1992

Learning Horn-Clauses as Classification Rules for Relations.

Mary Pamela Langley
Louisiana State University and Agricultural & Mechanical College

Order Number 9316980

Learning horn-clauses as classification rules for relations

Langley, Mary Pamela, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1992

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

LEARNING HORN-CLAUSES AS CLASSIFICATION RULES FOR RELATIONS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Mary Pamela Langley
B.S., McNeese State University, 1979
M.S., McNeese State University, 1982
December 1992

Table of Contents

List of Tables

List of Figures

Abstract

Explanation-based learning (EBL) has been applied numerous times in many different domains using as much background knowledge as possible to guide the learning process. We design and implement a new EBL based algorithm called FORGE (FOrming Rules from Ground Explanations) which operates utilizing limited background knowledge. The input to the FORGE algorithm consists of examples and counterexamples of the target concept in the form of ordered tuples along with example tuples of base relations. This limited input is used to construct explanation trees which produce ground rules. These ground rules are then generalized and evaluated using simple cover counts. Different heuristics for pruning the explanation trees are proposed and examined. The FORGE algorithm is compared to the empirically based FOIL algorithm and it is shown that the number of rules evaluated is often significantly reduced by utilizing the EBL methods of FORGE even in the limited knowledge based domain of FOIL.

Chapter 1

Classification and Explanation-Based Learning

1.1 Introduction

Concept learning has long been a principle area of machine learning research. Concept learning systems are supplied with information about entities whose class membership is known and produce from this information a characterization of each class. The characterizations produced are known as rules and are later used to classify entities or examples whose class membership is unknown. Concept learning is also known as learning from examples or classificaion, since one is learning rules to classify entities as examples or counterexamples of a concept.

The problem of classification has received much attention by the scientific community. There are two main types of classification problems under study today. One problem involves identifying groups of similar observations from other groups. The other problem is one of developing a description of a group when given known observations from the group.

The statistical community has developed procedures to handle each of these problems using observations of continuous type data. Cluster analysis is used to cluster like observations into groups [38]. Discriminant analysis [16], a method accredited to R.A. Fisher, uses observations from known groups to develop functions that will classify future observations whose group membership is unknown. Each of these statistical procedures were developed using continuous data and are effective in that domain. When the data consists only of nominal or ordinal type data, these procedures are not as effective.

Another deficiency of the current statistical procedures is the lack of simple classification rules that may be easily interpreted. These procedures produce statistical functions whose terms are almost always impossible to interpret. The shortcomings of these procedures lead to the re-emergence of the classification problems in the area of machine learning. This discipline has focused on developing procedures that produce simple, easily interpreted rules by using mostly nominal and ordinal type data.

One of the more popular procedures from the machine learning community has been developed by J.R. Quinlan and has been implemented in the ID3 system [31, 32]. ID3 uses decision trees to develop rules for defining a classification group when given both positive and negative examples from the group. Many advances have been made using the notion of decision trees as a means of producing classification rules [6, 28, 41]. Systems which employ decision trees use all given examples and counter examples of a concept in order to determine the common features in each. This type of learning is referred to as empirical learning or sometimes as similarity-based learning [10].

A more recent empirical learning approach, also proposed by J.R. Quinlan, is embodied in a system called FOIL (First-Order Inductive Learning). In this system, Horn-clause type rules are constructed and all examples and counter examples are used to evaluate and guide the rule building process [33]. Many improvements have been suggested for the FOIL system. Pazzani, Brunk, and Silverstein [29] have implemented FOCL which reduces the search of FOIL by using domain or background knowledge such as the types of the arguments involved. Richards and Mooney [34] have suggested a method of relational pathfinding aimed at improving the search of FOIL by viewing it as a hill climbing algorithm.

Another approach to the classification problem, which has been developed more recently than the empirical learning approach, is called Explanation-Based Learning (EBL) [10, 21, 24]. EBL focuses upon building an explanation of one example of a concept and then generalizing that explanation to cover more examples. The generalization process relies upon abundant domain knowledge to succeed in explaining only positive examples.

The system presented here, FORGE, FOrming Rules from Ground Explanations, utilizes the EBL paradigm within a FOIL type environment. The importance of FORGE is twofold. First it demonstrates the advantages of using EBL principles in place of the empirical learning method utilized in FOIL and secondly, it demonstrates an application of the EBL paradigm utilizing limited domain knowledge. The EBL paradigm is examined in more detail in the following section.

1.2 Explanation-Based Learning

As discussed above, approaches to learning from examples may be classified into two major categories. One being similarity-based learning or empirical learning, where multiple examples of a concept are examined in order to determine the features they have in common. The second major category of learning from examples and the one to which our learning algorithm adheres, is called explanation-based learning (EBL). In EBL generalizations are formulated after observing only a single example. EBL traditionally requires a great deal of domain knowledge to construct and generalize an explanation. We show that FORGE adheres to the EBL paradigm in some respects but successfully operates in an environment with limited domain knowledge.

The first step in all EBL-based systems is to build an explanation of a single input example. The second step involves generalizing the explanation derived in the first step. The generalizations preformed are usually justified by being explained in terms of the background or domain knowledge present thus requiring a great deal of domain knowledge. There may be more than one generalization produced from an explanation. In this case, if the domain knowledge doesn't provide a way of choosing from these generalizations, a method of choice must be adopted.

Mitchell [22] defines induction bias as any bias for choosing one generalization over another, other than strict consistency with the observed training instances. Dietterich [9] refines the definition of bias to include two distinct types. Declarative bias is defined in terms of direct statements about the domain enabling it to be evaluated before being used. On the other hand non-declarative bias can only be evaluated by testing consistency with examples since it cannot be immediately interpreted as a statement about the domain. Our algorithm employs both declarative and non-declarative types of bias when determining which generalization is preferred. Types of bias included in FORGE are discussed in more detail in Section 3.6.

EBL is a relatively new area of machine learning. Many different independently working researchers have produced results in a variety of different domains all using knowledge based learning from a single example. In an effort to unify these approaches, Michell, Keller and Kedar-Cabelli [24] have defined an approach called explanation-based generalization (EBG). Their specification of EBG is shown in Figure 1.1.

Given:

- *Target concept definition*: A concept definition describing the concept to be learned. (It is assumed that this concept definition fails to satisfy the operationality criterion.)
- *Training example*: An example of the target concept.
- *Domain theory*: A set of rules and facts to be used in explaining how the training example is an example of the target concept.
- *Operationality criterion*: A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed.

Determine:

- A generalization of the training example that is a sufficient concept description for the target concept and that satisfies the operationality criterion.

Figure 1.1. Mitchell et al.'s Specification of EBG.

Our algorithm adheres to the EBG formulization in the following ways. First the target concept definition is given as the set of target tuples in FORGE with the training example being a particular target tuple. The domain theory is the set of base relation tuples and the operationality criterion is to transform the learned concept into a set of Horn-clauses which may later be more easily utilized in determining the target concept.

