory *R*	*Section 4.5.2*

Figure 4.1. Generation operators discussed in this chapter

Instead of focussing in on important information like unification does, antiunification filters out unimportant information.

This section describes the operation of antiunification. First, term antiunification is described precisely and then it is adapted for use with rules and equations.

4.1.1 Term antiunification

The term is the basic logical entity of equational theories. To unify two terms A_1 and A_2, a more specific term B and a substitution σ are found such that $A_1\sigma = A_2\sigma = B$. Similarly, to antiunify two terms B_1 and B_2, a more general term A and an antisubstitution α are found such that $B_1 \uparrow \alpha = A = B_2 \uparrow \alpha$. This subsection defines the unfamiliar terminology and symbols in this definition.

Antibindings and antisubstitions are defined similarly to their inverses, namely bindings and substitutions (Definition 3.1).

Definition 4.1 *An* **antibinding** *is a pair written P/v where P is a set of positions and v is a variable. Two antibindings P_1/v_1 and P_2/v_2 are* disjoint *if $P_1 \cap P_2 = \emptyset$. An* **antisubstitution** *is a set of disjoint antibindings.*

For example, $\{\{1.1, 2\}/x, \{1.2\}/y\}$ is an antisubstitution, while $\{\{1, 2\}/x, \{2, 3.1\}/y\}$ is not. The greek letters α and β are usually used for antisubstitutions.

Definition 4.2 *An antibinding $\{p_1, \ldots, p_m\}/v$ is* **relevant** *to a term t if $\{p_1, \ldots, p_m\} \subseteq Pos(t)$ and if $\forall i, j\ t/p_i = t/p_j$. That is, each position mentioned in the binding refers to a position in the term and all subterms of a term to which a relevant antibinding refers must be equivalent. An antisubstitution is relevant to a term t if all of its component bindings are relevant to t.*

For example, the antibinding $\{\{1.1, 2\}/x, \{1.2\}/y\}$ is relevant to $f(g(a, a), a)$, but $\{\{1.3\}/x\}$ is not relevant, since it refers to a nonexistent position in the term, and $\{\{1\}/x, \{2\}/y\}$ is not relevant since the term at position 1, $g(a, a)$, is not the same as the term at position 2, a.

Application of substitutions (Definition 3.2) is the fundamental operation in unification

theory; here it is antiapplication:

Definition 4.3 *An antisubstitution $\alpha = \{P_1/v_1, ..., P_n/v_n\}$ can be* **antiapplied** *to a term s producing a term $s \uparrow \alpha$ that has all positions of s in P_i replaced with v_i. If α is not relevant to s then $s \uparrow \alpha$ is undefined. Otherwise, $s \uparrow \alpha$ is defined recursively*

$$s \uparrow \alpha = \begin{cases} s & \text{if } \alpha = \emptyset \\ s[p_1 \leftarrow v_1]...[p_m \leftarrow v_1] \uparrow \{P_2/v_2, ..., P_n/v_n\} & \text{if } \alpha = \{P_1/v_1, ..., P_n/v_n\} \\ & \text{and } P_1 = \{p_1, ..., p_m\} \end{cases}$$

For example, suppose $\alpha = \{\{1.1, 2\}/x, \{1.2\}/y\}$ and $s = f(g(a, h(b)), c)$. Then $s \uparrow \alpha = f(g(a, h(b)), c)[1.1 \leftarrow x][2 \leftarrow x] \uparrow \{\{1.2\}/y\} = f(g(x, h(b)), c)[2 \leftarrow x] \uparrow \{\{1.2\}/y\} = f(g(x, h(b)), x) \uparrow \{\{1.2\}/y\} = f(g(x, h(b)), x)[1.2 \leftarrow y] = f(g(x, y), x)$. Also, $h(b) \uparrow \{\{1.3\}/x\}$ is undefined.

Recall from Definition 3.5 that a substitution θ such that $s\theta = t\theta$ is called a unifier of s and t. The definition of antiunifier is similar:

Definition 4.4 *Suppose s and t are terms. An antisubstitution α is called an* **antiunifier** *if $s \uparrow \alpha = t \uparrow \alpha$ and α is relevant to s and t. The term $s \uparrow \alpha$ is the* **antiunificand** *of s and t with respect to α when α is an antiunifier of s and t. A non-variable antiunificand is called a* **proper antiunificand.**

For example, an antiunificand of $f(g(a, h(b)), a)$ and $f(g(b, b), h(b))$ is $f(x, y)$ using the antiunifier $\{\{1\}/x, \{2\}/y\}$; another is $f(g(x, y), x)$ using $\{\{1.1, 2\}/x, \{1.2\}/y\}$.

Unification theory speaks of most general unifiers that are produced by unification algorithms. A similar notion in antiunification theory is useful.

Definition 4.5 *Suppose α and β are antiunifiers of the terms s and t. If there is a proper θ such that $s \uparrow \alpha = (s \uparrow \beta)\theta$, then α is* **more specific than** *(or equally specific to) β. We write $\alpha \leq \beta$.*

For example, consider the antiunifiers $\beta = \{\{1\}/x, \{2\}/y\}$ and $\alpha = \{\{1.1, 2\}/z, \{1.2\}/w\}$ of the terms $f(g(a, h(b)), a)$, call it s, and $f(g(b, b), h(b))$. Then $s \uparrow \beta = f(x, y)$ and $s \uparrow \alpha = f(g(z, w), z)$. Since $s \uparrow \alpha = f(x, y)\{x/g(z, w), y/z\} = (s \uparrow \beta)\{x/g(z, w), y/z\}$, then $\alpha \leq \beta$.

Definition 4.6 *If s and t are terms, then an antiunifier α of s and t such that for each antiunifier β of s and t, $\alpha \leq \beta$, is called a* **most specific antiunifier** *of s and t and is denoted $Msa(s,t)$.*

In general, a given pair of terms might have several most specific antiunifiers. However, they only differ by the names of the variables used. Figure 4.2(i) and (ii) give two simple cases of most specific antiunifiers. In Figure 4.2(iii) the antiunificand is $f(x,x)$, not the more general $f(x,y)$.

The *antiunification operator* is central to this thesis and is used to compute the antiunificands of most specific antiunifiers:

Antiunification

$$\frac{s \quad t}{s \uparrow Msa(s,t)}$$

where s, t are terms.

The term $s \uparrow Msa(s,t)$ in this definition is called the *antiunificand of s and t.* The implementation of this operator requires the computation of the most specific antiunifier of s and t. The following is a declarative presentation of the straightforward algorithm to do this:

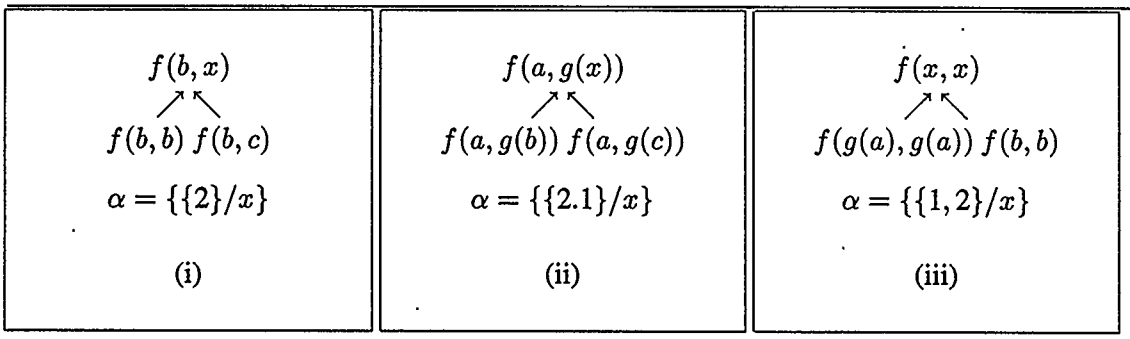


Figure 4.2. Most specific antiunifiers.

Antiunification Algorithm

$$Au(s,t) = \begin{cases} s & \text{if } s \equiv t \\ f(Au(s_1,t_1), Au(s_2,t_2) \ldots Au(s_n,t_n)) & \text{if } s \equiv f(s_1, s_2 \ldots s_n) \text{ and } t \equiv f(t_1, t_2 \ldots t_n) \\ v_{s,t} \text{ (a variable)} & \text{otherwise} \end{cases}$$

This algorithm produces the antiunificand $Au(s,t) = s \uparrow Msa(s,t)$, and its correctness is proven by Lassez, Maher and Marriott (1987). In the last line, $Au(s,t)$ always generates a unique but particular variable for each *pair* of terms, not a unique variable for each invocation of the *Au* function. For example, consider the terms in Figure 4.2 (iii). $Au(f(g(a),g(a)),f(b,b)) = f(Au(g(a),b),Au(g(a),b)) = f(x,x)$ where $v_{g(a),b} = x$.

The following proposition shows that $Msa(s,t)$ can be computed easily using an algorithm to compute most general unifiers (*Mgu*).

Proposition 4.1 *Suppose s and t are terms and a is the antiunification of s and t, that is,* $a = Au(s,t)$. *If* $Mgu(a,s) = \{v_1/s_1, v_2/s_2, ..., v_m/s_m\}$ *then* $\beta = Msa(s,t) = \{ Pos(v_1,a)/v_1, \ldots, Pos(v_m,a)/v_m, \}$

For example, suppose that the antiunification algorithm produces the term $a = f(g(x),h(y,x))$ from the terms $s = f(g(c),h(c,c))$ and $t = f(g(b),h(d,b))$. Then $Mgu(a,s) = \{x/c, y/c\}$ and $Mgu(a,t) = \{x/b, y/d\}$. By Proposition 4.1, $Msa(s,t) = \{\{1.1, 2.2\}/x, \{2.1\}/y\}$.

Both *Msa* and *Au* have been defined on pairs of terms. It is also useful to define them on sets of terms:

Proposition 4.2 *Define* $Au(S)$ *to be the* **antiunificand of a set of terms** S *computed by* $Au(S) = Au(s_1, Au(s_2, \ldots, Au(s_{n-1}, s_n)))$ *for* $s_i \neq s_j, 1 \leq i,j \leq n$ *where* $|S| = n$. $Au(S)$ *is independent of the order of application of the pairwise Au function. Similarly, define* $Msa(S)$ *to be the* **most specific antiunifier of a set of terms** S *computed by* $Msa(S) = Msa(s_1, Msa(s_2, \ldots, Msa(s_{n-1}, s_n)))$. $Msa(S)$ *is also independent of the order of application of the pairwise Msa function.*

The antiunification algorithm and most specific antiunifiers play a central role in the study of induction and ampliation, much like the role that unification and most general unifiers play in theorem proving. Although by themselves rather uninteresting, they are basic building blocks for more complicated operators. The remainder of this chapter details how term antiunification can be put to use.

4.1.2 Rule and equation antiunification

The next larger entity in equational logic is the equation or rule. Consider the rules $\{dog(barney) \rightarrow true, dog(rolf) \rightarrow true\}$. Their natural common generalization is that *all objects in this domain are dogs*, or $\forall x\ dog(x) \rightarrow true$. This more general rule can be produced by antiunifying the left hand sides, and leaving the right hand side unchanged. However, this does not work in general. Consider the rules $\{ dog(barney) \rightarrow happy(barney), dog(rolf) \rightarrow happy(rolf)\}$. The natural generalization is that *all dogs are happy*, or $\forall x\ dog(x) \rightarrow happy(x)$. This is not simply the antiunification of the left hand sides and right hand sides separately. The left hand sides and right hand sides must be antiunified simultaneously and so each rule is treated as a term and both terms are antiunified.

Rule Antiunification

$$\frac{L1 \rightarrow R1 \qquad L2 \rightarrow R2}{\mathcal{A}/1 \rightarrow \mathcal{A}/2}$$

where $\mathcal{A} = Au(f(L1, R1), f(L2, R2))$
and f is a new function symbol
and $Vars(\mathcal{A}/2) \subseteq Vars(\mathcal{A}/1)$

Rule antiunification is denoted by $Rau(r1, r2)$. The final condition prevents us from creating a rule that is not variable reducing (in which case, it would not be a rule in a term rewriting system). We sometimes write $Termify(r)$ as a shorthand for converting a rule into a term and $Makerule(t)$ for converting it back to a rule. Then $Rau(r1, r2) = Makerule(Au(Termify(r1), Termify(r2)))$.

Equation antiunification is barely more difficult. Since equations are unoriented, two rules for each equation need to be antiunified with each of the two from the other equation. Orienting e_1 and e_2 obtains (at most) the rules $e_{11}, e_{12}, e_{21}, e_{22}$. Next we compute $\{Rau(e_{11}, e_{21}), Rau(e_{11}, e_{22}), Rau(e_{12}, e_{21}), Rau(e_{12}, e_{22})\}$ and convert the resulting rules back into equations.

4.2 Reverse Term Rewriting

The use of unrestricted rules with rule antiunification is not powerful enough for many applications. Many machine learning algorithms use background theories to assist their inductions. Often, problems are specified by defining not only rules, but also a background theory. This section shows how a theory can be used to generate deductive theorems. In Section 4.3, these operators will be used as components of more complex operators that enable induction of various types.

4.2.1 Term expansion

Consider the background theory $dad(barney) \rightarrow harry$. A good generalization of $dog(harry)$ and $dog(dad(rolf))$ with respect to this rule would be $dog(dad(x))$ rather than $dog(x)$. This is because *fatherhood* might have some bearing upon what makes the two dogs similar. To obtain $dog(dad(x))$, $dog(harry)$ needs to be expanded to $dog(dad(barney))$ using the background theory before it is antiunified with $dog(dad(rolf))$.

Expansion is the dual of rewriting — instead of rewriting over a set of rules R, rewriting is done over R with its rules reversed. This is another instance where a proof technique is used in reverse for generation purposes. The *expansion* operator is defined as:

Expansion

t (a term)

$L_1 \rightarrow R_1$

$t[u \leftarrow L_1]\theta$

where $u \in Pos(t)$ and

$\theta = Match(t/u, R_1)$

The function $Exterms(t, R)$ denotes the set of all expansions (and expansions of expansions, and so on) of a term t with respect to a term rewriting system R. Each member of this set is called an **exterm**. Figure 4.3 demonstrates the application of the expansion operator. Each arrow in the middle of the figure indicates an expansion step. For example, expansion of the term $f(c, x)$ with respect to the rule set R yields the exterm $f(g(a), x)$ using the third rule in R, $g(a) \rightarrow c$. The figure is not a generalization lattice since, for example, $f(g(a), p(p(x)))$ is less general than $f(g(a), x)$ even though it is higher up in the expansion graph.

When the right hand side of a rule is a variable it can be used to expand any subterm of a term. Also, if $s \rightarrow t$ and $Vars(t) \subset Vars(s)$ (proper containment), expansion will introduce new variables. This gives a new way in which variables can be introduced into learning systems other than by the standard method of replacing constants with variables.

4.2.2 Rule expansion

Rules, not terms, are the objects of this study. The *rule expansion* operator is simply expansion done on rules instead of terms:

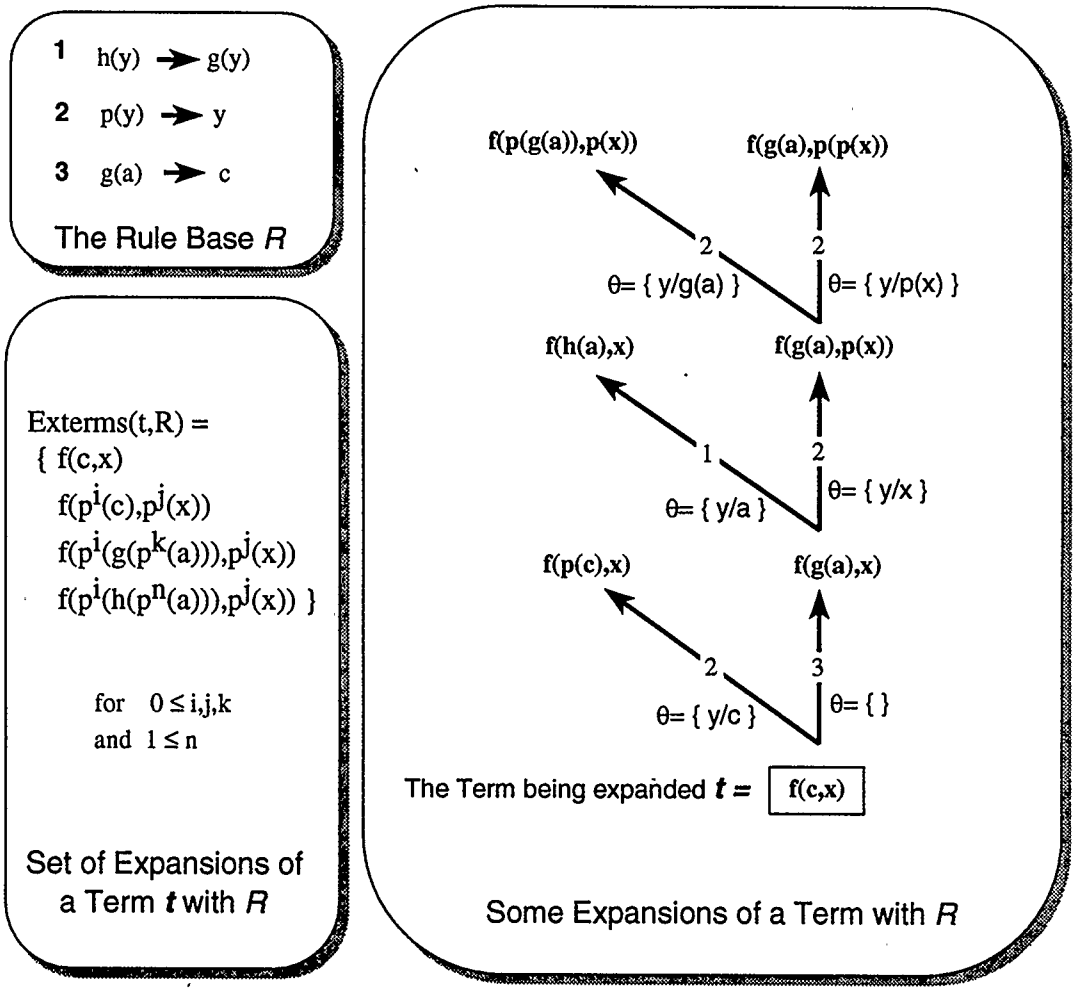


Figure 4.3. Expansion is rewriting with rules reversed.

$$\left[\begin{array}{ll}(1) & app(a.nil, nil) \rightarrow a.nil \\ (2) & app(nil, x) \rightarrow x\end{array}\right] \quad \left[\begin{array}{ll}(RW) & app(a.nil, nil) \rightarrow a.app(nil, nil) \\ (W) & app(a.app(nil, nil), nil) \rightarrow a.nil\end{array}\right]$$

Figure 4.4. Rule expansion example.

Rule Expansion

$$\frac{L_1 \rightarrow R_1 \qquad L_2 \rightarrow R_2}{r/1 \rightarrow r/2}$$

where $t = f(L_1, R_1)$
$r = t[u \leftarrow L_2]\theta$
$u \in Pos(t)$
$\theta = Match(t/u, R_2)$
$Vars(r/2) \subseteq Vars(r/1)$

The set of all rule expansions of a rule is called its set of **exrules** and is denoted $Exrules(r, R)$. A single expansion step is denoted $Ex(r_1, r_2)$. A **right expansion** is an operator that expands only the right hand side of a rule and is defined by replacing $u \in Pos(t)$ with $u \in Pos(R_1)$ in the above definition. Figure 4.4 shows a rule (1) that can be right-expanded by (2) to obtain *(RW)*. Regular expansion would also produce *(W)* by expanding the leftmost *nil* in (1) with rule (2). Note that the final condition in the definition of rule expansion is required because of the possible introduction of variables by expansion.[1]

Expansion can be used to generate deductive theorems. Everything created by expansion is a deductive theorem of the rule expanded, if the rule used to expand is a deductive theorem. In other words, it is *sound*. Also, all universal[2] negationless conjunctive deductive theorems of a theory can be generated with the expansion operator applied to all tautologies (rules of the form $t \rightarrow t$) using the axioms of the theory. In other words, it is a *complete* operator. We end this section by presenting these results.