One advantage in the representation chosen for FORGE is that the target concept definition, training example, domain theory and operationality criterion are so closely related in form. There is no costly computations involved in having to transform the target concept or the domain theory into a useful form.

Considering the specifications in Figure 1.1, one may ask in what sense does this represent learning since part of the input includes a definition of the target concept. Michalski [20] characterizes learning as "constructing or modifying representations of what is being experienced". The definition of the target concept in FORGE consists of simply a set of tuples belonging to the relation. This definition is nonoperational, meaning that it is almost impossible to use this form of the concept to recognize future instances of the concept. Our algorithm transforms the input concept definition, the set of target tuples, into a form that can be used as an efficient recognizer for the concept, a set of Horn-clauses.

One of the first systems developed which employed principles from EBL was STRIPS [12], a system for learning robot plans. STRIPS was developed in 1972 before the term EBL was even suggested. The input to STRIPS consists of an initial model state, a goal state, and a set of actions. The set of actions includes the preconditions a state must satisfy before the action may be applied, and a list of deletions and additions made to a state by this action when applied. The sequence of actions is recorded as they are applied and when the goal state is reached the recorded sequence of actions is generalized into a MACROP. MACROP's may later be employed whenever the preconditions for its application have been fulfilled. This system may be thought of as an EBL system because the MACROP results from generalizing a sequence of actions that were generated from one initial state, goal state pair.

One of the more recent EBL systems is GENESIS [28], which reads natural language stories that describe actions of people striving to achieve certain goals. From the story, it builds a specific schemata which is later generalized by first removing all nonpertinent information and then generalizing further by turning constants into variables. The general schemata is associated with the human goal the story was intended to describe, and it is then saved. If any future story fits this general schemata then it is classified as an example of the schemata's associated goal. The domain knowledge represented in GENESIS includes hierarchies of objects, object attributes, states, and actions. These hierarchies constitute the domain knowledge base that the system relies upon to build explanations or schemata.

All three systems, STRIPS, GENESIS and FORGE, may be classified as EBL systems because each develops a generalization after observing a single input example. Both STRIPS and GENESIS differ from FORGE by the intended purpose of the system, the representations employed by each, and

the amount of domain knowledge required by each. The domain knowledge base in both STRIPS and GENESIS is extensive in comparison to the domain knowledge available to FORGE. The actions utilized by STRIPS include known preconditions for the application of these actions. GENESIS includes several 'is-a' hierarchies that are utilized during the construction of schemata. FORGE, on the other hand, operates with a limited domain knowledge base. The domain knowledge required by FORGE consists only of examples of the base relations and the target relation. In this light, FORGE may be viewed as an EBL system operating with a limited domain knowledge base.

Quinlan's FOIL system [33] differs from those discussed above in that instead of trying to generalize from one single example, FOIL strives to explain as many positive examples as possible. FOIL, therefore, may be classified as an empirical learning method. We show that our system FORGE, which shares the same goals, representations and limited domain knowledge base as FOIL, greatly benefits from incorporating EBL techniques along with other aspects of FOIL.

Chapter 2

Definitions and Representation

2.1 Definitions

The output of FORGE is in the form of Horn-clauses, which form a subset of first order logic and constitute the clauses used in Prolog. To be complete, we will now review the basic definitions of first order logic and logic programming as presented by Lloyd [19] and Colcksin and Mellish [7].

First order logic may be represented by the Predicate Calculus. In Predicate Calculus objects are represented by terms. A term may have any of the following forms:

- A constant symbol which represents a single individual or concept.
- A variable symbol which may represent different individuals at different times.
- A compound term which consists of a relation symbol together with an ordered set of terms as its arguments.

The relation symbol of a compound term is referred to as a predicate symbol. The arity of a predicate refers to the number of arguments of a predicate. A predicate, P, with k arguments, $t_1, t_2, \ldots, t_k$, is called a k-ary predicate and is represented as $P(t_1, t_2, \ldots, t_k)$. This representation, $P(t_1, t_2, \ldots, t_k)$, is referred to as an atom.

A formula may consist of a single atom or more than one atom along with any of the connectives and quantifiers shown in Table 2.1.

Table 2.1. Connectives and Quantifiers of the Predicate Calculus.

Connective/ Quantifier	Represents
$\wedge$	conjunction (and)
$\vee$	disjunction (or)
$\rightarrow$	implication (note A $\rightarrow$ B may also be written as B $\leftarrow$ A)
$\leftrightarrow$	equivalence
$\sim$	negation
$\exists$	existential quantifier for variables (there exists),
$\forall$	universal quantifier for variables (for all).

For example, the formula $\forall X\ (p(X, g(X)) \leftarrow q(X) \wedge \neg r(X))$ is interpreted as for every X, if q(X) is true and r(X) is not true, then p(X, g(X)) is true.

A literal is an atom or the negation of an atom. A clause is the disjunction of literals. For example a clause is a formula of the form

$$\forall X_1\ \forall X_2\ ...\ \forall X_k\ (L_1 \vee L_2 \vee \cdots \vee L_m)$$

where each L_i is a literal and $X_1, X_2, \ldots, X_k$ are the set of variables occurring in $L_1 \vee L_2 \vee \cdots \vee L_m$.

If we let A_i denote the positive literals in a clause, and B_i denote the literals that are negated, then the clause

$$\forall\ X_i \cdots \forall X_k\ (A_1 \vee \cdots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n\ )$$

where A_i and B_j, $1 \le i \le m$, $1 \le j \le n$, are atoms and $\{X_i \mid 1 \le i \le k\}$ is the set of terms found in these atoms, may be represented as

$$\forall X_1 \cdots \forall X_k\ (A_1 \vee \cdots A_m \leftarrow B_1 \wedge \cdots \wedge B_n)$$

A Horn clause is a clause which contains one positive literal and may be written as

$$A \leftarrow B_1 \wedge \cdots \wedge B_n$$

The existential quantifiers are implicitly implied in all Horn clauses. In the above Horn clause, A is the head of the clause and $B_1 \wedge \cdots \wedge B_n$ is the body of the clause.

A Horn clause containing no negative atoms may be represented as

$$A \leftarrow$$

and is sometimes referred to as a fact. Using these definitions, we will now describe the representation used in our algorithm.

2.2 Representation

The input to FORGE consists of specific instances from a set of concepts or relations. One of these relations is specified as the target relation and the remaining relations will be known as the base or ground relations. Both the target and base relations will be represented by predicates. For example, if the target concept is a k-ary relation then the predicate used to represent the target concept will be a predicate with k arguments. If a k-ary relation R is represented by a k-ary predicate P, then a specific

instance of R is represented by $P(e_1, e_2, \cdots e_k)$, where the e_i, $i = 1, \cdots k$, are constants. We refer to a specific instance of a relation as a tuple. The set of specific instances of the target relation given as input will be known as the target tuples and the set of specific instances of the base relations are called the base tuples.

The set of all constants appearing in the base tuples is called the universe. We will assume that all constants appearing in the set of target tules are members of the universe. This is known as domain closure and its importance will be demonstrated later.

The output produced by FORGE is a set of rules which are in the form of Horn-clauses. We will use the same terminology for rules as was defined for Horn-clauses in the previous section.

We define a substitution $\Theta = \{X_1/a_1, X_2/a_2, \cdots X_n/a_n\}$, where the X_i are variables and the a_i are constants, to be a mapping such that when applied to a predicate $P(X_1, X_2, \cdots X_k)$ replaces all occurrences of each X_i in P with the corresponding a_i. We indicate the application of substitution Θ on predicate P by $P\Theta$.

We define an instantiation of a rule body $B = b_1 \wedge b_2 \wedge \cdots b_k$ to be the set of all distinct substitutions Θ_i, $i = 1, \cdots n$, such that $B\Theta_i = b_1\Theta_i \wedge b_2\Theta_i \wedge \cdots b_k\Theta_i$ and each $b_j\Theta_i \in$ base tuples $\cup$ target tuples, for $j = 1, \cdots k$ and $i = 1, \cdots n$.

If the set of arguments from head H of rule R is contained in the set of arguments from the body B of R then the cover of R is defined as

$$\text{cover}(R) = \{H\Theta_i: \Theta_i \in \text{instantiations of B}, i = 1, \cdots n\}$$

We note that $H\Theta_i$ is a specific instance of the target relation. In this case, we refer to R as being full.

If the set of arguments from head H of rule R is not contained in the set of arguments from body B of R, then R is not full and $H\Theta_i$ contains at least one variable. In this case we will need some preliminary definitions before specifying the cover of R.

Let n be the number of distinct variables in $H\Theta_i$ and let $v = (X_1, X_2, \cdots X_n)$ be an ordered n-tuple, where X_i is a variable contained in $H\Theta_i$ and $X_i \neq X_j$, $i \neq j$. We further let $\Psi = \{(e_1, e_2, \cdots e_n): e_i \in \text{universe}, i = 1, \cdots n\}$, which is the set of all possible n-tuples whose elements are derived from the universe. Finally, we let Φ be a set of substitutions Θ_j such that $\{v\Theta_j : \Theta_j \in \Phi\} = \Psi$. Now we may define the cover of rule R to be

$$\text{cover}(R) = \{H\Theta_i\Theta_j: \Theta_i \in \text{instantiationsofB, and}\Theta_j \in \Phi\}$$

In other words, if any variables remain after applying the substitution Θ_i to H, then the cover of R is determined by letting each of these variables take on all possible constant values from the universe in turn.

If $t_i \in \text{cover}(R)$ and $t_i \in$ target tuples then t_i is referred to as a $\oplus$ tuple; otherwise t_i is a $\ominus$ tuple. We define a valid rule to be a rule whose cover contains only $\oplus$ tuples.

We let S denote the set of rules derived by FORGE. Before the algorithm starts, $S = \varnothing$. As FORGE derives rules S is updated. Valid rules may be added or deleted from S according to guidelines which are described in Section 3.6. We define cover(S) by the following

If $S = \{R_1, R_2, \cdots R_n\}$, where R_i is a valid rule, then

$$\text{cover}(S) = \bigcup_i \text{cover}(R_i),\ i = 1, \cdots n;\ \text{otherwise cover}(S) = \varnothing.$$

We note that cover(S) contains only $\oplus$ tuples.

2.3 Example Representation

In order to present a clear view of the terminology, we present an example problem to illustrate the representation used in FORGE. Consider the graph in Figure 2.1 below. The concept of a node being linked to another node by one forward arc in the graph may be represented by a relation called linked-to.

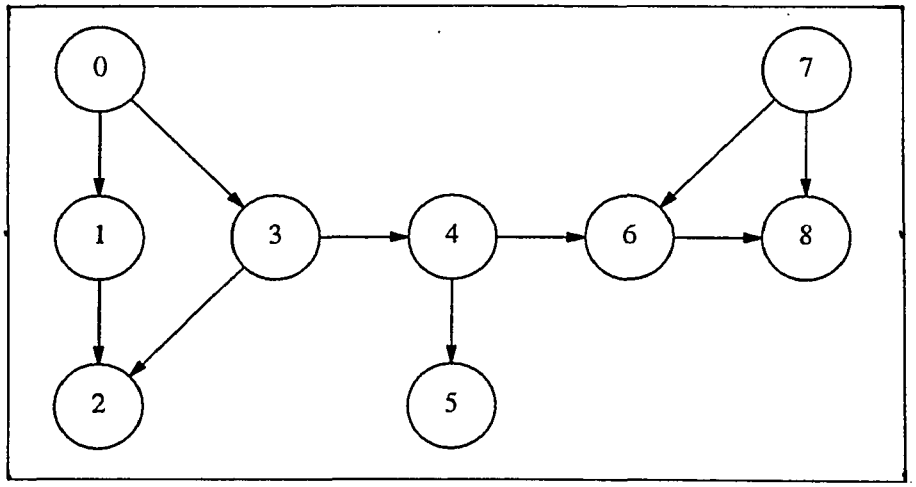

Figure 2.1. Graph Illustrating Linked-to Relation.

If we wished to learn rules for the concept of two nodes being linked by a path of exactly two forward arcs, then we would need target tuple values for this concept. In this case we let linked-by-2 represent the target concept and linked-to constitutes the set of base relations. The base and target tuples for both of these concepts are found in Table 2.2.

Table 2.2. Example Target and Base Relation Tuples.

Base Tuples		Target Tuples	
linked-to(0, 1)	linked-to(0, 3)	linked-by-2(0, 2)	linked-by-2(0, 4)
linked-to(1, 2)	linked-to(3, 2)	linked-by-2(3, 5)	linked-by-2(3, 6)
linked-to(3, 4)	linked-to(4, 5)	linked-by-2(4, 8)	linked-by-2(7, 8)
linked-to(4, 6)	linked-to(6, 8)		
linked-to(7, 6)	linked-to(7, 8)		

The universe in this case is {0, 1, 2, 3, 4, 5, 6, 7, 8}, which is the set of all constant arguments appearing within the set of base tuples. Notice that the set of constant arguments from the target tuples, {0, 2, 3, 4, 5, 6, 7, 8} is a subset of the universe. The target concept linked-by-2 may be defined by rule (1) listed below.

Rule (1): linked-by-2 (X, Y) ← linked-to (X, Z), linked-to (Z, Y).

Notice that rule (1) is a full rule, that is, the variables X and Y in the head of the rule are contained in the body of the rule. In this example, both the target and the base relations are binary relations and there is only one concept in the set of base relations. Our algorithm is not restricted by the arity of

the relations nor the number of concepts included in the base relations, however, the runtime order of the algorithm is closely related to both the arity and number of the input tuples. Once a rule has been constructed it is evaluated by determining its cover. We find every mapping from the variables X, Y, and Z to the universe that will match the literals in the body of rule (1) with the base tuples listed in Table 2.2. After finding a mapping, we apply it to the head of rule (1) to produce a tuple belonging to the cover of rule (1). If this cover tuple belongs to the set of target tuples, then it is a $\oplus$ tuple, otherwise it is a $\ominus$ tuple. We list these mappings, the cover tuple produced by each mapping, and the type of the cover tuple in Table 2.3.