[1] When reduction orderings are used for completion procedures (Section 3.2), this condition can be strengthened to $r/2 <_{rpos} r/1$ or $r/2 <_{kbo} r/1$ (etc.) to ensure that the rule is still reducing.

[2] An existential introduction operator can be used to generate existentially-quantified rules. It amounts to replacing subterms with existential variables.

Proposition 4.3 Expansion Soundness *Suppose that R is a complete term rewriting system for the equational theory $\mathcal{E}$. If $e \in DED(\mathcal{E})$ and $r \in R$, then $Ex(e, r) \in DED(\mathcal{E})$.*

This theorem states that rule expansions always produce rules whose corresponding equations are logical consequences of R and r. We can rest assured that using this operator will not introduce errors.

Proposition 4.4 Expansion Completeness *Suppose that R is a complete term rewriting system for the equational theory $\mathcal{E}$. Then $(a = b) \in DED(\mathcal{E})$ implies that $\exists t\ (a = b) \in Exrules(t \rightarrow t, R)$.*

This theorem implies that all logical consequences of a theory can be generated by enumerating all the terms in $Terms(\mathcal{E})$. Fortunately, all we wish to do is to generate theorems given particular terms as starting points. If we choose a t that would be used in a proof for $a = b$, this theorem shows us that we can be sure that $a = b$ will be generated.

4.3 Reverse Equational Unification

In Section 3.3, we discussed how terms can be unified with respected to a background theory. The use of background theory is also useful for theorem generation. In this section, we will develop the E-antiunification operator, the antinarrowing operator, and the complete antinarrowing operator, which generate rules through a combination of antiunification and the application of background theory.

First, we distinguish between two types of background theory, called explicit and implicit background theory.

Definition 4.7 *An* **explicit background theory** *is a set of rules specified at the outset of a generation problem. An* **implicit background theory** *is a set of rules derived from a set of examples using antiunification-based generation operators (not expansion).*

The explicit background theory might be the examples themselves or additionally include rules that are not to be generalized. The distinction between explicit background theory and examples is only an efficiency issue, not a logical one, and is irrelevant here. What is important is that generation operators that solely use the explicit background theory are

not as powerful as those that additionally use implicit background theory. Using the logical framework of Chapter 2, both the implicit and explicit background theory comprise the union of all extensions of the original theory, while the explicit background theory is exactly the original theory.

Section 4.1 discussed the rule antiunification operator as a method of generalizing rules. Section 4.2 discussed the expansion operator that enables the use of a background theory. In Section 4.3.1, these operators are first combined to obtain a method of generalization using background theories called "E-antiunification" that is sufficient for use with an explicit background theory. A more general operator called antinarrowing is developed in Section 4.3.2 that also allows the use of implicit background theory. Section 4.3.3 develops the complete antinarrowing operator to extend the theoretical power of the regular antinarrowing operator.

4.3.1 Antiunification with explicit background theory

The *E-antiunification operator*[3] is an operator that combines expansion and antiunification. In doing so, it allows us to use an explicit background theory, R, to generate an equivalence between two rules r_1 and r_2:

E-Antiunification

R	(a rewrite system)
r_1	(a rule)
r_2	(a rule)

$Rau(p, q)$

where $p \in Exrules(r_1, R), q \in Exrules(r_2, R)$
and $Valid(a, R)$

[3]The name *E-antiunification* is a modification of the term *E-unification* that is used for a unification operator used by some in the theorem proving community (Lassez, Maher & Marriott, 1987). The "E" signifies (anti)unification with respect to an equational theory.

The set of all rules that the E-antiunification operator generates is denoted $Eau(r_1, r_2, R)$. Notice the use of *Valid* in the definition. It is defined as

$$Valid(r, R) = Goodrule(r, R) \wedge Consistent(r, R)$$

Goodrule is a boolean test to see if a rule is variable-reducing:

$$Goodrule(r, R) = Vars(r/2) \subseteq Vars(r/1)$$

Goodrule might be augmented with other restrictions on the type of rules that one might wish to allow, such as the use of reduction orderings (Dershowitz, 1982). The *Consistent* test checks to see that the rule is consistent with the given set of rules. When *Valid* is replaced with *Goodrule* in the definition of E-antiunification, the *inconsistent E-antiunification* operator is derived and is denoted $Eau^i(r_1, r_2, R)$.

To illustrate the E-antiunification operator, let r_1 be $g(c) \rightarrow f(c)$ and let r_2 be $g(a) \rightarrow b$. Then $Rau(r_1, r_2)$ is undefined since their antiunification $g(x) \rightarrow y$ is not a rewrite rule (it has a variable in the right hand side that is not in the left). Now suppose we allow the use of the single-rule background theory $R = \{f(a) \rightarrow b\}$. Then an exrule of r_2, namely $g(a) \rightarrow f(a)$, is made possible by expanding the b in r_2 with the rule in R. This exrule, when antiunified with r_1 (a trivial exrule of itself), produces $g(x) \rightarrow f(x)$. Since there are no other expansions or antiunifications possible, $Eau(r_1, r_2, R)$ is $\{g(x) \rightarrow f(x)\}$.

Note that if $Exterms(t, R)$ and $Exterms(s, R)$ are finite, then $Eau(s, t, R)$ is finite, as in the example just given. When $Exterms(t_i, R)$ is infinite, $Eau(s, t, R)$ is often infinite. However, it might be finite like in the single rule theory $\{f(s(s(x))) \rightarrow f(s(x))\}$. Here $Eau(f(b), f(s(0)), R)$ is simply $\{f(x)\}$.

The E-antinarrowing operator does not work for some problems. Consider trying to generate the ampliative theorem $T \equiv list(cons(x, y)) \rightarrow list(y)$ from this theory:

$$R: \left[\begin{array}{l} list(nil) \rightarrow true \\ list(cons(a, nil)) \rightarrow true \\ list(cons(b, nil)) \rightarrow true \\ list(cons(c, cons(d, nil))) \rightarrow true \end{array}\right]$$

The theorem is the antiunification of $r_1 \equiv list(cons(a, nil)) \rightarrow list(nil)$ and $r_2 \equiv list(cons(c, cons(d, nil))) \rightarrow list(cons(d, nil))$. In order for the E-antiunification operator to generate T, both of these would have to be exrules of some pair of rules in R. The former, r_1, is an exrule since it is the second rule in R expanded with the first rule in R. However, r_2 is not an exrule, though it could have been an expansion of the fourth rule in R, if only the rule $B \equiv list(cons(x, nil)) \rightarrow true$ was part of the explicit background theory. However, B is part of the *implicit* background theory, since it is the antiunification of the second and third rules of R. But the E-antiunification operator only has the ability to use explicit background theory.

4.3.2 Antinarrowing

Another operator can be defined that takes into account both explicit and implicit background theory. In Section 3.3, narrowing was defined as a method of proving an equality by using a combination of unification and of rewriting. By using a combination of rule antiunification and a type of rule expansion called *widening*, we define the *antinarrowing operator*:

Antinarrowing

R (a rewrite system)
r_1 : $L1 \rightarrow R1$
r_2 : $L2 \rightarrow R2$

$\{a | a = Rau(p, q)$
where $p \in Wids(r_1, R), q \in Wids(r_2, R), Valid(a, R)\ \}$

This definition of the antinarrowing operator is similar to that of E-antiunification except that *Exrules* is replaced with *Wids*, described below. Each rule produced with the antinarrowing operator is called an **antinarrowing**. The set of all antinarrowings of two rules r_1 and r_2 is denoted $An(r1, r2, R)$ and the set of all antinarrowings of all pairs of rules in a rewrite system is denoted $An(R)$. Like E-antiunification, antinarrowing generates rules that comprise the implicit background theory.

To understand the antinarrowing operator, we first need to understand widening, and in particular, the function *Wid*. The expansion operator from Section 4.2.2 chooses a rule from an explicit background theory R and applies it in reverse to a given rule r. The **widening operator** is an expansion operator that may apply rules from *both* of the explicit and implicit background theories to obtain a new rule. In other words, it can expand a rule r with the E-antiunifications, or more generally, the antinarrowings of R. We denote the application of the widening operator as $Wid(w, a)$ where w is any rule (usually one produced by previous widenings), and a is any antinarrowing. Any rule that is produced with a series of applications (possibly zero) of the widening operator on a rule r is called a **widening** of the rule r. We denote the set of all widenings of a rule r with respect to a set of rules R as $Wids(r, R)$ and the set of all widenings of R as $Wids(R)$.

Consider the *list* example of the previous section. The rule $B \equiv list(cons(x, nil)) \rightarrow true$ is an antinarrowing of the second and third rules. Then, a widening of the fourth rule using B is $list(cons(c, cons(d, nil))) \rightarrow list(cons(d, nil))$. Another widening, $list(cons(a, nil)) \rightarrow list(nil)$ is the expansion of the first rule with the second rule. (Note that this widening is an exrule, unlike the first widening). When both of these widenings are antiunified, the antinarrowing $list(cons(x, y)) \rightarrow list(y)$ is produced.

Computing $Wids(r, R)$ is not straightforward, since it is defined in terms of antinarrowings, which are in turn defined by widenings that might be in $Wids(r, R)$. In other words, the definitions of the set of all widenings and the set of all antinarrowings are mutually dependent. Nevertheless, we can compute the set of all widenings as follows:

$$Wids(r, R) = \{x |\ x = r \vee x = Wid(w, a),$$
$$\text{where } a \in An(R), w \in Wids(r, R), |Wid(w, a)| > |w|\ \}$$

The $|w|$ symbol here denotes the widening size, or the number of applications of the widening operator required to make w from the given set of rules, R. The mutually dependent definitions of widening size and antinarrowing size follow:

Definition 4.8 *Given a rule set R, the* **widening size** *of $Wid(w, a)$ is n, denoted $|Wid(w, a)| = n$, iff $|w| \leq n-1$ and a is of antinarrowing size $n-1$ and $n = max(|w|, |a|)+1$. A widening r is of size 0 iff $r \in R$.*

Definition 4.9 *Given a rule set R, the* **antinarrowing size** *of* $a = Rau(w_1, w_2)$ *is* n*, denoted* $|a| = n$*, iff* $n = max(|w_1|, |w_2|)$*. An antinarrowing* r *is of size 0 iff* $r \in Rau(R)$*.*

Given these definitions, the generation of widenings and antinarrowings can be done in an orderly fashion starting from size 0 rules. For instance, in the *list* example the rules in R are widenings of size 0 and antinarrowings of size 0. The other antinarrowings of size 0 are the antiunifications of the rules in R:

$$list(x) \rightarrow true$$
$$list(cons(x, nil)) \rightarrow true$$
$$list(cons(x, y)) \rightarrow true$$

Size 1 widenings can be produced from the four size 0 widenings and the seven size 0 antinarrowings. There are 20 of these. All 20 of these size 1 widenings can then be antiunified to obtain the size 1 antinarrowings. The size 1 antinarrowings can be used on the size 1 widenings to obtain size 2 widenings and then size 2 antinarrowings and so on. The ampliative theorem we intended to produce, $list(cons(x, y)) \rightarrow list(y)$, is a size 1 antinarrowing of the examples.

Another operator similar to antinarrowing is called **inconsistent antinarrowing** and is defined by replacing *Valid* with *Goodrule* in the definition of antinarrowing. This allows the inconsistent antinarrowing operator to produce rules that may conflict with noise or rules that are exceptions to the general theory behind them. Some uses of this operator, denoted $An^i(R)$, are shown in Chapter 5.

Finally, it is useful to define *naive* versions of the antinarrowing operator and the inconsistent antinarrowing operator that differ from the regular ones in that they are restricted to expansion (widening) on the right hand side of a rule, or what we called *right expansion.* We call these **naive antinarrowing operators** because only a subset of antinarrowings will be produced if this policy is implemented. Since the programs that are developed in Chapter 5 use these naive operators only for efficiency reasons, we will only explicitly state the assumption, without defending it in this thesis:

Assumption. Left Expansive Maximality *The only useful antinarrowings are those produced without expansion (widening) on the left hand sides of rules.*

4.3.3 Complete antinarrowing

Antinarrowing computes rules that are strict biexemplar justified ampliative theorems, that is theorems that are only justified by exactly two instances. In other words, theorems that are justified by three or more instances are not generated by the antinarrowing operator defined in the last section. This is the result of defining antinarrowing to use the pairwise *Rau* operator instead of the set *Rau* operator.

To compute all biexemplar justified theorems, the **complete antinarrowing** operator antiunifies sets of (complete) antinarrowings instead of just pairs:

Complete Antinarrowing

R (a rewrite system)
S (a set of rules)

$\{a | a \in S \vee a = Rau(S')$
where $S' \subseteq Cwid(S, R), Valid(a)\}$

The set of all complete antinarrowings of a set is denoted $Can(S, R)$ and $Can(R) = Can(R, R)$. Also, the *inconsistent complete widening* operator, denoted, $Can^i(R)$, is formed by replacing *Valid* with *Goodrule* in the definition.

The **complete widening** operator (*Cwid*) is simply widening defined in terms of complete antinarrowing instead of regular antinarrowing:

$$Cwid(r, R) = \{x | \; x = r \vee x = Wid(w, a),$$
$$\text{where } a \in Can(R), w \in Cwid(r, R), |Wid(w, a)| > |w|\}$$

Also, $Cwid(S, R)$, the form found in the definition of complete antinarrowing, is defined to be the set of all complete widenings of a set of rules S with respect to a rewrite system R.

Computing $Can(S, R)$ requires set antiunifying all subsets of complete widenings, and can therefore be a very slow process. It turns out that we have the alternative of using the regular antinarrowing and widening operators and pairwise antiunification to compute $Can(S, R)$ and pairs of widenings of differing widening size need not be antiunified as a result of the following proposition:

Conjecture 4.1 *Complete antinarrowings can be computed with binary rule antiunification instead of general set unification, and widening and antinarrowing operators instead of complete widening and complete antinarrowing respectively. More precisely,*

$$\begin{aligned} Can(R, S) &= \{a|\ a \in S \vee a = Rau(S'), \text{ where } S' \subseteq Cwid(S, R)\} \\ &= \{a|a \in S \vee a = Rau(p, q), \text{ where } \{p, q\} \subset Wid(S, R), p \neq q, |p| = |q| \\ &\quad \text{or } p \in Wid(S, R), q \in An(S, R)\} \end{aligned}$$

The consequence of this is that only pairs of widenings of the same size and widenings with previously computed antinarrowings need to be antiunified for implementing complete antinarrowings.

4.3.4 Example of Antinarrowing

The "append" function is a theorem that can be generated through complete antinarrowing. We use the dot notation syntax for lists of objects. The constant *nil* signifies an empty list, and *head.tail* signifies a list made up of a head object and a list tail. For example, *a.b.nil* denotes the list made of the objects a and b. The tail of this list is *b.nil*. The rules that comprise the append function are:

```
(1) app(nil,x)   -> x
(2) app(x.y,z)   -> x.app(y,z)
```

Figure 4.5 illustrates how complete antinarrowing produces the above two rules from the following examples:

```
(1) app(nil,a.nil)   -> a.nil
(2) app(nil,nil)   -> nil
(3) app(b.nil,nil)   -> b.nil
(4) app(c.nil,d.nil)   -> c.d.nil
(5) app(e.f.nil,g.nil)   -> e.f.g.nil
```

The two rules of append are generated in box A and box G of the figure. Recall that complete antinarrowings are antiunifications of (possibly more than two) widenings. Box A is the antiunification of two trivial widenings, namely examples (1) and (2), and so is produced by complete antinarrowing. Box G is the antiunification of the three widenings produced in boxes C, D, and F. Box E shows the intermediate pairwise antiunification of widenings C and D which is then antiunified in box G with the third widening from box F. (Recall that Proposition 4.2 shows that to antiunify a set of terms, pairwise antiunification can be used.) Box C shows the expansion (a widening with a rule from the original examples) of example (3) with example (2). Box D shows the widening of example (4) with the antinarrowing produced in box A. Finally, box B shows the antiunification, or trivial antinarrowing, of examples (3) and (4) that is used in box F to expand rule (5). Note that regular antinarrowing would only produce the antinarrowing in box E, while complete antinarrowing produces the antinarrowing in box G.

4.4 Reverse Cover Set Induction

Recall that cover set induction is a method of proving inductive theorems. Since inductive theorems are specific types of ampliative theorems and antinarrowing is used for ampliation, a restriction of antinarrowing will allow the generation of inductive theorems. In this section, inductive antiunification is first developed and then extended to inductive antinarrowing.

4.4.1 Inductive antiunification

Inductive antiunification is a special case of antiunification. It allows the generation of a new variable x if and only if it is fully justified by the facts. We define this method in terms of cover sets. For example, the set of terms $\{f(s(0)), f(0), f(s(s(y)))\}$ can be inductively antiunified to the single term $f(x)$ since $\{0, s(0), s(s(y))\}$ forms a cover set of the *num* type.

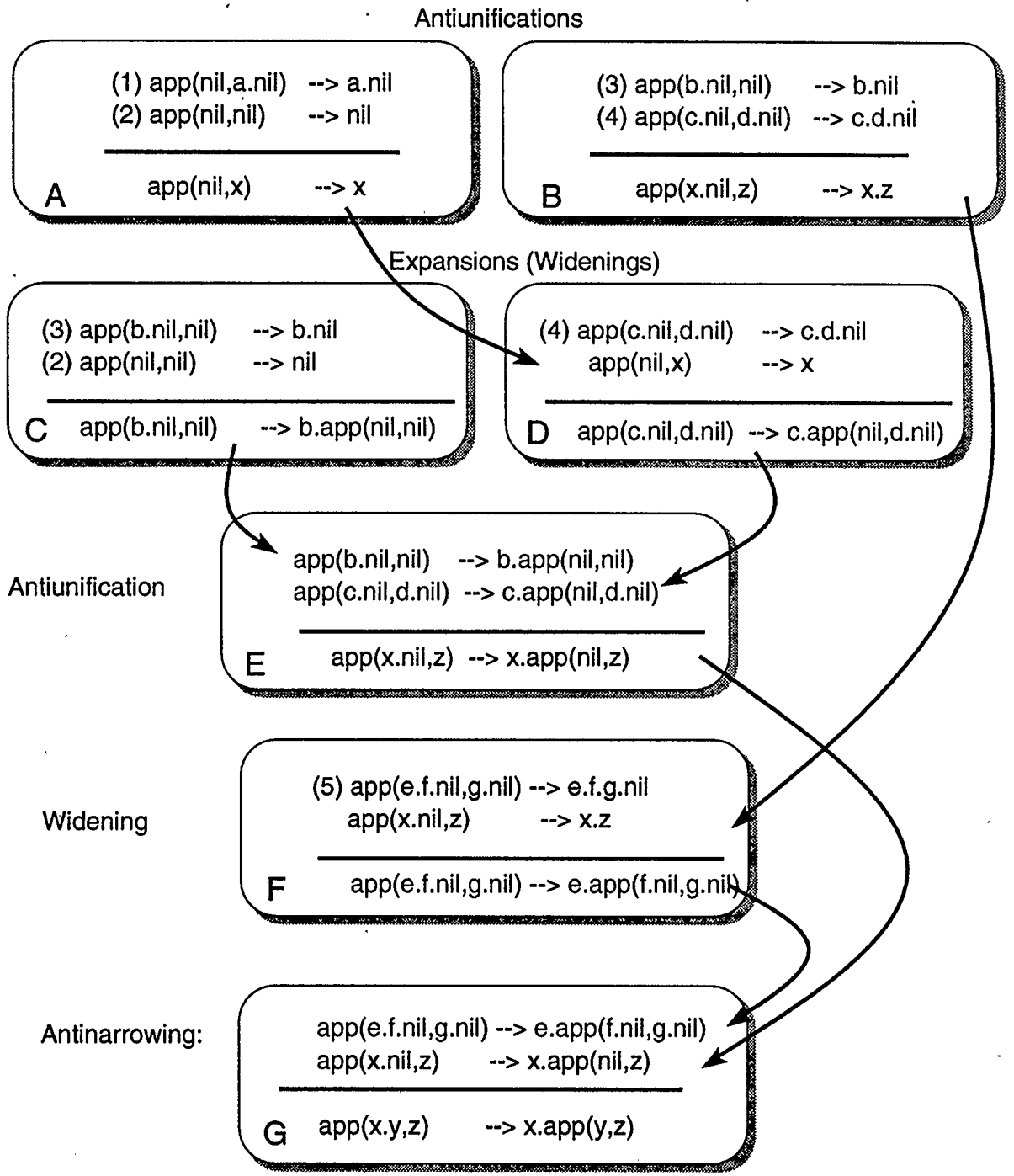


Figure 4.5. Example of complete antinarrowing

However, $\{f(s(0)), f(0)\}$ will not be inductively antiunified to $f(x)$ since $\{0, s(0)\}$ does not form a cover set for num. The operator is precisely defined on rules (not terms) as follows:

Inductive Antiunification

S (a set of rules)

$Rau(S)$

if $Msa(S) \equiv \{P_1/v_1, \ldots, P_n/v_n\}$ and
for each i, $1 \leq i \leq n$, p_i is some position in P_i,
$\{s_j/p_i \mid s_j \in S\}$ forms a cover set of $Type(v_i)$

The inductive antiunification of the set of rules S is denoted $Irau(S)$.