Table 2.3. Mappings, Cover Tuple and Type for Rule (1).

Mapping	Cover Tuple	Tuple Type
{X/0, Z/1, Y/2}	linked-by-2(0, 2)	$\oplus$
{X/0, Z/3, Y/2}	linked-by-2(0, 2)	$\oplus$
{X/0, Z/3, Y/4}	linked-by-2(0, 4)	$\oplus$
{X/3, Z/4, Y/5}	linked-by-2(3, 5)	$\oplus$
{X/3, Z/4, Y/6}	linked-by-2(3, 6)	$\oplus$
{X/4, Z/6, Y/8}	linked-by-2(4, 8)	$\oplus$
{X/7, Z/6, Y/8}	linked-by-2(7, 8)	$\oplus$

Notice that the first and second mappings listed in Table 2.3 produce the the same results. As we shall discuss in detail later, some learning algorithms try to use this information when evaluating a rule. FORGE, however, utilizes the set of target tuples covered, thus such information is not used in determining when a rule is appropriate. The list of tuples covered in Table 2.3 contains only $\oplus$ tuples, so rule (1) is a valid rule. Furthermore, the list of tuples covered in Table 2.3 contains all of the target tuples of the target concept so our set of rules to be output by FORGE would include only rule (1).

Since rule (1) is a full rule, each mapping of Table 2.3 produced only one tuple for cover(rule (1)). If the rule for which the cover is to be evaluated is not a full rule, then each mapping will produce many tuples to be added to the cover of the rule. For example, consider rule (2) listed below. Notice that the variable Y appears in the head of rule (2), but Y does not appear in the body of the rule. In such cases as this, the resulting mappings will not assign a constant value to Y, therefore to evaluate the cover we must assign Y every possible value from the universe.

Rule (2): linked-by-2 (X, Y) $\leftarrow$ linked-to (X, Z), linked-to (W, X).

The mappings and the cover tuples produced by each mapping for rule (2) is listed in Table 2.4. The cover tuples which are instances of the target relation are followed by the indication ⊕.

Table 2.4. Mappings and Cover Tuples for Rule (2).

Mapping	Cover Tuples		
{X/3, Z/2, W/0}	linked-by-2(3, 0)	linked-by-2(3, 1)	linked-by-2(3, 2)
	linked-by-2(3, 3)	linked-by-2(3, 4)	linked-by-2(3, 5)⊕
	linked-by-2(3, 6)⊕	linked-by-2(3, 7)	linked-by-2(3, 8)
{X/3, Z/4, W/0}	linked-by-2(3, 0)	linked-by-2(3, 1)	linked-by-2(3, 2)
	linked-by-2(3, 3)	linked-by-2(3, 4)	linked-by-2(3, 5)⊕
	linked-by-2(3, 6)⊕	linked-by-2(3, 7)	linked-by-2(3, 8)
{X/4, Z/5, W/3}	linked-by-2(4, 0)	linked-by-2(4, 1)	linked-by-2(4, 2)
	linked-by-2(4, 3)	linked-by-2(4, 4)	linked-by-2(4, 5)
	linked-by-2(4, 6)	linked-by-2(4, 7)	linked-by-2(4, 8)⊕
{X/4, Z/6, W/3}	linked-by-2(4, 0)	linked-by-2(4, 1)	linked-by-2(4, 2)
	linked-by-2(4, 3)	linked-by-2(4, 4)	linked-by-2(4, 5)
	linked-by-2(4, 6)	linked-by-2(4, 7)	linked-by-2(4, 8)⊕
{X/6, Z/8, W/6}	linked-by-2(6, 0)	linked-by-2(6, 1)	linked-by-2(6, 2)
	linked-by-2(6, 3)	linked-by-2(6, 4)	linked-by-2(6, 5)
	linked-by-2(6, 6)	linked-by-2(6, 7)	linked-by-2(6, 8)

The first and second mappings listed in Table 2.4 produce the same set of cover tuples, as do mappings three and four. This yields a total of 24 distinct ⊖ tuples in the cover of rule (2) while only a total of 3 distinct ⊕ tuples are in the cover. Obviously rule (2) is not a valid rule and would not be considered as a member of the set S of valid rules by the FORGE algorithm.

Having presented the representation utilized by FORGE and the method for evaluating a rule produced by FORGE, we will next explore the method by which FORGE constructs rules.

Chapter 3

The Explanation Tree

3.1 Explanations and Ground Rules

In this chapter we present our definition of an explanation of an example e of the target concept and we define and describe construction of an explanation tree $T_S(e)$. We use explanation trees, as they are defined here, to produce rules in the form of Horn-clauses.

Given a set of valid rules S, we define an explanation relative to S of $e \notin S$ to be a sequence $\sigma(e) = \langle e_j : 1 \leq j \leq n \rangle$ of distinct tuples which satisfy to following two conditions.

1. Each e_j is a tuple of a base relation or is in cover(S).
2. For each e_j, at least one of its arguments appear either in e or in a previous e_i, $1 \leq i < j$. We say the e_j is linked to either e or e_i as the case may be.

If $\sigma(e)$ and $\sigma'(e)$ are different orderings of the same e_j, then $\sigma(e)$ and $\sigma'(e)$ are considered equivalent. The justification for considering $\sigma(e)$ and $\sigma'(e)$ equivalent may be explained in the light of Horn-clauses. As discussed later in this section, these $\sigma(e)$'s will be used to produce bodies of Horn-clauses. A Horn-clause is satisfied whenever the literals within its body is satisfied. The order in which the literals are satisfied does not effect the results. We will therefore consider two Horn-clauses that differ only by the order of the body literals as equivalent clauses. The following two Horn-clauses, for example, are considered equivalent.

$$Q(X, Y) \leftarrow P(X), R(Y, X).$$

$$Q(X, Y) \leftarrow R(Y, X), P(X).$$

Notice that the body of the first clause is simply a permutation of the literals within the body of the second clause. We know that in actual Prolog implementations the ordering of literals may have profound effect on efficiency, but this is a characteristic of the implementation and not of the underlying theory. We will therefore consider $\sigma(e)$ equivalent to $\sigma'(e)$ if one is simply a permutation of the other.

We define $e \leftarrow \sigma(e)$ to be a ground rule where $\sigma(e)$ may be empty and we define the length of a ground rule to be the number of tuples in the explanation $\sigma(e)$. A ground rule with an empty

explanation is of length 0. Figure 3.1 contains an illustration of a ground rule with e = t(a, b) and $\sigma(e)$ = m(a, d), p(d, e). As the definition at the beginning of this section states, each tuple in $\sigma(e)$ must have at least one argument in common with either e of some previous e_i in $\sigma(e)$. We call these common arguments links and illustrate them in Figure 3.1 with connecting lines between common arguments. We say that m(a, d) is linked to t(a, b) by argument a and p(d, e) is linked to m(a, d) by argument d.

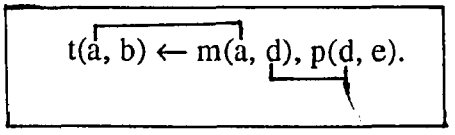

Figure 3.1. Ground Rule with Lines Indicating Links.

Ground rules will be generated from explanation trees, the construction of which is explained in the following section. It will also be shown in the following section that each path in an explanation tree will produce a distinct ground rule. By distinct we mean that no two ground rules produced from the paths of an explanation tree will be equivalent. The root node of the tree contains the tuple value of e and the interior nodes of the tree take on the values of the e_i. For example, the list of tuples in Table 3.1 will produce the explanation tree shown in Figure 3.2.

Table 3.1. Example e and e_i for Tree Construction.

e	e_i	
Target tuple	Base tuples ∪ Cover(S)	
t(a, b)	m(a, d)	∅
	n(f, y)	
	p(d, e)	
	q(d, a)	

We refer to an explanation tree as $T_S(e)$ and say that it produces explanations of e relative to S meaning that the interior nodes e_i are chosen from either the base tuples or the target tuples in cover(S). Notice that the tree in Figure 3.2 does not contain the tuple n(f, y) even though this tuple belongs to the set of base tuples. This is due to the fact that n(f, y) is not linked to e or any e_i within $T_S(e)$.

Each node in the explanation tree $T_S(e)$ has associated with it a ground rule represented by the path from the root of $T_S(e)$ to this node. If we use the tuple values e_i to label the nodes of the tree as in Figure 3.2, we see that even though each node has a unique ground rule associated with it, the labels of each node are not unique. There are at least two convenient ways of labeling the nodes so that each

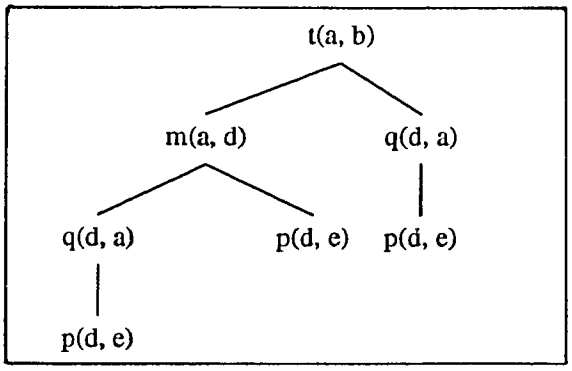

Figure 3.2. Typical Explanation Tree, $T_S(e)$.

node has a unique label. One way is to consecutively number the nodes from left to right and top to bottom, which is the method used in our implementation. A more informative way, however, would be to use the notation $N_{i,j}$ to denote the j^{th} node from the left on level i of the tree. Since each node $N_{i,j}$ has associated with it a unique ground rule which is constructed from the nodes in the path from the root node to node $N_{i,j}$, we also let $R_{i,j}$ denote the ground rule associated with node $N_{i,j}$. The value of node $N_{i,j}$ will be the value of the tuple e_i associated with this node. Table 3.2 contains a listing of all ground rules produced by the explanation tree $T_S(e)$ in Figure 3.2.

Table 3.2. Ground Rules Resulting from $T_S(e)$ in Figure 3.2.

Length	Ground Rules
0	$R_{0,1}$: t(a, b) ←.
1	$R_{1,1}$: t(a, b) ← m(a, d). $R_{1,2}$: t(a, b) ← q(d, a).
2	$R_{2,1}$: t(a, b) ← m(a, d), q(d, a). $R_{2,2}$: t(a, b) ← m(a, d), p(d, e). $R_{2,3}$: t(a, b) ← q(d, a), p(d, e).
3	$R_{3,1}$: t(a, b) ← m(a, d), q(d, a), p(d, e).

We have mentioned that it will be shown that each rule within an explanation tree is unique, but we also need to show that all explanations of a tuple e will be produced by an explanation tree. To do this we introduce the notion of an equivalence class of rules in the following section.

3.2 Equivalence Classes of Ground Rules

We define two ground rules, R_1 and R_2, as equivalent if the body of R_1 is a permutation of the body of R_2. An equivalent class of ground rules is the set of all equivalent ground rules. We define a representative ground rule from each equivalent class by the following REPEX, REPresentative EXplanation, algorithm.

Algorithm REPEX

Input: A ground rule R which contains an explanation of e and the set C of current facts which is ordered in some fashion.

Output: The unique representative ground rule R_0 in the equivalence class of R.

1. Let $A = \{a_1, a_2, \cdots, a_k\}$ be the ordered set of distinct arguments in the tuple e such that if the first occurrence of a_i appears before the first occurrence of a_j in e then a_i appears before a_j in set A.

2. Group the tuples in the body of R into disjoint classes $G_1, G_2, \cdots, G_k$ and G_0 according to the following.

   $G_1 = \{e_j: e_j \in$ body of R and e_j contains $a_1\}$

   $G_2 = \{e_j: e_j \notin G_1, e_j \in$ body of R and e_j contains $a_2\}$

   $\cdots$

   $G_n = \{e_j: e_j \notin \bigcup_{i=1}^{k-1} G_i, e_j \in$ body of R and e_j contains $a_n\}$

   G_0 = the set of remaining tuples in the body of R.

3. Rearrange the tuples in the body of R by moving the tuples in group G_1 to the leftmost positions, followed by the tuples in group G_2, etc. Now reorder the the tuples within each group G_j, $1 \le j \le k$, by placing then according to the ordering of the set of current facts C. Call this new rule R'; the tuples in G_0 form the end of R' and their relative order in R' is the same as that in R. Since all tuples in G_j, $j \ge 1$, contain a_j, it is clear that the body of R' is an explanation of e and R' is equivalent to R. We refer to the initial part of R' which is made of the tuples in $G = \bigcup_{i=1}^{k} G_i$ as the *processed* part and the remaining part of R', made of the tuples in G_0, as the *unprocessed* part.

4. If G_0 is empty or consists of only one tuple, then we let $R_0 = R'$. Otherwise, we reorder the part G_0 as follows. Let e' be the first tuple in the processed part of the body of R' which has an argument that occurs in the unprocessed part of the body of R'. Since the definition of an explanation requires that any tuple within an explanation must be linked to tuple e or some other tuple already in the explanation, we are assured the such an e' exists. Apply steps (1)-(3) using the tuple e' in the role of e to rearrange the tuples in within G_0. The processed part from G_0 is marked as processed and follows the previous processed part G in R'. Consider the unprocessed part in G_0 as the new unprocessed part, i.e., the new G_0. Clearly the new R' is equivalent to R and has at least one less tuple in the unprocessed part.