4.4.2 Inductive antinarrowing

To use background theory, another operator must be defined. We saw in Section 4.3 that applying an antiunification operator to exterms of the examples was not sufficient. Instead, the more comprehensive antinarrowing operator was developed to allow implicit background theory to be used in expansion. Here we develop a specific antinarrowing operator called *inductive antinarrowing*.

Like antinarrowing and complete antinarrowing, the definition of inductive antinarrowing is mutually dependant on the definition of inductive widening. Inductive widening is simply widening defined in terms of inductive antinarrowing instead of regular antinarrowing:

$$Iwid(r, R) = \{x \mid x = r \vee x = Wid(w, a),$$
$$\text{where } a \in Ian(R), w \in Iwid(r, R), |Wid(w, a)| > |w|\}$$

In other words, inductive widening is a type of rule expansion that also allows inductive antinarrowings of the given rules to be used to expand a rule. With this definition, inductive antinarrowing is defined as follows:

Inductive Antinarrowing
S (a set of rules)
R (a rewrite system)

$\{a | a \in R \vee a = Irau(p_1, \ldots, p_n)$
where $p_i \in Iwid(r_i, R)$ and $Valid(a, R)$ $\}$

The set of all inductive antinarrowings is denoted $Ian(S, R)$ and when $S = R$ we write $Ian(R)$. Notice that it is different from regular antinarrowing in that its need for a set of rules as input. If pairs of rules were input, as in regular antinarrowing, the only inductions possible would be inductions involving two-element cover sets.

The inductive antinarrowing operator can be used for advantage in machine learning. If there are inductive theorems that can be generated by a machine learner, it should generate them instead of non-inductive ampliative theorems since they are totally justified rather than just biexemplary justified.

4.5 Specialization

So far this chapter has only dealt with generalization operators. In some learning situations, generalizations, or rules or terms that cover positive examples, are checked for consistency against negative examples to determine if they are valid. Instead of being used simply for consistency checking of generalizations, negative examples can be used directly to produce valid generalizations that exclude them. *Specialization* refers to this use of negative examples. This process requires an already-generated hypothesis and a set of examples that need to be excluded from the hypothesis. Note that the examples that are to be excluded also could have underdetermined or overdetermined truth values, rather than only false truth values, as with negative examples. In other words, specialization is a method of reducing underdeterminacy and overdeterminacy (inconsistency) from a hypothesis.

Many popular machine learning algorithms can be viewed as specialization algorithms. A logic program generation technique, MIS (Shapiro, 1983) uses specialization to modify

incorrectly learned programs. Also, explanation-based learning methods (Mitchell, Keller & Kedar-Cabelli, 1986; Krawchuk & Witten, 1989), select specializations of concepts that cover a given example and that satisfy other criteria such as efficiency or simplicity. Classification systems also use specialization as their only method of generating their search spaces. We will study the particular specialization styles of ID3 (Quinlan, 1986), PRISM (Cendrowska, 1987) and Induct (Gaines, 1991) in detail in Chapter 6. To understand these algorithms more fully, specialization and its relationship with generalization will be explained here.

4.5.1 Term specialization

Definition 4.10 *A* **specialization** *of a term g is a set (or disjunction) of instances of g, $\{g\theta_1, g\theta_2, ..., g\theta_n\} = S$. $\{\theta_1, \theta_2, ..., \theta_n\}$ is called a* **specializer**. *S is a* **complete specialization** *of g if all proper instances of g are instances of some member of S. S is a* **minimal complete specialization** *of g if for all other complete specializations S_o, $|S| \leq |S_o|$. S is a* **most general specialization** *of g if there is no other specialization S' of g such that all terms in S are instances of some member of S'.*

For example, suppose that $Type(x) = Type(y) = num$. Then the set of all most general specializations of $f(x,y)$ is $\{\{f(0,y), f(s(z),y)\}, \{f(x,0), f(x,s(z))\}\}$. Both of these are minimal complete specializations since any other complete specializations contain more terms. An example of a specialization that is not complete is $\{f(0,y), f(s(0),y)\}$. One that is complete but not most general nor minimally complete is $\{f(x,0), f(x,s(s(z))), f(x,s(0))\}$.

More often specialization is done with respect to a set of negative examples. Suppose g is a term, X is a set of terms, and $S = \{g\theta_1, g\theta_2, ..., g\theta_n\}$ is a specialization of g. If $X \subseteq S$, then $S - X$ is the *specialization of a term g with respect to X*. For example, the *most general* specialization of $f(x,y)$ with respect to $\{f(w,s(z))\}$ is $\{f(x,0)\}$.

This definition is inadequate since it does not suggest any method of computing specializations. Doing so would require a search through all specializers of the general term: clearly an inefficient process. Recall that in Chapter 3 a similar difficulty was found in determining the inductive-theoremhood of a formula. There the solution was to use cover sets to reduce the search space. Here we use cover sets to directly produce a specialization:

Naive Cover Set Specialization

g (a term)
x $\in$ *Vars*(g)

$\{g\theta_1, ..., g\theta_n\}$
where $\theta_i = \{x/c_i\}$ and
$\{c_1, ..., c_n\}$ is a cover set of *Type*(x)

The set of all naive cover set specializations of g with a particular variable x is denoted $Nsp(g, x)$ and the set of all naive cover set specializations of g ranging over all of the variables in g is denoted $Nsp(g)$. It can be shown that when no two variables are of the same type in a term g, all of the naive cover set specializations of g are maximally general complete specializations of g.

We call this operator "naive" since it does not always produce all of the possible complete most general specializations. Consider the type A with the cover set $\{a, b\}$ and variables x and y of type A. The naive cover set specializations of the term $f(x, y)$ are $\{f(a, y), f(b, y)\}$ and $\{f(x, a), f(x, b)\}$. But the set $\{f(x, x), f(a, b), f(b, a)\}$ is a (complete) most general specialization of $f(x, y)$ that is ignored by this method. Notice that in this example variable y is renamed to another variable (x) in the term being specialized. Generalizing this idea a little, the naive method can be amended to allow these other most general specializations:

Cover Set Specialization

g (a term)
x $\in$ *Vars*(g)

$Nsp(g, x) \cup K$
where $K = \{g\theta | \theta = \{x/y\}$ for some $y \in$ *Vars*(g)
such that $x \neq y$ and *Type*(x) = *Type*$(y)\}$

The set of cover set specializations is written $Sp(g)$ and the cover set specializations of g with respect to a particular variable x is denoted $Sp(g, x)$. Note that naive cover set specializations and the cover set specializations are differentiated because some machine learning algorithms

opt for the naive method of specialization. Chapter 6 discusses this further. Some interesting facts result from this definition:

Proposition 4.5

1. If $S = Sp(g)$ and $|S| > 1$ then $Au(S) = g$
2. $Sp(Au(\{g\theta_1 ... g\theta_n\})) = \{\{g\theta_1 ... g\theta_n\}\}$ iff $\{\theta_1, ... \theta_n\}$ is a most general complete specializer of g.

This proposition shows the extent to which *Au* and *Spec* are inverses of each other. The first item says that antiunifying a most general specialization produced by cover set specialization returns the original general term. The qualification in the first item notes that sometimes a specialization with respect to a variable is a singleton. The second says that specialization returns the original set of terms given to antiunification if the set completely covers all instances of g. Since *Au* is a form of antiapplication, then *Spec* must be a particular form of application since it is an inverse of *Au*.

Specialization algorithms can be primed with a general term (or equation) g that is the antiunification of all the examples. If this is done, it is possible to show that all specializations are antiunifications of each member of the powerset of the positive examples.

Conjecture 4.2 *Let P be a set of positive examples and $g = Au(P)$. Then* $Sp(g) = \{\bigcup_i(\bigcup_j Au(S_{ij}))|\{S_{i1} ... S_{in}\}$ *is a partition of* $P\}$

Conjecture 4.2 suggests that anything that can be done with cover set specialization can be done with an antiunification method and vice versa.

Cover set specialization has been defined with respect to variables instead of with respect to negative examples, as is required for some machine learning systems. First, we define specialization on a set of terms, A:

$$Sp(A) = \{ .G|\ G \in Sp(g) \text{ and } g \in A\}.$$

Then specialization of a set of terms with respect to variables can be defined as:

$$Sp(A, N) = \begin{cases} \{A - N\} & N' \subseteq A \\ \{G \mid\ G \in Sp(Sp(A), N')\} & otherwise \end{cases}$$

$$\text{where } N' = \{n | n \in N \wedge \exists s \in A, \theta \; s\theta = n\}$$

Thus, specialization with respect to negative examples is defined by using regular specialization and is not conceptually more difficult. An efficient and straightforward implementation of $Sp(S, N)$ was done by Lassez and Marriott (1986).

4.5.2 Rule specialization

So far, specialization has only been discussed with respect to terms. However, rule specialization can also be defined by using term specialization:

Rule Cover Set Specialization

r	(a rule)
V	($\subseteq$ *Vars*(r))
R	(a rewrite system)

Makerule(*Nf*(g, R))
where $g \in$ *Sp*(*Termify*(r), V)

This operator converts the rule to a term (with the *Termify* function) and then specializes it with respect to variables. The resulting term is reduced with the given rewrite system (with the normal form function, *Nf*, from Section 3.2) and is then converted back into a rule (with *Makerule*). The set of all rules produced in this way is denoted *Spec*(r, V, R) and each member of this set is called a **rule specialization**. When a background theory is not used for specialization, then R is empty and we simply write *Spec*(r, V). When $V =$ *Vars*(r) we write *Spec*(r) or *Spec*(r, R). Also, if *Spec* in the definition is replaced with *Nsp*, the result **naive rule specialization** is defined and the whole set of them is denoted *Nspec*(r, V, R) (or its related forms).

Finally, sets of rules can be specialized simultaneously. A **naive rule set specialization** is defined as follows:

Rule Set Specialization

S	(a set of rules)
V	($\subseteq Vars(r)$)
R	(a rewrite system)

$\{X | X = Nspec(s, \{v\}, R)$ for an $s \in S$ and $v \in V\}$

The set of all such specializations is denoted $Nspec(S, V, R)$. Rule set specialization, $Spec(S, V, R)$, is defined by replacing *Nspec* with *Spec* in the definition.

We complete this section with a few propositions regarding the soundness of rule specialization.

Proposition 4.6 *Rule specialization is not an ampliatively sound operator. That is,* $r \in AMP(R) \not\Rightarrow \forall s, s \in Spec(r, R) \Rightarrow s \in AMP(R)$

However, when $r \in IND(R)$, all rule specializations will be inductive theorems. This is a result of the fact that inductive theorems have total justification, and thus all its instances will have total justification as well.

Proposition 4.7 *Rule specialization is an inductively sound operator. That is,* $r \in IND(R) \Rightarrow \forall s, s \in Spec(r, R) \Rightarrow s \in IND(\mathcal{E})(R)$

The consequence of this is that if rule specialization is used as the major form of theorem generation, it is necessary to check for the justification of the rules it generates unless the rule being specialized is an inductive theorem.

4.6 Completeness

We have shown that complete antinarrowing is ampliatively sound, and rule specialization is inductively sound. Though not ampliatively sound, rule specialization is deductively sound since doing rule specialization on a set of rules, a set of deductive theorems, will only produce more deductive theorems. We are finally in a position to show that applying

complete antinarrowing to the rule specializations of a theory will construct all possible conjuncts that can make up ampliative theorems.

Proposition 4.8 Prudential Completeness *If e is a conjunctive negationless universal prudent theorem of a prudent extension of a theory $\mathcal{E}$ with axioms R, then for each conjunct c of e, $c \in (Can^{i}(Spec(R)))$*

Since inductive, ampliative, and ignorative theorems are subclasses of prudent theorems, this result extends to them as well. By restricting the antinarrowing method to *Can* completeness is not effected since *Can* generates a subset of the rules generated by Can^{i}. This completeness result shows us that no other operators need be developed for effective generation of theorems.

4.7 Summary

This chapter began with a look at term antiunification. Term antiunification was modified slightly to allow the generalization of rules. The expansion operator was introduced as a technique for generating deductive theorems. These methods were combined into the E-antiunification operator to enable equational antiunification, that is, generalization of rules using explicit background theory. To enable the use of implicit background theory, antinarrowing was developed. To generate all biexemplar justified theorems, not just the strictly biexemplar justified ones, the complete antinarrowing operator was designed. Next, the inductive antinarrowing operator was created to generate inductive theorems. Also, another important method of induction, specialization, was shown to be an inverse of antiunification – in particular, the application of a set of substitutions. Specialization, together with inconsistent complete antinarrowing were suggested to be complete for generating a restricted subclass of prudent theorems.

Chapter 5

Balog: Automated Theorem Generation

Previous chapters developed a framework and some techniques for generating theorems. This chapter describes practical applications of these methods that are incorporated into a computer program called Balog. The Balog system has been developed as a test bed for various equational logic theorem proving and generation methods. Figure 5.1 lists some of the functions of Balog.

All examples, theories, and programs in Balog are specified in a custom functional programming language based on conditional term rewriting systems. Figure 5.2 is an example of a Balog program used to specify a generation problem. The `var` command declares variables, the `type` command declares type names. The infix ":" command declares a function by specifying its types (here the "x" means "cross"). For example, the *prescribe* function is declared to be a function of four arguments with types *age, spec, astig,* and *tear* respectively and which returns a term of type *lens*. Also, the function symbols *reduced* and *normal* are simultaneously declared to be arity 0 functions of type *tear*. The "*" indicates that these are constructor functions. The `use` command tells Balog to put all following rules into one of the several rule bases Balog maintains (in this case the rules are put in the example base). Several other programming language commands are available to control the operation of Balog. More control commands are available at the command line interface. Typically, Balog program files are loaded with the `load` command and then the loaded rules are processed by other commands, such as the `showx` command to show the example rule base or the `balog/amp` command to run the Balog/AMP ampliative theorem generation program developed in this Chapter.

Function	What it Implements	Where Used
Rewrite	***Nf*** *-- normal form (Section 3.2)*	Balog command line
Solve	***narrowing*** *(Section 3.2)*	Balog command line
Ind-proof-check	***cover set induction*** *(Section 3.2)*	Balog/IND *(Section 5.5)*
PhaseOne	***Can^i*** *-- inconsistent complete antinarrowings (Section 4.3.3)*	Balog/PRU *(Section 5.4)*
	Can *--complete antinarrowings when the Valid function is used instead of Goodrule (Section 4.3.3)*	Balog/AMP *(Section 5.1.1)*
	An *-- antinarrowings when Limit = 0 and the Valid function is used (Section 4.3.2)*	To speed up Balog/AMP and Balog/PRU
	Rau *-- rule antiunifications when Depth = 0 (Section 4.1.2)*	Balog/C *(Section 5.2)* Balog/CD *(Section 5.3)*
PhaseZero	***Spec*** *-- rule set specializations (Section 4.5.2)*	Balog/IND *(Section 5.5)*

Figure 5.1. Some functions used in Balog

```
;-------------------------------------------------------------
;       contact.b
;       contact lens example adapted from Cendrowska (1987)
;-------------------------------------------------------------
var     w,z,y,x
type    lens
        {hard,soft,none} : lens*
type    age     ; ppres is pre-presbyopic
        {young,ppres,pres} : age*
type    spec    ; myope or hypermyope
        {myope,hyper}: spec*
type    astig
        {yes,no} : astig*
type    tear    ; tear production
        {reduced,normal} : tear*
use     example
prescribe : age x spec x astig x tear -> lens
        prescribe(young,myope,no,reduced)       ->      none    ; 1
        prescribe(young,myope,no,normal)        ->      soft
        prescribe(young,myope,yes,reduced)      ->      none
        prescribe(young,myope,yes,normal)       ->      hard
        prescribe(young,hyper,no,reduced)       ->      none    ; 5
        prescribe(young,hyper,no,normal)        ->      soft
        prescribe(young,hyper,yes,reduced)      ->      none
        prescribe(young,hyper,yes,normal)       ->      hard
        prescribe(ppres,myope,no,reduced)       ->      none
        prescribe(ppres,myope,no,normal)        ->      soft    ; 10
        prescribe(ppres,myope,yes,reduced)      ->      none
        prescribe(ppres,myope,yes,normal)       ->      hard
        prescribe(ppres,hyper,no,reduced)       ->      none
        prescribe(ppres,hyper,no,normal)        ->      soft
        prescribe(ppres,hyper,yes,reduced)      ->      none    ; 15
        prescribe(ppres,hyper,yes,normal)       ->      none
        prescribe(pres,myope,no,reduced)        ->      none
        prescribe(pres,myope,no,normal)         ->      none
        prescribe(pres,myope,yes,reduced)       ->      none
        prescribe(pres,myope,yes,normal)        ->      hard    ; 20
        prescribe(pres,hyper,no,reduced)        ->      none
        prescribe(pres,hyper,no,normal)         ->      soft
        prescribe(pres,hyper,yes,reduced)       ->      none
        prescribe(pres,hyper,yes,normal)        ->      none    ; 24
```

Figure 5.2. A Balog Program: contact.b

Some of the proof techniques implemented in Balog are conditional term rewriting, narrowing, completion, reduction ordering, proof by consistency, and cover-set induction. Generation techniques include implementations of ID3, Prism and Induct, and those mentioned in this chapter — an ampliative theory learning system (Balog/AMP), a classification system (Balog/C), a default theory learner (Balog/CD), a prudent theorem generator (Balog/PRU) and an inductive theorem generator (Balog/IND). The two central algorithms in these systems, namely *PhaseZero* and *PhaseOne*, implement the *Spec* operator from Section 4.5.2 and the Can^i operator from Section 4.3.3 respectively.

5.1 Theory Learning

The kind of machine learning problems that we will consider first are *theory learning* problems. Here we are given an original theory A that is comprised of examples in the form of rules and possibly background theory in the form of rules. (The distinction between examples and background theory is not necessary). The goal of theory learning is to generate a more compact theory B made up of inductivelike theorems of the given theory A. We will restrict our attention to theories that are conjunctions of rules. Generating candidate theories, or *hypotheses*, for B is straightforward given the techniques of Chapter 4.

The next subsection describes the generation of ampliative theorems from a given theory — generally called "noiseless induction" in the machine learning paradigm — embodied in a program called Balog/AMP. The following subsections detail how hypotheses may be evaluated in order to choose the best among them.

5.1.1 Ampliative theory learning

Balog/AMP is an implementation of a two pass algorithm for inducing ampliative theories from a given theory. A standard example in logic programming and machine learning is the "append" program. One reason for its use here is that it is a simple function that is also recursive. The rules in Figure 5.3 constitute the append function that Balog/AMP will learn from the examples in Figure 5.4, and which we use as a recurring example to help describe the operation of the system.

```
(1) app(nil,x)   -> x
(2) app(x.y,z)   -> x.app(y,z)
```

Figure 5.3. The Append Function

```
[balog] showx
--- example base ---
(1) app(nil,a.nil)   -> a.nil
(2) app(nil,nil)   -> nil
(3) app(b.nil,nil)   -> b.nil
(4) app(c.nil,d.nil)   -> c.d.nil
(5) app(e.f.nil,g.nil)   -> e.f.g.nil
```

Figure 5.4. An example set for inducing the Append function

The *first phase* of Balog/AMP generates ampliative theorems of a given theory by implementing the complete antinarrowing operator. In particular, Balog/AMP generates all rules that are ampliative theorems of an example base. Hypotheses are combinations (conjunctions) of these rules. The algorithm for this phase is shown in Figure 5.5. For the append example, it produces the output shown in Figure 5.6 (when the *Depth* variable is set to 2; see below).

Phase One Algorithm

$PhaseOne(X, Depth, Limit, Valid)$

X	— example base
$Depth$	— maximum antinarrowing size
$Limit$	— maximum antiunification iterations
$Valid$	— function to restrict rules formed

begin
  $WL = X$
  $AL = X$
  $i = 0$
  while $(i \leq Depth)$
  begin
    $WW = \{a|\ a = Rau(p,q) \wedge Valid(a)$
      where $p, q \in WL, |p| < max(0, i-1), |q| = max(0, i-1)\}$
    $NewAU = WW$
    $j = 0$
    while $(j < Limit \wedge (j = 0 \neq Limit \vee NewAW \neq \emptyset))$
    begin
      $NewAW = \{a|\ a = Rau(p,q) \wedge Valid(a)$
        where $p \in AL, q \in NewAW, |p| = |q| = i\}$
      $AW = AW \cup NewAW$
      $j = j + 1$
    end
    $AL = AL \cup WW \cup AW$
    if $(i \neq Depth)$
      $WL = \{x|x = w/a$ where $a \in WW \cup AW, w \in WL, |w| = Depth - 1\}$
end
Output: AL — antinarrowings to a depth of *Depth*.

Figure 5.5. Phase one algorithm

```
[balog] balog/amp

Balog/AMP - Version 3.0

Phase 1. Candidate Rules
Compute Antinarrowings [0] (widenings=5 antinarrowings=5)
Augment Widenings      [1] (widenings=5 antinarrowings=8)
Compute Antinarrowings [1] (widenings=23 antinarrowings=8)
Augment Widenings      [2] (widenings=23 antinarrowings=13)
Compute Antinarrowings [2] (widenings=23 antinarrowings=13)
Removed (0) inconsistent rules.

Widenings
---------
6 [1] app(nil,a.nil)  -> a.app(nil,nil)
Source (Wid An)  : ((1 2) (1 6) (1 7))
7 [1] app(nil,a.nil)  -> app(a.nil,nil)
Source (Wid An)  : ((1 7) (1 8))
8 [1] app(b.nil,nil)  -> b.app(nil,nil)
Source (Wid An)  : ((3 2) (3 6) (3 7))
9 [1] app(b.nil,nil)  -> app(nil,b.nil)
Source (Wid An)  : ((3 6))
10 [1] app(c.nil,d.nil)  -> c.d.app(nil,nil)
Source (Wid An)  : ((4 2) (4 6) (4 7))
11 [1] app(c.nil,d.nil)  -> app(nil,c.d.nil)
Source (Wid An)  : ((4 6))
12 [1] app(c.nil,d.nil)  -> c.app(nil,d.nil)
Source (Wid An)  : ((4 6))
13 [1] app(c.nil,d.nil)  -> app(c.d.nil,nil)
Source (Wid An)  : ((4 7))
14 [1] app(c.nil,d.nil)  -> c.app(d.nil,nil)
Source (Wid An)  : ((4 7) (4 8))
15 [1] app(e.f.nil,g.nil)  -> e.f.g.app(nil,nil)
Source (Wid An)  : ((5 2) (5 6) (5 7))
16 [1] app(e.f.nil,g.nil)  -> app(nil,e.f.g.nil)
Source (Wid An)  : ((5 6))
17 [1] app(e.f.nil,g.nil)  -> e.app(nil,f.g.nil)
Source (Wid An)  : ((5 6))
18 [1] app(e.f.nil,g.nil)  -> e.f.app(nil,g.nil)
Source (Wid An)  : ((5 6))
19 [1] app(e.f.nil,g.nil)  -> app(e.f.g.nil,nil)
Source (Wid An)  : ((5 7))
20 [1] app(e.f.nil,g.nil)  -> e.app(f.g.nil,nil)
Source (Wid An)  : ((5 7))
21 [1] app(e.f.nil,g.nil)  -> e.f.app(g.nil,nil)
Source (Wid An)  : ((5 7) (5 8))
22 [1] app(e.f.nil,g.nil)  -> app(e.nil,f.g.nil)
Source (Wid An)  : ((5 8))
23 [1] app(e.f.nil,g.nil)  -> e.app(f.nil,g.nil)
Source (Wid An)  : ((5 8))

Antinarrowings:
---------------
1 [0] app(nil,a.nil)  -> a.nil
2 [0] app(nil,nil)  -> nil
3 [0] app(b.nil,nil)  -> b.nil
4 [0] app(c.nil,d.nil)  -> c.d.nil
5 [0] app(e.f.nil,g.nil)  -> e.f.g.nil
6 [1] app(nil,x)  -> x
From widenings: ((1 2) (2 6))
7 [1] app(x,nil)  -> x
From widenings: ((2 3) (3 7))
8 [1] app(x.nil,y)  -> x.y
From widenings: ((3 4) (4 8))
9 [2] app(x.nil,y)  -> app(nil,x.y)
From widenings: ((11 9))
10 [2] app(x.nil,y)  -> x.app(nil,y)
From widenings: ((12 8))
11 [2] app(x.nil,y)  -> x.app(y,nil)
From widenings: ((14 8))
12 [2] app(x.y,z)  -> x.app(y,z)
From widenings: ((23 8))
13 [2] app(x.y,z.nil)  -> x.app(y,z.nil)
From widenings: ((23 12))
```

Figure 5.6. Balog/AMP after phase one on append data

Conjecture 4.1 suggested that to generate all ampliative theorems with the complete antinarrowing operator, it is sufficient to antiunify widenings with widenings of equal sizes and antinarrowings that have smaller sizes. Balog/AMP implements this directly by generating a list of widenings and a list of antinarrowings, up to a specified induction depth. A size N widening means that N widening steps are required to create it from the original theory. (Each rule in the original theory has size 0.) A size N antinarrowing means that a size N widening is used to create the antinarrowing through rule antiunification with some other rule with size less than or equal to N. Balog/AMP terminates phase one when all antinarrowings of size equal to the maximum induction depth are generated.

To start this phase, Balog/AMP initializes both the widening list (WL) and the antinarrowing list (AL) to the original theory. Next, rule antiunifications of all of the widenings in WL are added to AL. For the append example, all five example rules (see Figure 5.4) will be first installed in both WL and AL. Next, rule antiunifications of all of these new antinarrowings in AL are antiunified with the widenings in WL and these are added to AL. This continues until no new antinarrowings are created. At this point, AL will contain all antinarrowings of size 0 of the original theory. In the append example, antinarrowings 1 to 8 in Figure 5.6 are all antinarrowings of size 0 — they are all the antiunifications of the given append data.

Then Balog/AMP starts working on the next level. First, all widenings of size 1 are created. These are all the widenings of size 0 widened with antinarrowings of size 0 (everything in AL so far). In general, widenings of size N are widenings of widenings of size $N-1$ or less with antinarrowings of size $N-1$ or less. Of course, Balog/AMP does not re-create widenings that have been created on previous levels. That is, only widenings that will be of size N are created. All of these new widenings are added to WL. In the append example, 18 new widenings (numbers 6 to 23) are created. Second, new antinarrowings are created and put in AL by antiunifying the new widenings with each rule in WL. In the append example, antinarrowings 9 through 13 are added. Finally, all new antinarrowings are antiunified with the widenings in WL, to create more antinarrowings. The operation is repeated with these newest antinarrowings until no new antinarrowings are created or until the maximum antiunification passes limit (*Limit*) is reached. At this point, all antinarrowings of level 1 (or N in general) will have been created and put in AL. In the append example,

antiunifying rules 9-13 with each other does not produce any new rules. Balog repeats this process for the next level, until the maximum induction depth (*Depth*) is reached.

Balog/AMP does not create inconsistent rules. Before an antinarrowing is put in AL, it is first checked to see if it is consistent with the original theory, and if not, it is discarded. This is better than doing the inconsistency removal in a separate pass because if inconsistent antinarrowings are left in, all of the antiunifications that use them will also be inconsistent, resulting in unnecessary work. Other systems described below (Balog/PRU and Balog/CD) simply retain all created rules, since they allow some inconsistency.

The *second phase* investigates combinations of the rules produced in phase one to generate possible theories. Figure 5.7 displays this algorithm. One method, *complete generation*, investigates all combinations of the rules created in the first phase. Each of these theories, called *hypotheses*, is evaluated and the hypothesis with the best evaluation is chosen to be the final generated theory. Section 5.1.2 describes possible evaluation methods. On any but very small problems, complete generation is excruciatingly slow. For example, if phase one generates 100 rules, then $2^{100} - 1 = 1.2 \times 10^{29}$ hypotheses will have to be tested. For the append example, 8 new antinarrowings are generated in phase one; so $2^8 = 256$ hypotheses will have to be evaluated. Accordingly, computation bounds, time limits, and searching hypotheses from shortest to longest can be used while doing complete generation. If there exists a short theory, it will be found relatively soon in the computation.

Usually the *incremental hypothesis generation* method is more practicable. This creates a hypothesis by adding appropriate rules to it, one at a time. A rule is chosen by a heuristic method, called an *increment function*, to be added to the hypothesis generated so far. The purpose of the increment function is to find the rule that is most likely, by some measure, to be included in the final theory. The adding of rules to hypotheses stops when the increment function fails to generate any new rules. Section 5.1.3 describes some possible increment functions.

When Balog/AMP is run on the append examples using the *generated coverage increment function* described in Section 5.1.3, the desired append function is generated in about 100 seconds. The complete generation method using the *compression heuristic* described in Section 5.1.2 also generates the same set of two rules, but takes about 125 seconds.

Phase Two Algorithm

$PhaseTwo(An, Cflag, F)$

	An	— antinarrowings
	$Cflag$	— if true use complete hypothesis generation
	F	— increment function if $Cflag$ is false
		evaluation function if $Cflag$ is true

```
begin
   if (Cflag)
   then
     H = F(Powerset(An))
   else begin
     A = F(An, ∅)
     while (A ≠ ∅)
     begin
          H = H ∪ A
          An = An − {A}
          A = F(An, H)
     end
   end
end
```

Output: H — a chosen hypothesis.

Figure 5.7. Phase two algorithm

Phases one and two together form the complete Balog/AMP algorithm exhibited in Figure 5.8. Because Balog/AMP uses the *Valid* function instead of the *Goodrule* function, it uses complete antinarrowing instead of inconsistent complete antinarrowing.

5.1.2 Evaluating hypotheses

Balog requires a method of judging the relative merits of the hypotheses generated by the complete generation method of phase two. The easiest method, at least from Balog's view, is to use an *oracle*. This could be another specialized program which performs experiments using the hypotheses and chooses one based on the results. Alternatively, the oracle could

be an expert user who is required to choose the best hypothesis.

Balog can also use heuristics to decide between hypotheses. One is the *coverage heuristic*. Given a set of examples and a theory H, the number of examples that are true in H is called the *coverage* of H in E and is written $coverage(H, E)$. In more colloquial words, $coverage(H, E)$ is the number of examples that H explains or are redundant. Since a hypothesis H should already cover all of the examples of the original theory E (because it ampliatively subsumes E), $coverage(H - E, E)$ is a more informative measure — only the rules in H that are not in E are tested for coverage. When Balog uses this heuristic, the hypothesis that has the largest coverage is chosen.[1]

A variation on the coverage heuristic is the *cover set heuristic*. This measures the number of *possible* examples that a hypothesis can cover, even ones that it has not seen. Suppose the cover set of the function symbol $f/2$ is $\{(a,a),(a,b),(b,a),(b,b)\}$. Then the hypothesis $\{f(x,x) \rightarrow x\}$ would only cover two out of a possible four. The hypothesis $\{f(x,y) \rightarrow y\}$ covers all four and is thus to be preferred. This method can be used with infinite data types, since cover sets are always finite. Note that while the coverage heuristic measures the range of the rules in hypotheses, the cover set heuristic measures their domain. The cover set heuristic is usually used in combination with other heuristics.

Another possibility is the *hypothesis length heuristic* which counts of the number of rules in the hypotheses and chooses the shortest. Very often, the best hypothesis is the one with least rules because it is usually the easiest to understand. This heurisitic is generally used in conjunction with other heuristics for tie breaking.

The most powerful heuristic used in Balog is the *hypothesis complexity heuristic*. There are many ways of defining the complexity of a set of rules. One is to count the number of variables in the hypotheses, and prefer those with more variables. The number of symbols or the nesting levels of the terms in the hypotheses could also be counted. A principled method of determining complexity, applied recently to several machine learning programs, for example, Cigol (Muggleton & Buntine, 1988), is based on Kolmogorov algorithmic complexity theory (Kolmogorov, 1965). Balog uses a modification of the approach used in Cigol that computes the "information content" loss between the original theory and the

[1] In Balog, coverage can be determined with respect to another set of rules called the *testbase*, which is distinct from the example base.

hypotheses and prefers the one with the most loss.

5.1.3 Hypothesis increment functions

An hypothesis increment function takes a temporary hypothesis, H, and chooses a rule r from among a set of rules A that is the most likely rule to be part of the final desired theory. In Balog, A is the set of antinarrowings produced from phase one. Once a rule is chosen by the increment function, it can be added to H to create a new temporary hypothesis, $H \cup \{r\}$. The most straightforward method is to use one of the hypothesis evaluation methods described in the previous section applied to $\{H \cup \{r\} | r \in A\}$. However, less computationally costly methods are available.

A common increment function is the *coverage increment function.* This function chooses the rule that covers the most example rules. It is particularly useful for classification problems (see the next section) rather than general problems, since the definition of coverage applied to single rules might not be applicable in general problems. For example, it is impossible to compute which examples in Figure 5.4 are covered by rule number 2 in Figure 5.3 without using rule number 1. For general rules, we might count how many examples were used to generate the rule instead. For instance, examples 3 and 4 were used to generate the antinarrowing that was antiunified with example 5 to create rule 2. So a total of three example rules (3,4,5) were used to create rule 2. This increment function is called the *generated coverage increment function.* Another increment function that can be used is the *syntactic simplicity increment function.* This selects the rule that uses the fewest syntactic symbols (and the most variables). There are many possibilities for increment functions besides the ones mentioned here.

5.2 Classification Learning

Balog/C is a classification learning program like ID3 (Quinlan, 1986) and PRISM (Cendrowska, 1987). A classification system only allows examples, and only generates rules, that have a class identifier as their right hand side. Also, no background theory is used. Such systems are a particular kind of theory learning system. In fact, if

the antinarrowing size is set to zero in Balog, no background theory can be used since no widening will be done, and a classification induction system results. The Balog/C algorithm is exactly the same as that of Balog/AMP (Figure 5.8) except that *Depth* is forced to be 0. This enables only antinarrowings of size 0, or rule antiunifications (*Rau*), to be produced. Also, Balog/C restricts its input (X) to the kind of rules described below.

5.2.1 Representing classifiers as rewrite rules

Classification programs deal with entities much simpler than the general rules of Balog. Each example instance is made up of an *attribute/value vector* and a *class identifier* where each vector is comprised of attribute/value pairs. One representation of these is:

[(a1,v1), (a2,v2) ... (am,vm)] of Class

where Class is the class identifier, [...] represents the vector, and each (a_i,v_i) is an attribute/value pair. For example, suppose a set of rules is sought for determining if it is a Saturday based solely on the weather. An example instance in this domain might be:

[(outlook,overcast),(temp,hot),(humidity,high),(windy,false)] of sat

and another:

[(outlook,sunny),(temp,mild),(humidity,high),(windy,false)] of other

It is quite natural to represent these as term rewrite rules: the attribute value vector is a term on the left hand side of the rule and the class is a constant on the right hand side. For the former, a new function symbol is invented with any name and with one argument for each attribute. The arguments are then filled with their corresponding values.

conditions : outlook × temp × humid × wind → day
conditions(overcast,hot,high,false) → sat
conditions(sunny,mild,high,false) → other

Further restrictions on these rules include the lack of specification or use of background theory and the use of only finite types. In summary, the representation in a classification system is equivalent to a term rewriting system representation in which

- there is one arbitrarily named function symbol f of arity a;
- each argument i of f has a finite type, $attribute_i$;
- the right hand side of each rules is a constant of type $class$;
- all $attribute_i$ are disjoint with $class$;
- all $attribute_i$ are disjoint with each other.

The last two conditions are dropped in Balog/C, enabling it to address a wider class of classification problems than specialization-based classification learners such as ID3.

5.2.2 Example of classification

Figure 5.2 specifies the well-known contact lens example (Cendrowska, 1987) in Balog's functional language. Three categories of contact lens — hard, soft, or none at all — constitute the classes in this problem. Four attributes are deemed to be relevant to prescribing lenses — age, tear production level, presence of astigmatism, and specification. Each has a set of possible values; for example, the attribute *spec* can be either myope or hypermyope. Finally, for each combination of the values of attributes, the appropriate type of contact lens is specified in Figure 5.7. It is the responsibility of Balog/C to determine the best theory that compactly contains all the knowledge of these 24 rules.

During the first phase, Balog/C generates 53 new rules that may be used in the final theory; they are listed in Figure 5.9. However, many of these rules are inconsistent with the examples and, like Balog/AMP, Balog/C checks for inconsistency and only records antinarrowings that are consistent. In the figure, each rule that is inconsistent is labelled with an "Inconsistent:" field that lists the rules found to be inconsistent with it. Figure 5.10 shows the 32 new consistent rules generated by the first phase. The original 24 rules can also be used in the generated theory, so they are included in the list of antinarrowings produced by the first phase.