5. Apply step (4) repeatedly until the unprocessed part becomes empty or consists of only one tuple.

□

If we apply the algorithm REPEX to the explanation "t(a, d) ← p(a, b), q(b, b), p(c, d)" and we assume that the order of the tuples within the current facts C is p(a, b) < p(c, d) < q(b, b), then we get G_1 = {p(a, b)}, G_2 = {p(c, d)}, G_0 = {q(b, b)}, and R' = p(a, b), p(c, d), q(b, b) after step (3). The processed part is p(a, b), p(c, d) and the unprocessed part is q(b, b). Since G_0 contains only one tuple we let $R_0 = R'$ and the algorithm terminates.

The construction of the explanation tree $T_S(e)$ closely parallels the REPEX algorithm. It will become apparent that the rules produced by an explanation tree $T_S(e)$ are the unique representatives from the equivalent classes of rules for e.

3.3 Construction of $T_S(e)$

We define the set C to be the union of all base tuples and all target tuples in cover(S). We call this set C to remind us that it is the set of current facts. Now, let G denote the set of tuples from C that are linked to e and let G_0 be the set difference C – G. One of the important concerns when constructing an explanation tree $T_S(e)$ is to prohibit the production of equivalent paths within the tree. If the set G of tuples from which the interior node values are produced remained constant throughout tree

construction, then the process would be identical to listing all combinations or subsets of the set G. However, as will become evident, the set G is continually changing. To ensure that equivalent paths are not produced we will employ a partial ordering on the set G. G is first constructed in the following manner.

1. Let $A = \{a_1, a_2, \cdots, a_n\}$ be the ordered set of distinct arguments from tuple e. The order of A is such that if the first occurrence of a_i appears before the first occurrence of a_j within the tuple e, then a_i appears before a_j within A.

2. Generate the sets G_i as follows

$$G_1 = \{e_i: e_i \in C \text{ and } e_i \text{ contains argument } a_1\}$$

$$G_2 = \{e_i: e_i \in C, e_i \notin G_1 \text{ and } e_i \text{ contains argument } a_2\}$$

$$\vdots$$

$$G_k = \{e_i: e_i \in C, e_i \notin \cup G_j, j = 1, \cdots, k-1 \text{ and } e_i \text{ contains argument } a_k\}$$

The order of e_i within each set G_j, $j = 1, \cdots, k$, is consistent with the order of the e_i in the set C, i.e., if e_i and $e_k \in G_j$ and e_i appears before e_k in C then e_i will appear before e_k in G_j.

3. Let $G = \bigcup_{j=1}^{k} G_j$, where the union is order preserving such that for all $i < j$ the elements of G_i appear in G before elements of G_j and in the same order as in G_i. □

We denote this ordered process of selecting $e_i \in C$ which are linked to e by the notation

$$\text{order}\{e_i: e_i \in C \text{ and } e_i \text{ is linked } e\}$$

and we refer to it as the ordered selection process. We now define a partial ordering on the set $G = \{e_1, e_2, \cdots, e_n\}$ such that $e_i < e_j$ whenever $i < j$ so we have $e_1 < e_2 < \cdots < e_n$.

We construct the first level of $T_S(e)$ by adding nodes $N_{1,i}$ whose values are $e_i \in G$ as children of $N_{0,1} = e$, for $i = 1, .., n$ in left to right order, as shown in Figure 3.3.

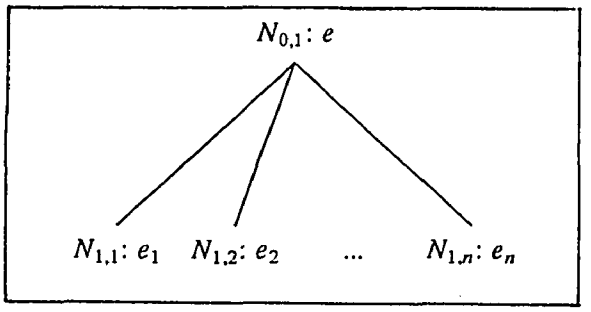

Figure 3.3. Construction of Level One of $T_S(e)$.

Since the set of child tuples may be different for each node, we expand our notations to let $G(N_{i,j})$ denote the set of tuples which is used as child nodes of $N_{i,j}$ and $G_0(N_{i,j})$ to be the corresponding G_0 set. The original G and G_0 in the new notation become $G(N_{0,1})$ and $G_0(N_{0,1})$, respectively. We also let the parent node of node $N_{i,j}$ be denoted $N^p{}_{i,j}$. $N_{0,1}$ has no parent since it is the root node.

We now give the definition for building the sets $G(N_{i,j})$ and $G_0(N_{i,j})$ in general and then illustrate the use of these sets by expanding the tree in Figure 3.3 from one node by one level.

$$G(N_{i,j}) = \{e_k : e_k \in G(N^p{}_{i,j}) \text{ and } e_k > \text{value of } N_{i,j}\} \cup$$
$$\text{order}\{e_k : e_k \in G_0(N^p{}_{i,j}) \text{ and } e_k \text{ is linked to the value of } N_{i,j}\},$$

$$G_0(N_{i,j}) = \{e_k : e_k \in G_0(N^p{}_{i,j}) \text{ and } e_k \notin G(N_{i,j})\}$$

The above formalizations may be applied to the root node $N_{0,1}$ by replacing $G(N^p{}_{0,1})$ with $\varnothing$ and $G_0(N^p{}_{i,j})$ with the set C of current facts as described above. We note that the set union operator used in the definition of $G(N_{i,j})$ is order preserving and the definition of $G_0(N_{i,j})$ is equivalent to the set difference of $G_0(N^p{}_{i,j})$ and $G(N_{i,j})$.

Before illustrating these sets with an example, we make some observations about the sets $G(N_{i,j})$ and $G_0(N_{i,j})$. First, these two sets are disjoint which implies that the sets $G(N^p{}_{i,j})$ and $G_0(N^p{}_{i,j})$ are disjoint since they resulted from the same formalization. The set $G(N_{i,j})$ is composed of two sets, the first of which is derived from $G(N^p{}_{i,j})$ and the second is derived by the ordered selection process from

set $G_0(N^p_{i,j})$. This implies that the two sets which make up $G(N_{i,j})$ are disjoint. These facts allow us to extend the partial order which was defined on $G(N^p_{i,j})$ to the set $G(N_{i,j})$ without any conflicts.

Now to illustrate the use of these sets within the construction process of $T_S(e)$, we extend the tree in Figure 3.3 by one level from node $N_{1,2}$. We have

$$G(N_{1,2}) = \{e_3, e_4, \cdots e_n\} \cup \{e'_1, e'_2, \cdots e'_m\} \text{ and}$$

$$G_0(N_{1,2}) = G_0(N_{0,1}) - G(N_{1,2}).$$

where each e'_i, $i = 1, \cdots, m$, is linked to e_2, the value of $N_{1,2}$ and we have the partial ordering $e_3 < e_4 < \cdots e_n < e'_1 < e'_2 < \cdots e'_m$. The resulting tree is shown in Figure 3.4.