Balog/AMP - Balog Ampliative Algorithm

Balog/Amp(*X,Depth,Limit,Cflag,F*)

	X	— example base
	Depth	— maximum antinarrowing size
	Limit	— maximum antiunification passes
	Cflag	— do complete phase two search if true
	F	— evaluation or increment function

begin

$R = PhaseTwo(PhaseOne(X,Depth,Limit,\text{Valid}),Cflag,F)$

end

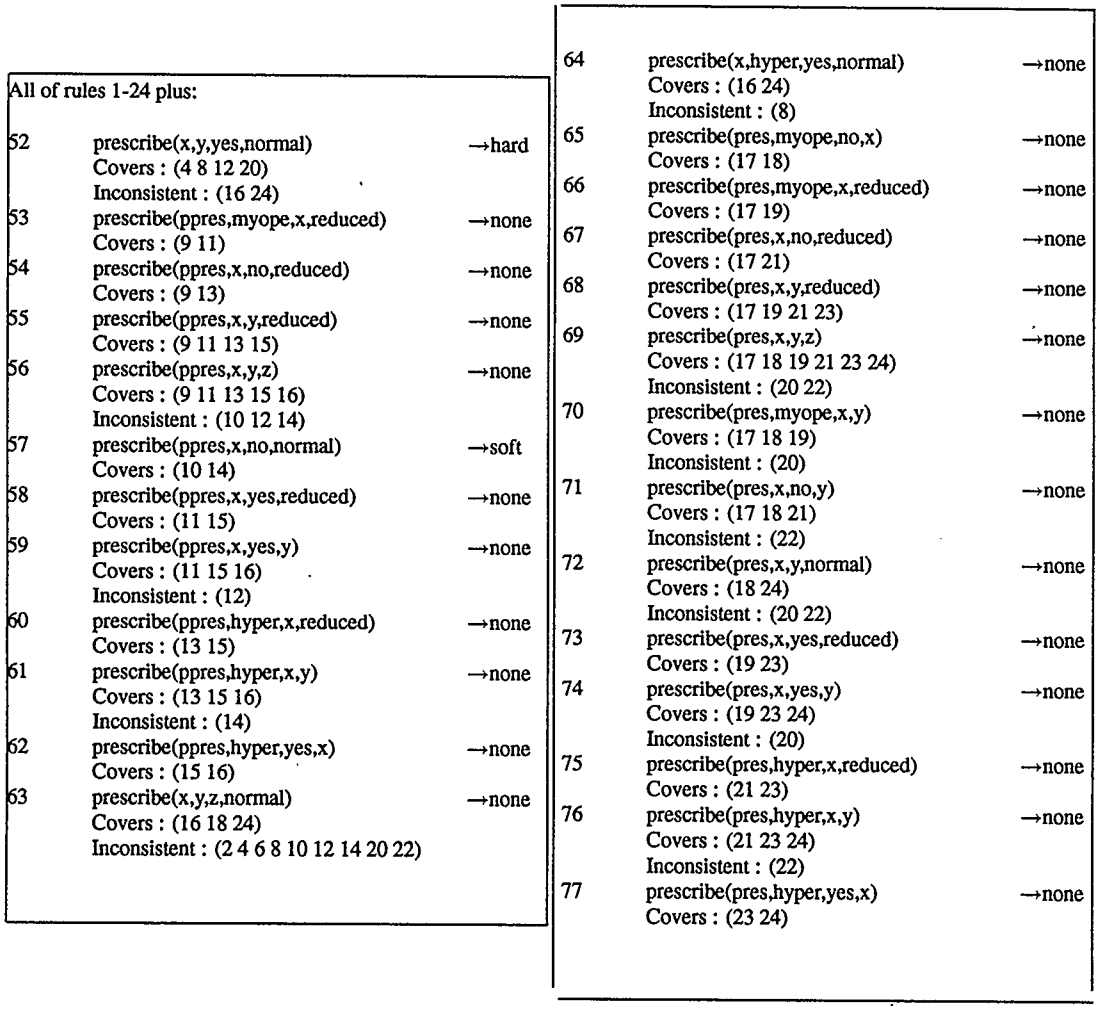
Output: *R* — a list of rules.

Figure 5.8. Balog/AMP: Algorithm

All of rules 1-24 plus:

25 prescribe(young,myope,x,reduced) →none
Covers : (1 3)
26 prescribe(young,x,no,reduced) →none
Covers : (1 5)
27 prescribe(young,x,y,reduced) →none
Covers : (1 3 5 7)
28 prescribe(x,myope,no,reduced) →none
Covers : (1 9 17)
29 prescribe(x,myope,y,reduced) →none
Covers : (1 3 9 11 17 19)
30 prescribe(x,y,no,reduced) →none
Covers : (1 5 9 13 17 21)
31 prescribe(x,y,z,reduced) →none
Covers : (1 3 5 7 9 11 13 15 17 19 21 23)
32 prescribe(x,y,z,w) →none
Covers : (1 3 5 7 9 11 13 15 16 17 18 19 21 23 24)
Inconsistent : (2 4 6 8 10 12 14 20 22)
33 prescribe(x,myope,no,y) →none
Covers : (1 9 17 18)
Inconsistent : (2 10)
34 prescribe(young,x,no,normal) →soft
Covers : (2 6)
35 prescribe(x,myope,no,normal) →soft
Covers : (2 10)
Inconsistent : (18)
36 prescribe(x,y,no,normal) →soft
Covers : (2 6 10 14 22)
Inconsistent : (18)
37 prescribe(young,x,yes,reduced) →none
Covers : (3 7)
38 prescribe(x,myope,yes,reduced) →none
Covers : (3 11 19)
39 prescribe(x,y,yes,reduced) →none
Covers : (3 7 11 15 19 23)
40 prescribe(x,y,yes,z) →none
Covers : (3 7 11 15 16 19 23 24)
Inconsistent : (4 8 12 20)
41 prescribe(x,myope,y,z) →none
Covers : (1 3 9 11 17 18 19)
Inconsistent : (2 4 10 12 20)
42 prescribe(young,x,yes,normal) →hard
Covers : (4 8)
43 prescribe(x,myope,yes,normal) →hard
Covers : (4 12 20)
44 prescribe(young,hyper,x,reduced) →none
Covers : (5 7)
45 prescribe(x,hyper,no,reduced) →none
Covers : (5 13 21)
46 prescribe(x,hyper,y,reduced) →none
Covers : (5 7 13 15 21 23)
47 prescribe(x,hyper,y,z) →none
Covers : (5 7 13 15 16 21 23 24)
Inconsistent : (6 8 14 22)
48 prescribe(x,y,no,z) →none
Covers : (1 5 9 13 17 18 21)
Inconsistent : (2 6 10 14 22)
49 prescribe(x,hyper,no,normal) →soft
Covers : (6 14 22)
50 prescribe(x,hyper,yes,reduced) →none
Covers : (7 15 23)
51 prescribe(x,hyper,yes,y) →none
Covers : (7 15 16 23 24)
Inconsistent : (8)

Figure 5.9. The depth zero antinarrowings the contact lens examples



All of rules 1-24 plus:

52 prescribe(x,y,yes,normal) →hard
Covers : (4 8 12 20)
Inconsistent : (16 24)

53 prescribe(ppres,myope,x,reduced) →none
Covers : (9 11)

54 prescribe(ppres,x,no,reduced) →none
Covers : (9 13)

55 prescribe(ppres,x,y,reduced) →none
Covers : (9 11 13 15)

56 prescribe(ppres,x,y,z) →none
Covers : (9 11 13 15 16)
Inconsistent : (10 12 14)

57 prescribe(ppres,x,no,normal) →soft
Covers : (10 14)

58 prescribe(ppres,x,yes,reduced) →none
Covers : (11 15)

59 prescribe(ppres,x,yes,y) →none
Covers : (11 15 16)
Inconsistent : (12)

60 prescribe(ppres,hyper,x,reduced) →none
Covers : (13 15)

61 prescribe(ppres,hyper,x,y) →none
Covers : (13 15 16)
Inconsistent : (14)

62 prescribe(ppres,hyper,yes,x) →none
Covers : (15 16)

63 prescribe(x,y,z,normal) →none
Covers : (16 18 24)
Inconsistent : (2 4 6 8 10 12 14 20 22)

64 prescribe(x,hyper,yes,normal) →none
Covers : (16 24)
Inconsistent : (8)

65 prescribe(pres,myope,no,x) →none
Covers : (17 18)

66 prescribe(pres,myope,x,reduced) →none
Covers : (17 19)

67 prescribe(pres,x,no,reduced) →none
Covers : (17 21)

68 prescribe(pres,x,y,reduced) →none
Covers : (17 19 21 23)

69 prescribe(pres,x,y,z) →none
Covers : (17 18 19 21 23 24)
Inconsistent : (20 22)

70 prescribe(pres,myope,x,y) →none
Covers : (17 18 19)
Inconsistent : (20)

71 prescribe(pres,x,no,y) →none
Covers : (17 18 21)
Inconsistent : (22)

72 prescribe(pres,x,y,normal) →none
Covers : (18 24)
Inconsistent : (20 22)

73 prescribe(pres,x,yes,reduced) →none
Covers : (19 23)

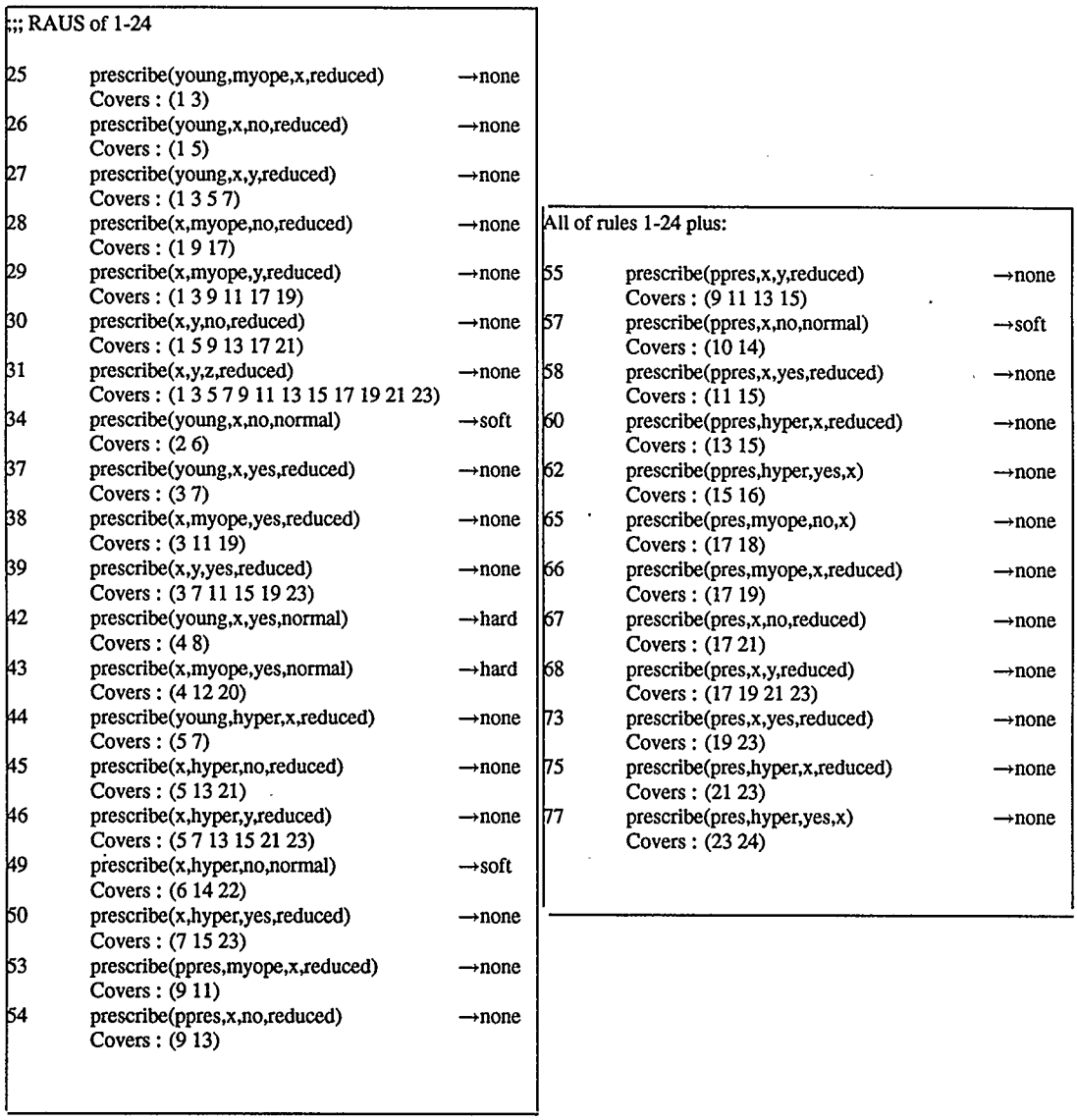
74 prescribe(pres,x,yes,y) →none
Covers : (19 23 24)
Inconsistent : (20)

75 prescribe(pres,hyper,x,reduced) →none
Covers : (21 23)

76 prescribe(pres,hyper,x,y) →none
Covers : (21 23 24)
Inconsistent : (22)

77 prescribe(pres,hyper,yes,x) →none
Covers : (23 24)

Figure 5.9. Continued.



;;; RAUS of 1-24

25 prescribe(young,myope,x,reduced) →none
Covers : (1 3)
26 prescribe(young,x,no,reduced) →none
Covers : (1 5)
27 prescribe(young,x,y,reduced) →none
Covers : (1 3 5 7)
28 prescribe(x,myope,no,reduced) →none
Covers : (1 9 17)
29 prescribe(x,myope,y,reduced) →none
Covers : (1 3 9 11 17 19)
30 prescribe(x,y,no,reduced) →none
Covers : (1 5 9 13 17 21)
31 prescribe(x,y,z,reduced) →none
Covers : (1 3 5 7 9 11 13 15 17 19 21 23)
34 prescribe(young,x,no,normal) →soft
Covers : (2 6)
37 prescribe(young,x,yes,reduced) →none
Covers : (3 7)
38 prescribe(x,myope,yes,reduced) →none
Covers : (3 11 19)
39 prescribe(x,y,yes,reduced) →none
Covers : (3 7 11 15 19 23)
42 prescribe(young,x,yes,normal) →hard
Covers : (4 8)
43 prescribe(x,myope,yes,normal) →hard
Covers : (4 12 20)
44 prescribe(young,hyper,x,reduced) →none
Covers : (5 7)
45 prescribe(x,hyper,no,reduced) →none
Covers : (5 13 21)
46 prescribe(x,hyper,y,reduced) →none
Covers : (5 7 13 15 21 23)
49 prescribe(x,hyper,no,normal) →soft
Covers : (6 14 22)
50 prescribe(x,hyper,yes,reduced) →none
Covers : (7 15 23)
53 prescribe(ppres,myope,x,reduced) →none
Covers : (9 11)
54 prescribe(ppres,x,no,reduced) →none
Covers : (9 13)

All of rules 1-24 plus:

55 prescribe(ppres,x,y,reduced) →none
Covers : (9 11 13 15)
57 prescribe(ppres,x,no,normal) →soft
Covers : (10 14)
58 prescribe(ppres,x,yes,reduced) →none
Covers : (11 15)
60 prescribe(ppres,hyper,x,reduced) →none
Covers : (13 15)
62 prescribe(ppres,hyper,yes,x) →none
Covers : (15 16)
65 prescribe(pres,myope,no,x) →none
Covers : (17 18)
66 prescribe(pres,myope,x,reduced) →none
Covers : (17 19)
67 prescribe(pres,x,no,reduced) →none
Covers : (17 21)
68 prescribe(pres,x,y,reduced) →none
Covers : (17 19 21 23)
73 prescribe(pres,x,yes,reduced) →none
Covers : (19 23)
75 prescribe(pres,hyper,x,reduced) →none
Covers : (21 23)
77 prescribe(pres,hyper,yes,x) →none
Covers : (23 24)

Figure 5.10. Consistent antiunifications of the contact lens examples

The second phase uses the same algorithm as Balog/AMP for hypothesis creation and testing. By default, the hypothesis increment function is coverage testing. (A command at the command-line can be used to change the increment function to be used). The result is shown in Figure 5.11. Balog/C generates 9 general rules that contain all the information in the 24 original ones.

```
Balog/C - Version 3.0

Phase 1. Candidate Rules
Compute Antinarrowings [0] (widenings=24 antinarrowings=24)
Augment Widenings      [1] (widenings=24 antinarrowings=77)
Removed (21) inconsistent rules.
Phase 2. Hypothesis Testing
Increment Method: Coverage

---- Induced Rules ----
(1) prescribe(pres,hyper,yes,x)  -> none
(2) prescribe(pres,myope,no,x)  -> none
(3) prescribe(ppres,hyper,yes,x)  -> none
(4) prescribe(ppres,x,no,normal)  -> soft
(5) prescribe(young,x,yes,normal)  -> hard
(6) prescribe(young,x,no,normal)  -> soft
(7) prescribe(x,hyper,no,normal)  -> soft
(8) prescribe(x,myope,yes,normal)  -> hard
(9) prescribe(x,y,z,reduced)  -> none
Elapsed Time:  50130 (real) 49302 (cpu)
[balog]
```

Figure 5.11. Balog/C: contact lens data results

5.3 Default Classification

The generation of default classification theories extends the usual type of classification that was described in the previous section.

Definition 5.1 *A* **default theory** *is an ordered sequence of rules.*

The notion of defaults comes in since rules earlier in the sequence can be viewed as exception cases while those later in the sequence can be seen as general defaults. Default theories are, in general, more readable and more compact than their equivalent coding as regular theories. This is accomplished by allowing inconsistencies into the rule base (the theory), and by ensuring that if there is a rule that is inconsistent with the examples, rules that override this inconsistency are placed earlier in the sequence of rules.

For example, a useful use of default rules is in the definition of the list membership function. Suppose $member(x, y)$, where x is a term and y is a list, returns *false* if x is not in y and *true* if it is in y. Then the following definition is possible:

$member(x,nil) \rightarrow false$
$member(x, x.y) \rightarrow true$
$member(x, z.y) \rightarrow member(x, y)$

Note that the second and third rules are in contention since all those that match the second also match the third. However, the second rule always gets tried first in any computation. Without default theories, this function can only be defined by using a condition placed on the second and third rules that tests for the equivalence of x and z. Adding such conditions to rewrite rules is currently a topic of intense study (eg. see Kaplan, 1984).

Balog/CD is a classification system that generates default theories. It differs from Balog/C in just two aspects. First, all rules created through antinarrowing in phase one, including inconsistent rules, are possible candidates for inclusion in hypotheses. In other words, Balog/CD produces size 0 inconsistent antiunifications. Thus, all the rules in Figure 5.9 are created in phase one for the contact lens example. Second, the hypothesis increment function is coverage, but with priority given to covering currently inconsistent examples. Hypotheses are built by starting with the most general cases and then adding exception cases.

For instance, the first rule that is chosen by Balog/CD is rule 42 since it covers the most example rules (fifteen).

32 $prescribe(x, y, z, w) \rightarrow none$

Covers : (1 3 5 7 9 11 13 15 16 17 18 19 21 23 24)

Inconsistent : (2 4 6 8 10 12 14 20 22)

The nine examples that are inconsistent with it are of no concern since they have not yet been covered by any rules in the hypothesis. Next, the algorithm chooses rule 36 since it covers five rules that have not been covered by rule 32.

36 $prescribe(x, y, no, normal) \rightarrow soft$

Covers : (2 6 10 14 22)

Inconsistent : (18)

Example rule 18 is inconsistent with rule 36, and has already been covered by rule 32. The next step is to make sure that rule 18 is covered, along with as many of the uncovered rules (4 8 12 20) as possible. It turns out that rules which cover rule 18 do not cover any in this list. However, rule 65 covers rules 17 and 18, so this rule is chosen in preference to the more specific rule 18.