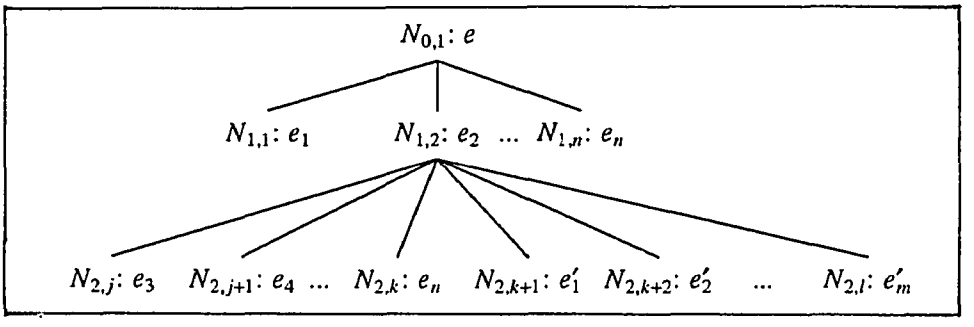

Figure 3.4. The Tree of Fig. 3.3 Extended One Level from Node $N_{1,2}$.

Note that the illustration in Figure 3.4 does not show the entire second level of $T_S(e)$ but just the section of level 2 produced by adding the child nodes of $N_{1,2}$.

In the following discussion we only consider paths within $T_S(e)$ which initiate at the root $N_{0,1}$. If two paths P_1 and P_2 are of the same length n, then the ending nodes of P_1 and P_2 lie on level n of $T_S(e)$ and we say path P_1 lies to the left of path P_2 if the ending node of P_1 lies to the left of the ending node of P_2 in level n of $T_S(e)$. Furthermore any extension of path P_1 lies to the left of path P_2 or any extension of P_2. All paths within $T_S(e)$ will have at least one common node $N_{0,1}$ at the beginning of each path. If P_1 and P_2 are two paths with the first n nodes of each path being equal, then we say the P_1 and P_2 share a common initial subpath P'_n of length n.

Now that we have described the construction of $T_S(e)$, we need to show that no two distinct paths in the tree will produce equivalent ground rules. This will insure that the tree does not produce more than one rule from each eqivalence class of rules, and that a minimum number of rules will be produced. We accomplish this in the following lemma.

Lemma 1. If P_1 and P_2 denote two paths in $T_S(e)$ from the root where P_1 lies to the left of P_2, then there exists some e_k belonging to P_1 which does not belong to P_2.

Proof. Since P_1 and P_2 are paths within $T_S(e)$ we know that they have a common initial subpath P'_n where n ≥ 1. Let $N_{i,j}$ denote the last node of P'_n and consider the child node values of $N_{i,j}$. Let e_{p1} be the child node value of $N_{i,j}$ which belongs to P_1, and let e_{p2} be the child node value of $N_{i,j}$ which belongs to P_2. Since P_1 lies to the left of P_2, e_{p1} must lie to the left of e_{p2}, which implies that $e_{p1} \leq e_{p2}$. Now by the definition of $G(N_{i,j})$ and $G_0(N_{i,j})$ we know that e_{p1} and $e_{p2} \notin G_0(N_{i,j})$ which implies that $e_{p1} \notin G_0(e_{p2})$ and since $e_{p1} \leq e_{p2}$, we know that $e_{p1} \notin G(e_{p2})$ therefore e_{p1} which belongs to P_1 does not belong to P_2 or any extension of P_2. □

Lemma 1 above assures us that no two distinct paths in an explanation tree produce equivalent rules. We now turn our attention to the unique representative ground rules produced by the algorithm REPEX. The following lemma and proof shows that any rule produced by a path within an explanation tree will be in the representative form of the REPEX algorithm.

Lemma 2. Any path P_i will produce a representative ground rule R_0.

Proof. Let R be the rule produced by path P_i in $T_s(e)$. By the construction of $T_s(e)$ it is clear that the tuples in the body of R will consist first of the tuples linked to e in the order of the arguments of e and the order of the current facts C. This order is consistent with the order present in the resulting processed part of steps (1)-(3) of algorithm REPEX. The remaining tuples in the body of R will be ordered according to their links to the processed part of R and the order of C. This is also consistent with the ordering of the unprocessed part of a rule by algorithm REPEX. □

We now turn to the implementation of the tree building algorithm. The following section describes the details of the implementation of the tree building algorithm.

3.4 Tree Building Algorithm

The FORGE system is implemented in Franz Lisp. The pseudocode of the algorithm for generating $T_S(e)$ is presented and discussed in this section. Only the major functions utilized by the tree building algorithm are presented here. A complete pseudocode listing may be found in the Appendix.

The FORGE algorithm requires that the tree be constructed one level at a time. After constructing one level of the tree, all previous levels of the tree are no longer required, therefore after a level is constructed the previous level in the tree is discarded. The information stored at each node is shown in Table 3.3.

Table 3.3. Information Stored at a Node in $T_S(e)$.

Label	Description
VALUE	The tuple value for this node
G	The set G for this node
G0	The set G_0 for this node
PARENT	A pointer to the parent node of this node
PCOVER	Previous cover; the cover from the parent of this node
COVER	Cover of rule associated with this node
CURRMAP	The resulting maps from the instantiations of this node

PCOVER and COVER are both lists which consist of 2 integers. The first integer indicates the number of $\oplus$ tuples covered and the second is the number of $\ominus$ tuples covered. This information is used when pruning the tree and is only saved when the pruning option has been indicated. Tree pruning will be described further in Chapter 5.

The key to the tree building algorithm is to assign the proper sets G and G_0 to a node when it is created. This is accomplished by the function BUILD-FAMILY whose pseudocode is listed in Figure 3.5. Another important issue is preserving the order of the sets G and G_0. All functions utilized within the tree building algorithm are order preserving.

Line 9 of function BUILD-FAMILY assigns the proper set G to the property list of newnode. Analyzing line 9, we see that the set G for the newnode is made of two parts, the first of which is

tail (G)

Since the do-while loop is iterating through the parent's set G, this statement will produce all elements of parent's set G that are > childvalue. The second part

GENERATE-G (REMOVE-DUPLICATES (tail (childvalue)), G0)

first removes duplicates from the arguments of childvalue then uses these distinct arguments to choose from the parent's set G_0 all those e_i which contain at least one of these distinct arguments. This selection is done according to the ordered selection process which was described in the previous section. Line 10 of BUILD-FAMILY adds the proper set G_0 to the property list of newnode. From this line we see that the set G_0 assigned to newnode is

G0 minus GET-G (newnode)

which is the set difference of the parent's set G_0 and the child's set G. BUILD-FAMILY also adds the value of the node (line 8), the ground rule associated with this newnode (line 11), and the previous cover (line 12) to the property list of the newnode. BUILD-FAMILY returns a list of the newnodes that have been created. The pseudocode for BUILD-FAMILY is given in Figure 3.5.

```
Function:   BUILD-FAMILY (parent)

Input:      parent is the tree node for which children are to be created.

Output:     The list of children created for parent. If parent has no children, then nil is returned.

Note:       This function is called by EXTEND-TREE to produce all the children nodes for one parent. It produces one node for each element of the set G of parent. It also adds the value of the node, the ground rule associated with the node, and the correct G and G0 sets to the property list of each child node.

Method:
 1)  G ← GET-G (parent)
 2)  If G = nil then return nil
 3)  G0 ← GET-G0 (parent)
 4)  childlist ← nil
 5)  Do while G ≠ nil
 6)      childvalue ← head (G)
 7)      childlist ← childlist ⊕ (newnode ← GENERATE-LABEL)
 8)      SET-VALUE (newnode, childvalue)
 9)      SET-G (newnode, append (tail (G),
             GENERATE-G (REMOVE-DUPLICATES (tail (childvalue)), G0)))
10)      SET-G0 (newnode, G0 minus GET-G (newnode))
11)      SET-PARENT (newnode, parent)
12)      If *prune-by-count* ≠ nil then
13)          SET-PCOVER (newnode, GET-COVER (parent))
14)      G ← tail (G)
15)  return childlist
```

Figure 3.5. Pseudocode for BUILD-FAMILY.