65 $prescribe(pres, myope, no, x) \rightarrow none$

Covers : (17 18)

Note that if a rule is chosen later that causes an inconsistency with only rule 17, then rule 65 will be moved higher up in the list (since it does not introduce inconsistency itself), so that it will resolve both inconsistencies at once. The remaining four example rules that have yet to be covered are straightforwardly covered by rules 43 and 42. (Note that example 4 is covered by both of these). Figure 5.12 shows the result of Balog/CD on the contact lens data. While Balog/C creates nine rules on this data, Balog/CD produces only five. These five rules are more understandable — they do not need to contain as much detail to rule out inconsistencies.

```
Balog/CD - Version 3.0

Phase 1. Candidate Rules
Compute Antinarrowings [0] (widenings=24 antinarrowings=24)
Augment Widenings      [1] (widenings=24 antinarrowings=77)
Removed (0) inconsistent rules.
Phase 2. Hypothesis Testing
Increment Method: Coverage + Inconsistency Reduction

---- Induced Rules ----
(1) prescribe(young,x,yes,normal)  -> hard
(2) prescribe(x,myope,yes,normal)  -> hard
(3) prescribe(pres,myope,no,x)  -> none
(4) prescribe(x,y,no,normal)  -> soft
(5) prescribe(x,y,z,w)    -> none
Elapsed Time: 51716 (real) 50014 (cpu)
[balog]
```

Figure 5.12. Balog/CD: result on contact lens data

5.4 Learning With Noise

While Balog/CD allows inconsistency in the hypotheses that are generated, the more general Balog/PRU can also ignore noisy data. Balog/PRU is the same as Balog/AMP except that it allows inconsistent rules to be generated in phase one. Phase two can be configured to use a similar hypothesis increment function as Balog/CD, namely coverage increment with inconsistency coverage priority. Another alternative is to use one of the evaluation functions discussed in Section 5.1.2. In this case, hypotheses that include rules produced from noisy data will likely have poor evaluations – consistent hypotheses will be preferred.

```
[balog] showx
--- example base ---
(1) app(nil,a.nil)  -> a.nil
(2) app(nil,nil)  -> nil
(3) app(b.nil,nil)  -> b.nil
(4) app(c.nil,d.nil)  -> c.d.nil
(5) app(e.f.nil,g.nil)  -> e.f.g.nil
(6) app(a.nil,f.nil)  -> f.nil
```

Figure 5.13. Balog/PRU: Append examples with noise

Consider the example set in Figure 5.13, which adds a new rule, Rule 6, to the examples in Figure 5.4. Figure 5.14 illustrates the result of running Balog/PRU on this new set. While Rule 6 produces some extra rules in phase one, the normal definition of append is still generated since its rules cover more of the database than do the false generalizations. The generated theorem, namely $\forall x\ app(nil, x) \rightarrow x \wedge \forall x, y, z\ app(x.y, z) \rightarrow x.app(y, z)$ is a prudent theorem of the theory specified in Figure 5.13.

```
[balog] balog/pru

Balog/PRU - Version 3.0

Phase 1. Candidate Rules
Compute Antinarrowings [0] (widenings=6 antinarrowings=6)
Augment Widenings      [1] (widenings=6 antinarrowings=12)
Compute Antinarrowings [1] (widenings=55 antinarrowings=12)
Augment Widenings      [2] (widenings=55 antinarrowings=17)
Compute Antinarrowings [2] (widenings=55 antinarrowings=17)

Antinarrowings
--------------
1 [0] app(nil,a.nil)  -> a.nil
2 [0] app(nil,nil)  -> nil
3 [0] app(b.nil,nil)  -> b.nil
4 [0] app(c.nil,d.nil)  -> c.d.nil
5 [0] app(e.f.nil,g.nil)  -> e.f.g.nil
6 [0] app(a.nil,f.nil)  -> f.nil
7 [1] app(nil,x)  -> x
From widenings: ((1 2) (2 7))
8 [1] app(x,y.nil)  -> y.nil
From widenings: ((1 6) (6 8))
9 [1] app(x,nil)  -> x
From widenings: ((2 3) (3 9))
10 [1] app(x,y)  -> y
From widenings: ((2 6) (2 8) (6 7) (6 10) (6 12))
11 [1] app(x.nil,y)  -> x.y
From widenings: ((3 4) (4 3) (4 11))
12 [1] app(x,y)  -> x
From widenings: ((3 10))
13 [2] app(x.nil,y)  -> app(nil,x.y)
From widenings: ((22 16))
14 [2] app(x.nil,y)  -> x.app(nil,y)
From widenings: ((23 15))
15 [2] app(x.nil,y)  -> x.app(y,nil)
From widenings: ((26 15))
16 [2] app(x.y,z)  -> x.app(y,z)
From widenings: ((44 15))
17 [2] app(x.y,z.nil)  -> x.app(y,z.nil)
From widenings: ((44 23))
Phase 2. Hypothesis Testing
Method: Inconsistency Reduction
Elapsed Time: 23811 (real) 23150 (cpu)

---- Induced Rules ----
(1) app(nil,x)  -> x
(2) app(x.y,z)  -> x.app(y,z)
[balog]
```

Figure 5.14. Balog/PRU: Learning append even with noise

5.5 Inductive Theorem Generation

Balog/IND is the subsystem of Balog that generates inductive theorems. The Balog/IND algorithm is given in Figure 5.15 and an example of its operation is shown in Figure 5.16.

Balog/IND - Summative Induction only algorithm.

Input: X — example base
$Depth$ — maximum antinarrowing size
$Limit$ — maximum antiunification passes
$SpecLevels$ — number of levels to specialize.

$R = PhaseTwo(PhaseOne(PhaseZero(X,SpecLevels),$
$Depth, Limit, Valid),$
$False, Ind\text{-}proof\text{-}check)$

Output: R — a list of rules.

Figure 5.15. Balog/IND: Algorithm

Balog/IND generates inductive theorems, the subclass of ampliative theorems that are totally justified. In Balog/AMP, pairs of rules were antinarrowed to create biexemplar justified ampliative theorems. Similarly, we might expect that the best way to create inductive theorems would be to antiunify *sets* of rules, checking to see that a variable is introduced when a cover set is formed by the terms it replaces with variables. In fact, this describes the *Irau* operator from Section 4.4.1. However, checking for antiunifications of sets of widenings can be very expensive. Instead, Balog/IND generates biexemplar justified rules in the manner of Balog/AMP and then, using the *Ind-proof-check* function, determines those that are cover set justified and thus inductive theorems. *Ind-proof-check* implements cover-set induction (Section 3.4).

```
[balog] showx
--- example base ---
(1) app(nil,x)  -> x
(2) app(x.y,z)  -> x.app(y,z)
[balog] balog/ind

Balog/IND - Version 3.0

Phase 0. Computing Specializations
Level (0) Specs (2)
Level (1) Specs (6)
Computed (6) specializations:
(1) app(nil,nil)  -> nil
(2) app(nil,x.y)  -> x.y
(3) app(x.nil,y.z)  -> x.y.z
(4) app(x.nil,nil)  -> x.nil
(5) app(x.y,nil)  -> x.app(y,nil)
(6) app(x.y,z.w)  -> x.app(y,z.w)
Phase 1. Candidate Rules
Compute Antinarrowings [0] (widenings=6 antinarrowings=6)
Augment Widenings      [1] (widenings=6 antinarrowings=13)
Compute Antinarrowings [1] (widenings=30 antinarrowings=13)
Augment Widenings      [2] (widenings=30 antinarrowings=27)
Compute Antinarrowings [2] (widenings=30 antinarrowings=27)
Removed (2) inconsistent rules.
Phase 2. Check for inductive rules.
Elapsed Time: 44560 (real) 44140 (cpu)

---- Induced Rules ----
(1) app(x,nil)  -> x
```

Figure 5.16. Balog/IND example

Recall from Section 4.6 that the rule specialization operator (Section 4.5.1) is required for completeness. By default, it is not used in Balog/AMP since in practice, examples usually have no variables to specialize. However, many of the interesting inductive theorems require it — since in practice we wish to determine inductive theorems of theories, not examples — and it is implemented in a new phase, *Phase Zero*. A depth variable, *SpecLevels*, controls how many times it may be applied. For instance, if $SpecLevels = 1$ then only the rule specializations of the example base will be generated. However, if $SpecLevels = 2$ then the rule specializations of the rule specializations generated at specialization level 1 will also be generated. The output of phase zero becomes the input of phase one as if it were the original set of examples.

When Balog/IND is run on the append function (not the examples!) in Figure 5.13, the inductive theorem $\forall x\ app(x, nil) \rightarrow x$ is produced. Phase zero produces 6 specializations after one application of the rule specialization operator on the original two rules. These 6 rules are passed to phase one which produces 25 antinarrowings. In phase two, only one of the antinarrowings is determined to be a non-trivial inductive theorem.

Chapter 6

Other Machine Learning Systems

The theory developed so far can help to describe other important machine learning systems. In particular, Section 6.1 shows that a wide range of classification systems can be characterized through their use of specialization. Cigol, a system similar to Balog/AMP, is examined in Section 6.2 and several other machine learning systems are briefly discussed in Section 6.3.

6.1 Other Classification Systems

Most classification learning algorithms are not antiunification based like Balog. Instead, *specialization* forms the main computational mechanism. These induction systems begin with an overly general rule and specialize it until it is consistent with the example set (and itself). The main operational difference between ID3 and PRISM/INDUCT is not the specialization process, but what they do with the specializations.

The next two sections describe the algorithms of ID3 and PRISM in terms of rewrite rules. During their descriptions, suppose that the original set of examples is:

X:

$$\begin{bmatrix} f(a1,b1) \rightarrow c1 \\ f(a1,b2) \rightarrow c2 \\ f(a2,b3) \rightarrow c2 \\ f(a2,b1) \rightarrow c1 \\ f(a3,b3) \rightarrow c3 \end{bmatrix}$$

ID3 Algorithm

Input: C — set of classes $(c_1, c_2, ..., c_m)$.
X — example set such that
for all $r \in X$ there are n terms $t_1, ..., t_n$
and a $c \in C$ such that $r \equiv f(t_1, ..., t_n) \rightarrow c$

$V = \{x_1, x_2, ..., x_n\}$, a set of variables, such that *Typeof*(x_i) = *Typeof*(f/i)
$S = \{f(x_1, x_2, ..., x_n) \rightarrow z \mid z \in C\}$
$F = \emptyset$
While $(S \neq \emptyset)$ {
$S = DelExtras(Best(Nspec(S, V), X), X)$
$F = \{r \mid r \in S \wedge Consistent(r, S \cup X)\}$
$S = S - F$
}
Output: F — a set of rules

Figure 6.1. ID3

6.1.1 ID3 as a specialization system

ID3 (Quinlan, 1986) is an algorithm that has often been implemented in commercial systems. Its popularity is due to its simplicity, speed, and power for learning classification rules from data. Figure 6.1 shows the ID3 algorithm that is implemented in the Balog system.

Section 4.5 distinguished between a "naive" type of specialization and a more comprehensive type. This algorithm, in its use of the *Nspec* operator, shows that the central operator in ID3 is *naive rule set specialization*. This immediately suggests that ID3 can be improved by dropping the *naive* restriction and using the *rule set specialization* operator *Spec* instead.

Let us work through an example to demonstrate this algorithm. ID3 is given a set of rules, X, and a set of classes, C. We use the set of five rules listed above to be X and $\{c1, c2, c3\}$ for C. Then ID3 determines the most general specialization, S, that will be subsequently specialized. In the first line of the algorithm, $V = \{x, y\}$ and so $S = \{f(x, y) \rightarrow c1, f(x, y) \rightarrow c2, f(x, y) \rightarrow c3\}$. We write S more compactly as $\{f(x, y) \rightarrow z\}$

where z can be any class constant in C. On entering the while loop, the first step is to check if S is empty. Rules in S are ones that are currently inconsistent with the examples, and remain to be specialized further. Since S is not yet empty, the loop body is entered. Then ID3 calculates the naive rule specialization of all rules in S with respect to variables in V. $Nspec(S, V) = \{S1, S2\}$ where $S1 = \{f(a1, y) \rightarrow z, f(a2, y) \rightarrow z, f(a3, y) \rightarrow z\}$ and $S2 = \{f(x, b1) \rightarrow z, f(x, b2) \rightarrow z, f(x, b3) \rightarrow z\}$. Each of $S1$ and $S2$ specify nine rules. Next, the function *Best* chooses the best specialization in $Nspec(S, V)$ using a coverage heuristic. ID3 chooses S2. The function *DelExtras* removes rules with no instances in X. Thus S is updated to the set $\{f(x, b1) \rightarrow c1, f(x, b2) \rightarrow c2, f(x, b3) \rightarrow c2, f(x, b3) \rightarrow c3\}$. Next F is updated to the set $\{f(x, b1) \rightarrow c1, f(x, b2) \rightarrow c2\}$ since both of these rules are consistent with X. S is updated to $S - F$ leaving the inconsistent rules $\{f(x, b3) \rightarrow c2, f(x, b3) \rightarrow c3\}$. Since S is still not empty, the while loop is entered again. This time, $Nspec(S, V) = \{S1\}$ where $S1 = \{f(a1, b3) \rightarrow c2, f(a2, b3) \rightarrow c2, f(a3, b3) \rightarrow c2, f(a1, b3) \rightarrow c3, f(a2, b3) \rightarrow c3, f(a3, b3) \rightarrow c3\}$. Since there is only one rule set specialization, $S1$, $Best(\{S1\}) = S1$. Next, $S = DelExtras(S1, X) = \{f(a2, b3) \rightarrow c2, f(a3, b3) \rightarrow c3\}$ because these two rules are the only ones with instances in X. Now, both of the rules in X are consistent with X, and so F is updated to $\{f(x, b1) \rightarrow c1, f(x, b2) \rightarrow c2, f(a2, b3) \rightarrow c2, f(a3, b3) \rightarrow c3\}$ and S is updated to $\emptyset$. ID3 terminates and returns F as its final result.

6.1.2 Prism and Induct as Specialization Algorithms

Prism (Cendrowska, 1988) improves upon ID3 to create a more general set of rules. While ID3 maintains a set of rules that are specialized simultaneously, Prism specializes only a single rule at a time. The final specialization is produced one rule at a time rather than all at once like ID3. This allows each specialization, a single rule, to be evaluated on its own merits rather than in combination with other rules. Figure 6.2 shows the Prism algorithm. This time, the *naive rule specialization* operator, $Nspec(r, V)$, is used rather than the *naive rule set specialization* operator, $Nspec(S, V)$.

This algorithm considers each class one at a time; ID3 considers them all concurrently. It sets the most general rule possible for a class and finds the best specialization of it that is consistent with the examples not yet covered by a generated rule for the class.

PRISM Algorithm.

Input: X — example set
C — set of classes $(c_1, c_2, ..., c_m)$.

$F = \emptyset$
$SavedX = X$
while (more unprocessed classes) {
choose unprocessed $c \in C$; mark it as processed
$s = f(x_1, x_2, ..., x_n) \rightarrow c.$
$X = SavedX$
while(s has instances in X) {
while ($\neg$*Consistent*$(\{s\}, X)$).
$\{s\}$ =*Best*$(\{\{r\} | r \in \cup$*Nspec*$(s, V)\}, X)$
Remove instances of s from X.
$F = F \cup \{s\}$
$s = f(x_1, x_2, ..., x_n) \rightarrow c.$
}}
Output: F — a set of rules

Figure 6.2. PRISM

Using the same example set X and set of classes C as used to demonstrate ID3, we illustrate the operation of PRISM. First, we choose an unprocessed class arbitrarily. This time let c be c_1. Next, let s be the rule $f(x, y) \rightarrow c1$. Since s has two instances in X, namely $r1$ and $r4$, the second while loop is entered. We see that s is not consistent with rules $r2$, $r3$ and $r5$ since the left hand side of s matches the left hand sides of these rules — for example, $f(x, y)$ matches $f(a1, b2)$ of rule $r2$ — but the right hand sides of these rules are different from $c1$. Thus the innermost while loop is entered. Next, the union of the specializations of s is computed: $\cup$ *Nspec*$(s, \{x, y\}) = \{f(a1, y) \rightarrow c1, f(a2, y) \rightarrow c1, f(a3, y) \rightarrow c1, f(x, b1) \rightarrow c1, f(x, b2) \rightarrow c1, f(x, b3) \rightarrow c1\}$. Then, one specialization is chosen from among these rules using the function *Best*. Prism chooses $f(x, b1) \rightarrow c1$ because it covers both of rules $r1$ and $r4$. Since the left hand side of this new s, $f(x, b1)$, covers only the left hand sides of $r1$ and $r4$, and since both of these rules have right hand sides $c1$, the specialization s is now consistent with X. The inner while loop is exited, $r1$ and $r4$ are removed from X leaving $X = \{r2, r3, r5\}$, and F is updated

to $\{f(x,b1) \rightarrow c1\}$. s is reset to the general rule $f(x,y) \rightarrow c1$. Now s no longer has any instances in the updated X, so the second while loop is exited. We choose another unprocessed c. Suppose the algorithm chooses $c2$. This pass through the loop produces two rules $\{f(x,b2) \rightarrow c2, f(a2,y) \rightarrow c2\}$ which are added to F. The last pass, using $c = c3$, produces the rule $f(a3,y) \rightarrow c3$. At the very end, after the three classes have been processed, F is the set of rules $\{f(x,b1) \rightarrow c1, f(x,b2) \rightarrow c2, f(a2,y) \rightarrow c2, f(a3,y) \rightarrow c3\}$.

The Induct algorithm (Gaines, 1991) is similar except that X is not reset to the original set of examples on each pass through the outer while loop. In this example, Induct would produce: $\{f(x,b1) \rightarrow c1,\ f(x,b2) \rightarrow c2,\ f(x,y) \rightarrow c3\}$. It also uses a different formulation of the *Best* function, but the difference is not relevant to basic classification learning.

6.2 Cigol

The Cigol system (Muggleton & Buntine, 1988) adopts a similar approach to the theorem generation of Balog/AMP. While Balog/AMP inverts narrowing, since its representation language is equational theories, Cigol inverts Horn clause resolution, since it generates Prolog programs. Another difference is that no semantics is given for Cigol's operators. Nevertheless, we can describe its operators in terms of the theory developed in this thesis. However, conditional rewrite systems must first be reviewed to facilitate the explanation of Cigol's operators.

6.2.1 Conditional rewrite systems

Dealing with conditionals in basic term rewriting systems has been a major problem for theorem proving researchers (Kaplan, 1984). Instead of modifying the syntax of term rewriting systems, we can introduce a function, *if*, that acts like a conditional:

$$\begin{aligned} \mathit{if}(\mathit{true}, x, y) &\rightarrow x \\ \mathit{if}(\mathit{false}, x, y) &\rightarrow y \end{aligned}$$

Unfortunately, this method results in inefficient theorem proving since the first argument is not always evaluated first[1] resulting in the evaluation of x or y or both. It is also undesirable for theorem generation for a few reasons. Since the right hand side of both rules is a variable, it can be used to widen any rule at any subterm in the rule. Also, widening with these rules introduces new variables.

The most popular method of handling conditionals has been to use *conditional rewrite rules*. These are basic rewrite rules with a conditional part added on.

$$L1 \rightarrow R1 \Leftarrow C1 \wedge C2 \wedge ... \wedge Cn$$

Before $L1$ is allowed to rewrite to $R1$, all of $C1$ to Cn will have to first be rewritten to the constant function symbol $true$.

While conditional rules are more convenient to use than basic rewrite rules, they are no more expressive. It is always possible to write a set of conditional rules as a set of rules without conditions (Zhang, 1988).

To extend antiunification to conditional rules, it is necessary to specify how the conditional part of the rule is to be generalized. Basically, given two conditional rules, conjuncts that are not common to both rules are dropped, and common conjuncts are antiunified. Also, subterms that are antiunified to variables in the rewrite part of the conditional rule that also occur in the conditional part may need to be assigned to the same variables in the generalization. For example, antiunifying $f(a) \rightarrow b \Leftarrow g(a)$ and $f(c) \rightarrow b \Leftarrow g(c)$ should produce the antiunification $f(x) \rightarrow b \Leftarrow g(x)$ rather than $f(x) \rightarrow b \Leftarrow g(y)$. To develop a correct antiunification method for conditional rules is not difficult, but requires a treatment beyond the scope of this thesis.

[1] Lazy evaluation methods must be employed to force execution of the first argument of an *if* term before its other arguments.

6.2.2 Operators in Cigol

There are three main operators in Cigol: the *truncation operator*, the *absorption operator* and the *intraconstruction operator*. The truncation operator is simply the term antiunification operator[2].

To describe the absorption operator in terms of our theory, we will first require functions to convert between clause form and term form. Assume that these have been developed such that $Termify(\{p(x), \neg q(x), \neg r(x)\}) = clause(p(x), q(x), r(x))$ and $Clausify(clause(p(x), q(x), r(x))) = \{p(x), \neg q(x), \neg r(x)\}$.

Absorption

$C = \{t_1, \neg t_2, \neg t_3 ... \neg t_n\}$ (a Horn clause)
$\{s\}$ (a unit clause)

$Clausify(c \uparrow \alpha)$
where $c = Termify(C \cup \{\neg s\}\theta_1)$
$Domain(\alpha) \subset Pos(c) - \{\epsilon\}$
$\theta_1 = Match(t_1, s)$

Using this formulation, we immediately see that the absorption operator computes antiapplications. In general, there will be more than one, since the clause c will in general have many positions in which constants can be replaced with variables.

The *intra-construction operator* is the operator that does *constructive induction* in Cigol. For our presentation here, we simply convert Cigol's Horn clauses into conditional rewrite rules. Zhang (1988) developed an algorithm for doing this. Then the operator can be defined as follows:

[2]This illustrates the contention of Chapter 1 that a common language for these issues has not been developed or used.

intra-construction operator

$$p_1 \rightarrow true \Leftarrow q_1 \wedge q_2 \wedge ... \wedge q_m$$
$$p_2 \rightarrow true \Leftarrow r_1 \wedge r_2 \wedge ... \wedge r_m$$

$$p \rightarrow true \Leftarrow A \wedge f$$
$$t\theta_1 \rightarrow true$$
$$t\theta_2 \rightarrow true$$

where $p = Au(p_1, p_2), \theta_1 = Mgu(p_1, p), \theta_2 = Mgu(p_2, p)$
$A = r_1 \wedge ...r_m \wedge q_1 \wedge ...q_n$
$t = f(v_1, \ldots, v_m)$
$V = \cup\, Domain(\theta_i)$
$\{v_1, \ldots, v_m\} = V - \{v | v \in V, Vars(v\theta_j) \notin Range(\theta_j) - \{v\theta_j\} \text{ for } j = 1 \text{ or } j = 2\}$

The symbol f is the newly constructed function of arity m. The contruction of $\{v_1, \ldots, v_m\}$ ensures that f is kept as simple as possible (smallest possible arity). The terms excluded from V are those that do not change in different instantiations of f. Applying the intra-construction operator to the rules $min(x, s(x).y) \rightarrow true \Leftarrow min(x, y)$ and $min(x, s(s(x)).y) \rightarrow true \Leftarrow min(x, y)$ results in the set of rules $\{min(x, z.y) \rightarrow true \Leftarrow min(x, y) \wedge f(x, z), f(x, s(x)) \rightarrow true, f(x, s(s(x))) \rightarrow true\}$. Note that the construction of f did not include a place for y, since y is irrelevant to what makes $min(x, s(x).y)$ and $min(x, s(s(x)).y)$ different. The constructed function in this case, f, is the common less than ($<$) function.

6.3 Other Systems

There are several other important machine learning systems. Unfortunately, investigating all of them in terms of the theory in this thesis would require a lengthy treatise. Nonetheless, a few comments on some of these would be interesting.

Marvin (Sammut & Banerji, 1986) and Alvin (Krawchuk & Witten, 1988) are systems that learn Horn clauses by asking a teacher questions. The process of generating questions can be viewed as hypothesis specialization and might be described by using a form of the *Spec* operator. *Direct questioning* in Alvin can be seen as the presenting of a set of specializations,

```
                                                ack(0,0)   -> 1
                                                ack(0,1)   -> 2
ack(0,m)          -> s(m)                       ack(1,0)   -> 2
ack(s(n),0)       -> ack(n,1)                   ack(1,1)   -> 3
ack(s(n),s(m))    -> ack(n,ack(s(n),m))         ack(1,2)   -> 4
                                                ack(2,0)   -> 3
                                                ack(2,1)   -> 5
```

Figure 6.3. Ackermann's function and examples needed to generate it

whereas the *indirect questioning* style of both Alvin and Marvin can be seen as presenting single specializations. The *crucial objects* of Alvin can be seen as specializations unique to the hypothesis being tested (no other hypotheses would have these particular specializations). Both Marvin and Alvin use *elaboration* as one of their central algorithms. It might be described in terms of some form of the widening operator.

Version spaces (Mitchell, 1982) are simply lattices of rules. We might provide a general semantics for these spaces by simply stating that a version space is the lattice of prudent theorems under the prudent modelling relation. This would be a very general statement provided that we port the notion of prudent theorems to representation systems other than term rewriting systems. The candidate elimination algorithm for learning in version spaces can be seen as doing specialization of its G sets, and antinarrowing of its S sets.

MIS, the Model Inference System (Shapiro, 1983), learns Prolog programs through a process of directed (by a teacher) specialization with respect to negative examples. Shapiro provides a semantics of the process as being *(standard) model identification.* It would be interesting to apply his methods to non-standard models.

Finally, function induction systems might also be described in terms of this theory. In particular, the building of expressions can be seen as an application of antinarrowing. Indeed, the term rewriting system learning developed in this thesis can directly provide both model-theoretic and operational semantics to many function induction systems. In fact, Balog can be seen as a way of inducing general recursive functions. For example, Balog/AMP generates the celebrated Ackermann's function when given the seven simplest examples of it. The function and the required examples are given in Figure 6.3.

Chapter 7

Conclusions

This thesis has

- described induction;
- introduced a model theoretic semantics;
- produced some proof and generation methods for induction;
- developed programs for induction;
- described existing programs for induction.

In this final chapter, we summarize how the problems presented in Chapter 1 have been addressed. Recall the first problem:

Problem 1. *What is meant by the term "induction"? More precisely, how can induction be formalized in logic in a clear way, capturing the intended meaning of machine learning and automated theorem proving researchers?*

The semantics of Chapter 2 provided a solution to this problem. When we say that a system is doing "induction", we can say instead that it generates inductivelike theorems. Induction as conceived by theorem proving researchers was captured by noting that they prove theorems in the *IND* class — that is, theorems defined with inductivelike modelling over standard models and using total justification. A particular type of inductive proof called cover set induction was reviewed in Chapter 3. On the other hand, machine learning aims at generating prudent theorems, although most systems so far have only investigated ampliative ones. Ampliation, that is, induction over statements whose truth is not known, was given a model theoretic semantics by the development of the class of ampliative theorems (AMP). Ignoration, that is, induction despite noise, is embodied in ignorative theorems (IGN). Prudent induction, or induction despite noise and unknown values, is embodied in

prudent theorems (PRU). In the development of these classes, two new truth values, namely underdetermined and overdetermined, were introduced so that statements whose truth values are not determined to be true or false could be manipulated explicitly within a logical setting. By also applying preferencing techniques that are used in the study of nonmonotonic logics, these classes of theorems were given a precise semantics.

Problem 2. *How can inductive theorems be generated? Can some existing theorem proving techniques be adapted to generate theorems? In particular, how might such theorems be generated in an equational theory?*

Chapter 4 dealt with this problem in detail. The general idea was to reverse theorem proving techniques to obtain corresponding generation techniques. From term unification and substitution application, term antiunification and antiapplication were developed. Rewriting was reversed to obtain expansion. Equational unification was reversed to obtain E-antiunification for use with explicit background theories. Narrowing was reversed to obtain antinarrowing which can be used with implicit background theories. To generate all biexemplar justified theorems, not just the strictly biexemplar justified ones, the complete antinarrowing operator was designed. Next, cover set induction was reversed to obtain the inductive antinarrowing operator for generating inductive theorems. Finally, specialization was formalized and shown to be an inverse of antiunification. Specialization, together with inconsistent complete antinarrowing were suggested to be complete for generating a restricted subclass of prudent theorems. By using these operators, it becomes possible to generate inductivelike theorems.

Specific algorithms employing these operators were presented in Chapter 5. These included the use of antinarrowing in Balog/AMP and Balog/PRU, and rule antiunification in Balog/C and Balog/CD. By using antinarrowing and specialization, Balog/IND generated ampliative theorems and then tested them for inductive theoremhood using cover set induction as an alternative to using the inductive antinarrowing operator. In Chapter 5, we also developed some heuristics to choose interesting theorems from among the generated theorems. In Chapter 6, we showed that the theorems generated by other machine learning programs could be seen as using rule specialization, antinarrowing, or antiapplication.

Hopefully, the broad theory presented in this thesis aids the description of the plethora of approaches in machine learning and adds to the much-needed research on theorem finding.

Bibliography

Angluin, D. & Smith, C. (1983). Inductive inference: theory and methods. *Computing Surveys, 15*(3), 237–269.

Aristotle (1928). Prior analytics. In D. Ross (Ed.), *The Works of Aristotle*, volume 1. Oxford, UK: Oxford Press.

Avenhaus, J. & Madlener, K. (1990). Term rewriting and equational reasoning. In R. Banerji (Ed.), *Formal Techniques in Artificial Intelligence. A Sourcebook.* Elsevier Science.

Belnap, N. (1975). A useful four-valued logic. In J. Dunn & G. Epstein (Eds.), *Modern uses of multiple-valued logic* (pp. 8–37). D. Reidel.

Boyer, R. & Moore, J. (1979). *A computational logic.* New York: Academic Press.

Buntine, W. (1987). Induction of horn clauses: methods and the plausible generalization algorithm. *Int. J. Man Machine Studies, 26*, 499–519.

Burstall, R. (1969). Proving properties of programs by structural induction. *The Computer Journal, 12 (1)*, 41–48.

Carnap, R. (1962). *Logical Foundations of Probability.* Chicago, IL: Chicago University Press.

Cendrowska, J. (1987). Prism: An algorithm for inducing modular rules. *Int. J. Man Machine Studies, 27*, 349–370.

Dershowitz, N. (1982). Orderings for term rewriting systems. *J. Theoretical Comp. Sci., 17*, 279–310.

Dershowitz, N. & Sivakumar, G. (1989). Goal-directed equation solving. In *Proc. of the 7th Natl. Conf. on Artificial Intelligence*, (pp. 166–170)., San Mateo, CA. Morgan Kaufmann.

Dietterich, T. & Michalski, R. (1986). Learning to predict sequences. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach Vol 2* (pp. 63–106). Los Altos, CA: Morgan Kaufmann.

Fitting, M. (1990). *First-order logic and automated theorem proving.* New York: Springer Verlag.

Gaines, B. (1991). The tradeoff between knowledge and data in knowledge acquisition. In J. Boose & B. Gaines (Eds.), *Knowledge Discovery in Data Bases.* AAAI Press.

Genesereth, M. & Nilsson, N. (1987). *Logical Foundations of Artificial Intelligence.* Los Altos, CA: Morgan Kaufmann.

Heise, R. (1989). Demonstration instead of programming: focussing attention in robot task acquisition. Master's thesis, University of Calgary, Calgary, AB, Canada.

Helft, N. (1989). Induction as nonmonotonic inference. In R. Brachman, H. Levesque, & R. Reiter (Eds.), *Proceedings of the First Intl. Conf. on Principles of Knowledge Representation and Reasoning* (pp. 149–156). San Mateo, CA: Morgan Kaufmann.

Hintikka (1964). Towards a theory of induction generalization. In *Int. Congress for Logic Methodology and Philosophy of Science*, (pp. 47–90). North-Holland.

Hsiang, H. (1986). *Theorem Proving and Program Generation.* PhD thesis, Yale, Cambridge, MA.

Huet, G. & Oppen, D. (1980). Equations and rewrite rules: a survey. In R. Book (Ed.), *Formal Languages: Perspectives and Open Problems* (pp. 349–405). Academic Press.

Hullot, J. (1980). Canonical forms and unification. In Bibel, W. & Kowalski, R. (Eds.), *5th Conf. on Automated Deduction*, volume 87 (Lecture Notes in Computer Science), (pp. 318–334)., Berlin. Springer Verlag.

Kaplan, S. (1984). Conditional rewrite rules. *J. of Theoretical Comp. Sci., 33*, 175–193.

Kapur, D. & Musser, D. (1987). Proof by consistency. *Artificial Intelligence, 31*, 125–57.

Kapur, D. & Narendran, P. (1985). An equational logic approach to theorem proving in first-order predicate calculus. Technical report, General Electric Company, Schenectady, NY: GE Corporate Reaseach and Development.

Knuth, D. & Bendix, P. (1970). Simple word problems in universal algebras. In R. Leech (Ed.), *Computational Problems in Abstract Algebra* (pp. 263–297). Pergamon Press.

Kodratoff, Y. (1988). *Introduction to Machine Learning.* London: Pitman.

Kolmogorov, A. (1965). Three approaches to the quantitative definition of information. *Prob. Inf. Trans, 1*, 1–7.

Krawchuk, B. & Witten, I. (1988). On asking the right questions. In *Proc Fifth Int. Conference on Machine Learning*, (pp. 15–21)., San Mateo, CA. Morgan Kaufmann.

Krawchuk, B. & Witten, I. (1989). Explanation based learning: a problem solving approach. *J. Experimental and Theoretical Artificial Intelligence, 1*(1), 44–72.

Langholm, T. (1988). *Partiality, Truth and Persistence*. Stanford, CA: Center for the Study of Language and Information.

Lankford, D. (1981). A simple explanation of inductionless induction. In *Mechanical Theorem Proving*, volume 14, Ruston, LA. Louisiana Tech University.

Lassez, J., Maher, M., & Marriott, K. (1987). Unification revisited. In *Eighth Int. Conf. on Automated Deduction*, (pp. 67–113)., Berlin. Springer Verlag.

Lassez, J. & Marriott, K. (1987). Explicit representation of terms defined by counter examples. *J. of Automated Reasoning, 3*, 301–318.

Lloyd, J. (1984). *Foundations of logic programming*. Berlin: Springer Verlag.

Maulsby, D. L. & Witten, I. H. (1989). Inducing programs in a direct-manipulation environment. In *Human factors in computing systems: Proc. CHI '89*, (pp. 57–62)., Austin, Texas.

McCarthy, J. (1980). Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence, 13*, 27–39.

Michalski, R. (1983). A theory and methodology of inductive learning. *Artificial Intelligence, 20*, 111–161.

Mitchell, T. (1982). Generalization as search. *Artificial Intelligence, 18*, 203–226.

Mitchell, T., Keller, R., & Kedar-Cabelli, S. (1986). Explanation-based generalization: a unifying view. *Machine Learning, 1*(1), 47–80.

Muggleton, S. & Buntine, W. (1988). Machine invention of first order predicates by inverting resolution. In Laird, J. (Ed.), *Fifth Int. Conf. on Machine Learning*, (pp. 339–352)., San Mateo, CA. Morgan Kaufmann.

Phan, T. (1989). Equal value search. Master's thesis, University of Calgary, Calgary, AB, Canada.

Plotkin, G. (1970). A note on inductive generalization. *Machine Intelligence, 5*, 153–163.

Plotkin, G. (1971). A further note on inductive generalization. *Machine Intelligence, 6*, 101–124.

Popplestone, R. (1970). An experiment in automatic induction. *Machine Intelligence, 5*, 203–215.

Quinlan, R. (1986). Induction of decision trees. *Machine Learning, 1*(1), 81–106.

Quinlan, R. (1987). Generating production rules from decision trees. *Int. Joint Conf. on Artificial Intelligence, 1.*

Rescher, N. & Brandom, R. (1979). *The logic of inconsistency.* Totowa, New Jersey: Rowman and Littlefield.

Rety, P. (1987). Improving basic narrowing techniques. In *Rewriting Techniques and Applications*, (pp. 228–241)., Berlin. Springer Verlag.

Sammut, C. & Banerji, R. (1986). Learning concepts by asking questions. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine Learning: an Artificial Intelligence Approach Vol 2* (pp. 167–192). Los Altos, CA: Morgan Kaufmann.

Shapiro, E. (1983). *Algorithmic Program Debugging.* PhD thesis, Yale, Cambridge, MA.

Shoham, Y. (1988). *Reasoning about change.* Cambridge, MA: MIT Press.

Skyrms, B. (1975). *Choice and Chance: An Introduction to Inductive Logic.* Oxford, UK: Oxford Press.

Vere, S. (1977). Induction of relational productions in the presence of background information. In *Int. Joint Conf. on Artificial Intelligence*, (pp. 349–355)., Cambridge, MA. Morgan Kaufmann.

Vere, S. (1980). Multilevel counterfactuals for generalizations of relational concepts and productions. *Artificial Intelligence, 14*, 139–164.

Von Wright, G. (1957). *The Logical Problem of Induction.* Oxford, UK: Oxford Press.

Wos, L. (1988). *Automated Reasoning: 33 Basic Research Problems.* Englewood Cliffs, New Jersey: Prentice Hall.

Zhang, H. Kapur, D. & Krishnamoorthy, M. (1988). A mechanizable induction principle for equational specifications. In Lusk, E. & Overbeek, R. (Eds.), *9th Conf. on Automated Deduction*, (pp. 162–181)., Berlin. Springer Verlag.

Zhang, H. (1988). *Reduction, Superposition and Induction: Automated Reasoning in an Equational Logic".* PhD thesis, University of Iowa.

Appendix A

Proofs

Proposition 2.1 *Let S be the set of all standard models, let U be all underdetermined models, let O be all overdetermined models, and let F be all full models of a negationless theory $\mathcal{E}$. Then $S \subseteq U \subseteq F$ and $S \subseteq O \subseteq F$.*

Proof. We will show $S \subseteq U$. Proof of the other containments are analogous to this proof and pose no extra difficulty. First, choose a model $\mathcal{M}_s \in S$. We need only show that $\mathcal{M}_s \in U$. Now, the models in S are different than those in U, since the truth mapping of the former, $\mathcal{T}_s$ is two-valued, while that of the latter, $\mathcal{T}_l$, is three-valued. However, some models in U only map statements to **T** and **F**. So $\mathcal{M}_s \in U$ means that there is an underdetermined model $\mathcal{M}_s$ whose three-valued truth mapping that looks like a two-valued one in this way.

Now, we can construct an undetermined interpretation $\mathcal{M}_u$ from $\mathcal{M}_s$ such that the everything is the same, including the equational truth mapping, that is $G_s(e) = G_u(e)$, but the three valued truth mapping $\mathcal{T}_l$ is used instead of $\mathcal{T}_s$. To show that $\mathcal{M}_u$ is a model, we need only show that if $\mathcal{T}_s(w) = \mathbf{T}$ then $\mathcal{T}_l(w) = \mathbf{T}$ for all w. That is, we need to show that true things in the standard model are still true in the underdetermined one. Since the domain mapping function $\mathcal{K}$ is the same in $\mathcal{M}_s$ and $\mathcal{M}_u$, we really only need to show that $\mathcal{T}_2(w) = \mathbf{T}$ implies $\mathcal{T}_3(w) = \mathbf{T}$.

By each considering each possible syntactic form (except negation) of w, it will be shown that when w is **T** using $\mathcal{T}_2$ (see Figure 2.2), then w must also be true using $\mathcal{T}_3$ (see Figure 2.6). For each syntactic case, assume $\mathcal{T}_2(w) = \mathbf{T}$. Also, as an inductive hypothesis assume the proposition is true for W, A and B. That is, $\mathcal{T}_2(W) = \mathbf{T} \Rightarrow \mathcal{T}_3(W) = \mathbf{T}$, $\mathcal{T}_2(A) = \mathbf{T} \Rightarrow \mathcal{T}_3(A) = \mathbf{T}$, and $\mathcal{T}_2(B) = \mathbf{T} \Rightarrow \mathcal{T}_3(B) = \mathbf{T}$.

Case 1 (Base Case): $w \in \mathit{Eqns}(D)$. Since $\mathcal{T}_2(w) = \mathbf{T}$ is assumed, $G_s(w) = \mathbf{T}$. But since $G_s = G_u$, it directly follows that $\mathcal{T}_3(w) = T$.

Case 2: $w \equiv A \wedge B$. Since w is true under $\mathcal{T}_2$, both A and B are true under $\mathcal{T}_2$, and from the inductive hypothesis they are also true under $\mathcal{T}_3$. But then the three valued definition of $\wedge$ (Figure 2.3b) must assign w to true. So w is also true under $\mathcal{T}_3$.

Case 3: $w \equiv A \vee B$. Since w is true under $\mathcal{T}_2$, one of A and B are true under $\mathcal{T}_2$, and from the inductive hypothesis one is true under $\mathcal{T}_3$. But then the three valued definition of $\vee$ (Figure 2.3b) must assign w to true. So w is also true under $\mathcal{T}_3$.

Case 4: $w \equiv \exists x\, W$. Since w is true under $\mathcal{T}_2$, $\exists d \in D_{\mathit{Type}(x)}$ such that $\mathcal{T}_2(W\{x/d\}) = \mathbf{T}$.

Since $W\{x/d\}$ is an instance of W, it is true when W is true. But by the inductive hypothesis, W is true under $\mathcal{T}_3$, and so $W\{x/d\}$ is also true in $\mathcal{T}_3$. But then w must also be true in $\mathcal{T}_3$.

Case 5: $w \equiv \forall x\ W$. Since w is true under $\mathcal{T}_2$, $\forall d \in D_{Type(x)}$, $\mathcal{T}_2(W\{x/d\}) = \mathbf{T}$. Since all $W\{x/d\}$ are instances of W, they are true when W is true. But by the inductive hypothesis, $\mathcal{T}_3(W) = \mathbf{T}$ so $\mathcal{T}_3(W\{x/d\}) = \mathbf{T}$. But then w must also be true under $\mathcal{T}_3$.

Thus, regardless of the form of w, something true in $\mathcal{M}_s$ will be true in $\mathcal{M}_u$. Therefore $\mathcal{M}_s \in$ U, and then S $\subseteq$ U. The other inclusions pose no extra difficulties, except that $\times$ must be considered as well as **T** when considering O $\subseteq$ F.

Lemma 2.1 *Let S be the set of all standard models, let A be all avoidant preferred models, let K be all uncommitted preferred models, and let P be all uncommitted avoidant preferred models of a negationless theory $\mathcal{E}$. Then $S \subseteq A \subseteq P$ and $S \subseteq K \subseteq P$.*

Proof. Consider that P is the set of all preferred full models under the uncommitted avoidant preference relation $\sqsubset_{l\times}$. Proposition 2.1 shows that all undetermined models are full models. However, using $\sqsubset_{l\times}$ to prefer certain underdetermined models is exactly equivalent to using $\sqsubset_l$ on them since there are no $\times$ truth values in underdetermined models. Since A is the set of preferred underdetermined models with $\sqsubset_l$ and equivalently, the set preferred full models with $\sqsubset_{l\times}$, $A \subseteq P$. Similar considerations prove the other subset relations.

Proposition 2.2 *$IND(\mathcal{E}) \subseteq AMP(\mathcal{E}) \subseteq PRU(\mathcal{E})$ and $IND(\mathcal{E}) \subseteq IGN(\mathcal{E}) \subseteq PRU(\mathcal{E})$ for a negationless theory $\mathcal{E}$.*

Proof. Suppose $S \in IND(\mathcal{E})$ and $\mathcal{M}$ is a standard model of $\mathcal{E}$. Then Lemma 2.1 shows that $\mathcal{M}$ is also an uncommitted preferred model, an avoidant preferred model, and an uncommitted avoidant preferred model of $\mathcal{E}$. Also, $\mathcal{M}$ is an inductive modelling of $IND(\mathcal{E})$ (from the definition of inductive theorem). To show $IND(\mathcal{E}) \subseteq AMP(\mathcal{E})$ it is required that $\mathcal{M}$ is also an ampliative modelling of S. But since $\mathcal{M}$ is an underdetermined model, we need only show that S is biexemplary justified in $\mathcal{M}$. But since $S \in IND(\mathcal{E})$, S must be totally justified, and since anything totally justified is biexemplary justified, S is biexemplary justified as well. The remainder of the subset relations follow from Lemma 2.1 and the fact that S is biexemplary justified in $\mathcal{M}$ when it is totally justified.

Lemma 3.1 (Hullot, 1980) *Let $\mathcal{E}$ be an equational theory $\mathcal{E}$, R be a term rewriting system that is complete in $\mathcal{E}$, and s and t be two terms. The set of all solutions of a narrowing derivation of $s = t$ is complete.*

Proof. (Hullot, 1980).

Proposition 3.1 *Let $\mathcal{E}$ be an equational theory and R be a term rewriting system that is complete in $\mathcal{E}$. Then narrowing on R is a complete procedure for proving theorems in $DED(\mathcal{E})$, if $\mathcal{E}$ is conjunctionless and negationless.*

Proof. First note that before proving a theorem S, that all of its universal variables will have been skolemized out. Then that only leaves disjuncts, each of which have existentially quantified variables. We then apply Hullot's Lemma to each disjunct in turn. Since the whole theorem is true, at least one of its disjuncts will be true. Since the Lemma says that narrowing returns all solutions for existential variables in a disjunct, then if there is at least one solution, then narrowing will return it for a particular disjunct, and the whole theorem will be proven at that point.

Proposition 3.2 *Completion and narrowing together are complete for proving theorems in DED($\mathcal{E}$), for completable theories $\mathcal{E}$.*

Proof. A completion procedure produces a complete term rewriting system for a completable theory. So Proposition 3.1 can be applied.

Proposition 4.1 *Suppose s and t are terms and a is the antiunification of s and t, that is, $a = Au(s,t)$. If $Mgu(a,s) = \{v_1/s_1, v_2/s_2, ..., v_m/s_m\}$ then $\beta = Msa(s,t) = \{ Pos(v_1,a)/v_1, \ldots, Pos(v_m,a)/v_m, \}$*

Proof. To start, we assume Lassez, Maher and Marriott's (1987) proof that their algorithm, $Au(s,t)$, produces $s \uparrow Msa(s,t)$. To prove the relationship between *Msa* and *Mgu* given by this proposition, we must first show that β is an antisubstitution and that it is a most specific one. First, β is composed of antibindings (eg. $Pos(v_1,a)/v_1$). Also, $Pos(v_i,a)$ and $Pos(v_j,a)$ does not contain common positions since that would imply two different variables occupy the same position in a. So β is a disjoint antibinding. Since each position in a is a position in s, β is relevant to s. Also, the positions in β are also positions in t, because those positions are the places at which there are conflicts between s and t, and variables produced in a by the *Au* operator. So β is relevant to t as well and can be antiapplied to t to produce a. Then $t \uparrow \beta = a = s \uparrow \beta$ implies that β is an antiunifier.

Now suppose that β is not the most specific antiunifier of s and t. Then there is an $\alpha = \{Q_1/u_1, \ldots, Q_m/u_m\}$ and a proper substitution $\theta = \{v_1/m_1, \ldots, v_k/m_k\}$ such that $s \uparrow \alpha = (s \uparrow \alpha)\theta$. Then if $f = (s/Pos(v_i,s))/\epsilon$ and $g = (t/Pos(v_i,s))/\epsilon$, then $f \neq g$. But $Au(f(...), g(...))$ is always a variable and m_i is not a variable (since θ is a proper substitution), and so the variable v_i would have been at that position in a, not m_i. This contradiction shows that β must be the most specific antiunifier.

Proposition 4.2 *Define $Au(S)$ to be the* **antiunificand of a set of terms** *S computed by $Au(S)$ =$Au(s_1$,$Au(s_2, \ldots$,$Au(s_{n-1}, s_n)))$ for $s_i \neq s_j, 1 \leq i,j \leq n$ where $|S| = n$. $Au(S)$ is independent of the order of application of the pairwise Au function. Similarly, define $Msa(S)$ to be the* **most specific antiunifier of a set of terms** *S computed by $Msa(S) = Msa(s_1$,$Msa(s_2, \ldots$,$Msa(s_{n-1}, s_n)))$. $Msa(S)$ is also independent of the order of application of the pairwise Msa function.*

Proof. The proof of the *Au* case is by induction on the length of S. Base case: Suppose |S|

= 2 and $\{s_1, s_2\}$. Then we must prove the commutativity of *Au*, namely that $Au(s_1, s_2) = Au(s_2, s_1)$. Consider the algorithm to compute *Au* in Section 4.1.1. When $s_1 \equiv s_2$, s_1 will be returned whether it is given as the first or second argument to *Au*. If the terms do not match at their root function symbol, then either the variable v_{s_1,s_2} or the variable v_{s_2,s_1} is returned, depending on the order of arguments to *Au*. If s_1 and s_2 are subterms in two different positions of terms s and t respectively, then the same variable should be produced for both positions. This is accomplished by the second condition of the algorithm, which keeps the subterms of s as the first argument to *Au*. Otherwise, if they are both terms that start with the same function symbol, then that function is returned with a recursive application of *Au* to its arguments in each of the terms. This will be commutative if each of the applications of *Au* is commutative. Thus the pairwise *Au* operator is commutative.

Inductive case: Assume the order of application is independent for a set of length n. Consider when $|S| = n + 1$. Then $Au(S) = Au(s_1,Au(s_2,\ldots,Au(s_{n-1},Au(s_n, s_{n+1}))))$ The inductive hypothesis implies that $S - \{s_i\}$ is independent of the order of application of *Au*. Because of this, and from the commutativity of the pairwise *Au* operator, $Au(S) = Au(Au(S - \{s_i\}), s_i) = Au(s_i,Au(S - \{s_i\}))$. This means any element of S can be chosen first to be the first argument of the pairwise *Au* operator, and so the proposition is true for $|S| = n + 1$ as well.

By the principle of mathematical induction, the setwise *Au* operator can use the pairwise *Au* operator to compute *Au*(S), without needing to pairwise antiunify them in any particular order. By Proposition 4.1, this result extends to the *Mgu* operator as well.

Proposition 4.3 Expansion Soundness *Suppose that R is a complete term rewriting system for the equational theory $\mathcal{E}$. If $e \in DED(\mathcal{E})$ and $r \in R$, then $Ex(e, r) \in DED(\mathcal{E})$.*

Proof. Let $e \equiv L_1 = R_1$ and $r \equiv L_2 \rightarrow R_2$ and suppose that e is expanded on the term L_1 at position u with matching substitution θ to obtain the equation $Ex(e, r) \equiv L_1[u \leftarrow L_2]\theta = R_1$. Suppose that this equation is false. Then $L_1[u \leftarrow L_2\theta] = L_1$ must also be false since $R_1 = L_1$. Since the only position at which $L_1[u \leftarrow L_2\theta]$ and L_1 are different is u, then $(L_1[u \leftarrow L_2]\theta)/u \neq L_1/u$. Also, $L_2\theta \neq L_1/u$ (since $t[u \leftarrow s]/u = s$ for all s, t, u) and $R_2\theta \neq L_1/u$ (since $L_2 = R_2$) and $R_2\theta \neq Gnd(L_1/u)\theta$. But this means R_2 and $Gnd(L_1/u)$ are not unifiable. This contradictorily means that L_1 could not have been expanded by r at position u. Thus $Ex(e, r)$ must be true.

Proposition 4.4 Expansion Completeness *Suppose that R is a complete term rewriting system for the equational theory $\mathcal{E}$. Then $(a = b) \in DED(\mathcal{E})$ implies that $\exists t\ (a = b) \in Exrules(t \rightarrow t, R)$.*

Proof. Term rewriting proves an equation by reducing each side to its normal form to see if they are identical. That is, $(a = b) \in DED(\mathcal{E}) \Rightarrow Nf(a, R) \equiv Nf(b, R)$ if R is complete in $\mathcal{E}$. We must show that $a = b$ is an expansion of this unique normal form, which we call t in the proposition. Equivalently, we just need to show that $p \rightarrow q$ via R iff $p = Ex(q, R)$

since *Nf* uses only reduction steps and *Exrules* uses only expansion steps. Suppose that $p \rightarrow q$. Then $q = p[u \leftarrow R_i]\theta = p[u \leftarrow R_i\theta]$ where $p \in Pos(q)$, $\theta = Match(t/u, L_i)$, and $L_i \rightarrow R_i \in R$. By using the same θ and position u in the expansion case, we get that $p = q[u \leftarrow L_i]\theta = q[u \leftarrow L_i\theta]$. We check that expanding p then reducing back obtains p again. $p = q[u \leftarrow L_i\theta]$ and by substituting out q. $p = (p[u \leftarrow R_i\theta])[u \leftarrow L_i\theta]$. Since $(t[u \leftarrow s_1])[u \leftarrow s_2] = t[u \leftarrow s_1])$, then $p = p[u \leftarrow R_i\theta]$. But the position at u in p is exactly $R_i\theta$, otherwise no match would have occurred. So $p[u \leftarrow R_i\theta]$ is p itself and expanding, then reducing, obtains the same term. Similarly, $q = p[u \leftarrow R_i\theta] = (q[u \leftarrow L_i\theta])[u \leftarrow R_i\theta] = q[u \leftarrow L_i\theta] = q$. So $p \rightarrow q$ via R iff $p = Ex(q, R)$ and thus the proposition holds.

Proposition 4.5

1. If $S = Sp(g)$ and $|S| > 1$ then $Au(S) = g$
2. $Sp(Au(\{g\theta_1 ... g\theta_n\})) = \{\{g\theta_1 ... g\theta_n\}\}$ iff $\{\theta_1, ... \theta_n\}$ is a most general complete specializer of g.

Proof. (1) Suppose $S = Sp(g)$ and $|S| > 1$. Then $S = \{g\theta_i\}$ for $\theta_i = \{x/c_i\}$ where $x \in Vars(g)$ and $C = \{c_1, ..., c_n\}$ is a most general cover set for the type of x. But each term in C must be rooted with a different function symbol, that is $c_j/\epsilon \neq c_k/\epsilon$ (otherwise $c_j/\epsilon(x_1, ..., x_n)$, where n is the arity of c_j/ϵ, could replace both c_j and c_k in C making an even more general cover set). But then $Au(C)$ is a variable which can be renamed to x. Thus, $Au(S)$ is the term g (up to variable renaming).

(2) Suppose $\{\theta_1, ... \theta_n\}$ is a most general complete specializer of g. Consider some $x_i/t_i \in \theta_i$. There is an $x_j/t_j \in \theta_j$ where $x_i = x_j$. If this were not the case, then x_i/t_i could be removed from θ_i to obtain a more general specializer. Also $t_i/\epsilon \neq t_j/\epsilon$ otherwise $x_i/(t_i/\epsilon(v_1, ..., v_n))$, where n is the arity of t_i/ϵ, could replace x_i/t_i and x_j/t_j to obtain a more general specializer. But since $t_i/\epsilon \neq t_j/\epsilon$, *Au* will replace t_i and t_j with a variable which can be renamed to x_i. Thus, $Au(\{g\theta_1 ... g\theta_n\}) = g$ (up to variable renaming).

Now suppose $Sp(Au(\{g\theta_1 ... g\theta_n\})) = \{\{g\theta_1 ... g\theta_n\}\}$. Consider if $\{\theta_1, ... \theta_n\}$ were not a most general complete specializer of g. Then there is a most general specializer of g $\{\gamma_1, ... \gamma_m\}$ where $m \leq n$ such that each $g\theta_j = g\gamma_i\beta_i$ for some i. But then, *Sp* would have generated $g\gamma_i$ instead of $g\theta_j$ since *Sp* generates the most general instances of a term. Thus case (2) holds.

Proposition 4.6 *Rule specialization is not an ampliatively sound operator. That is,* $r \in AMP(R) \not\Rightarrow \forall s, s \in Spec(r, R) \Rightarrow s \in AMP(R)$

Proof. We show this by a counter example. Suppose $R = \{f(a) \rightarrow c, f(b) \rightarrow c\}$. Then $r = f(x) \rightarrow c$ is an inductive theorem of R. But the rule $f(c) \rightarrow c$ is a specialization of e, yet is not an ampliative theorem of R.

Proposition 4.7 *Rule specialization is an inductively sound operator. That is,* $r \in IND(R) \Rightarrow \forall s, s \in Spec(r, R) \Rightarrow s \in IND(\mathcal{E})(R)$

Proof. Suppose r is an inductive theorem. Then it is cover set justified, that is, each of its instances are deductive or inductive theorems. No matter which cover set was used to justify r, the most general cover set C can also be used to justify r. Then for all $c \in C, r\{x/c\}$ must be a deductive or inductive theorem of R. But a maximal complete specialization is a set of rules formed from C, so $s = r\{x/c\}$ for some $c \in C$. Then s is a deductive or inductive theorem of R.

Lemma 4.1 *If e is a conjunctive universal prudent theorem of a theory* $\mathcal{E}$ *with ground axioms R, then for each conjunct c of e,* $c \in A$ *where* $A = \{a | a \in Rau(S)$ *where* $S \subseteq Eau(R)\}$.

Proof. If c is a prudent theorem, then it must be biexemplar justified, that is, there are at least two $g_i = c\theta_i$ such that each g_i is true or biexemplar justified in all models. Then, from expansion completeness (Proposition 4.4), g_i must also be in $Exterms(R)$ if it is true, or in $Eau(R)$ if only biexemplar justified. Furthermore, from the definition of biexemplar justification, all g_i are most general terms that are "different" enough from each other so that $Rau(\{g_i\}) = e$. So $c \in A$.

Proposition 4.8 Prudential Completeness *If e is a conjunctive negationless universal prudent theorem of a prudent extension of a theory* $\mathcal{E}$ *with axioms R, then for each conjunct c of e,* $c \in (Can^i(Spec(R)))$

Proof. Suppose e is a (conjunctive negationless universal) prudent theorem of a prudent extension $\mathcal{E}$ with axioms R. Then c is also a prudent theorem of e since all conjuncts of a prudent theorem are prudent theorems. Let E be the rules of the prudent extension of $\mathcal{E}$, and let P be $E - R$. Then P contains only prudent theorems of some extension of $\mathcal{E}$ smaller than E. We will prove this proposition by induction on the length of P.

Consider when $|P| = 0$. $Spec(R)$ contains all specializations including R itself and all ground specializations of R. Then from Lemma 4.1, $c \in A$ where $A = \{a | a \in Rau(S)$ where $S \subseteq Eau(Spec(R))\}$. But then this is the Can^i operator applied to $Spec(R)$, using only expansions instead of all widenings. So $c \in (Can^i(Spec(R)))$.

Assume that the proposition is true for all extensions E where $|P| = n$, that is, that if c is a prudent theorem of E then $c \in (Can^i(Spec(R)))$. Then consider a new extension $E' = E \cup \{p\}$ where p is a prudent theorem of E. Then $|E' - R| = n + 1$. Supposing that c is a prudent theorem of E', then either c is a prudent theorem of E or c is a prudent theorem that uses p to biexemplar justify it. If the former, then by the inductive hypothesis, $c \in (Can^i(Spec(R)))$. If the latter, then there is a set of rules Q such that $Rau(Q) = c$ and there is some $p_w \in Q$ where p_w is a deductive theorem of E' (but not E). Now, $p \in (Can^i(Spec(R)))$ by the inductive hypothesis, because p is a prudent theorem of E. But then $p_w = Cwid(q_w, E \cup \{p\})$ for some $q_w \in DED(E)$ (by expansion soundness). Then, for

each $q \in Q$, including p_w, $q \in Cwid(Spec(R))$. But since $c = Rau(Q)$, $c \in Can^{i}(Spec(R)))$. By the principle of mathematical induction, the proposition holds for all sizes of P and thus for all prudent extensions of a theory.