Figure 3.6 displays the pseudocode for the EXTEND-TREE function. The function EXTEND-TREE simply sends parents to BUILD-FAMILY one at a time, and collects each family of children returned from BUILD-FAMILY. The families are collected in one list of child nodes and that list is returned by this function. After a family of children for a parent node has been created, EXTEND-TREE reduces the property list of the parent in an effort to save space. The property values that will no longer be needed for this parent node are removed from the property list. The sets G and G_0 are removed from the parent node's property list along with the cover and pcover values.

Function: EXTEND-TREE (parent-list)

Input: parent-list is a list of tree nodes for which children are to be created.

Output: A list of children created by the function which is the next level of the explanation tree. If no children are created, then nil is returned.

Note: This function is used to extend the tree by one level. It is called by FIND-VALID-RULES. It calls BUILD-FAMILY to actually create the new nodes. After a family has been created the parent node is released.

Method:

```
1)  If parent-list = nil then return nil
2)  parents ← parent-list
3)  childlist ← nil
4)  Do while parents ≠ nil
5)      childlist ← append (childlist, BUILD-FAMILY (head (parents)))
6)      REDUCE-PROPLST (head (parents))
7)      parents ← tail (parents)
8)  return childlist
```

Figure 3.6. Pseudocode for function EXTEND-TREE.

3.5 Generalizing Ground Rules

It has been shown in Section 3.1 that ground rules are produced from paths within the explanation tree $T_S(e)$. Ground rules are generalized to produce general rules by utilizing a substitution Θ which will substitute a distinct variable for each distinct constant argument within the ground rule. Table 3.4 contains an example of a ground rule and its generalization with the substitution Θ = { a/X, b/Y, d/Z, e/V }.

Table 3.4. A Ground Rule and its Generalization.

Ground Rule	t (a, b) ← m (a, d), p (d, e).
Generalized Rule	t (X, Y) ← m (X, Z), p (Z, V).

It is well known [12, 25] that this straight forward substitution of variables for constants may sometimes lead to incorrect rules. This happens when certain constants are used in a special sense within the input tuples. To avoid this type of complication, almost all existing learning systems must somehow have these special constants identified to them. We will assume that all special constants have been replaced by

predicates. This transformation is similar to a process called flattening in Rouveirol and Puget [36] and will increase the size of the universe of the problem and also the number of input base relations.

For example, assume we are given the base relations color-preference, female, and male as listed in Table 3.5, and the target concept to be learned was females who preferred red (fpfer-red) whose tuples are also listed in Table 3.5.

Table 3.5. Example Base and Target Tuples Containing Special Constants.

Base Tuples	color-preference(patti, yellow) color-preference(tom, red) color-preference(julie, red)	color-preference(james, blue) color-preference(bonnie, red)
	female(patti) female(julie)	female(bonnie)
	male(tom)	male(james)
Target Tuples	fpfer-red(bonnie)	fpfer-red(julie)

The universe for the example concepts given in Table 3.5 is { bonnie, james, julie, patti, tom, blue, red, yellow }. Using the tuples given above for the base relations, the best rule that can be derived for the fpfer-red concept is

fpfer-red(X) <- color-preference (X, Y), female (X).

We see that this rule does exclude all males from the fpfer-red concept, but it does not exclude females who do not prefer red, e.g., patti, from the target concept. We also notice that the variable Y in the above rule does not have any restrictions placed upon it by any other literal in the rule, therefore, the color constant may take on any value of color that exists in the universe of this problem. In addition to this fact, the concept to be learned, females who prefer red, has the constant red specified, but it does not appear as part of the target tuples. These facts lead us to realize that the color constants are special constants and should be specified as such before beginning the rule learning process. The transformation on the given base tuples above yields the input tuples found in Table 3.6.

Table 3.6. Base Tuples of Table 3.5 after Transformation for Special Constants.

Transformed Base Relations	color-preference(patti, y) color-preference(tom, r) color-preference(julie, r)	color-preference(james, b) color-preference(bonnie, r)
	female(patti) female(julie)	female(bonnie)
	male(tom)	male(james)
Additional Base Relations	red_p(r)	
	$yellow_p$(y)	
	$blue_p$(b)	

The target concept stays unchanged. We now see that a rule may be developed from the transformed input tuples that will cover only the specified target tuples. This new rule is

fpfer-red(X) <- color-preference (X, Y), female (X), red_p (Y).

The price paid for this transformation includes the universe being expanded by the constants r, y, and b, and the addition of three input base relations red_p, $yellow_p$, $blue_p$. This expense increases the search space of the problem, but it also allows for a valid rule to be found.

As depicted above, we adopt the Prolog convention of letting constants be denoted by lower case letters while variables are denoted by upper case letters. Notice that the generalized rules above are in the form of Horn-clauses. The terminology utilized for Horn-clauses is also be used for generalized rules.

We define a full rule to be a generalized rule R in which each variable within the head of R appears in at least one literal within the body of R. Notice that in the example in Table 3.4 above, the generalized rule is not full; the variable Y appears within the head t (X, Y), but Y does not appear as a variable of any literal within the body of this rule.

We now present and prove some useful lemmas concerning explanation trees and generalized rules.

Lemma 3. If e is the target tuple selected to be root of the explanation tree $T_S(e)$ then each generalized rule resulting from $T_S(e)$ must cover e.

Proof. If R is a full rule resulting from $T_S(e)$ then every argument in the head of the rule is linked to an argument in the body of the rule which implies that there is an instantiation of tuples, i.e., the exact ones used to construct the rule, which will cover e. If R

is not a full rule then there exists at least one argument in the head of the rule which is not linked to a tuple in the body of the rule. Arguments which are linked will be instantiated to the correct values by using the exact tuples used to construct the rule, arguments which are not linked will take on all values present in the universe. We are guaranteed by the domain closure assumption, that the universe contains all arguments appearing within the target tuples, therefore R will cover e. □

Lemma 4: If two generalized rules R_1 and R_2 are equivalent then they resulted from the same explanation tree $T_S(e)$.

Proof: If $R_1 \equiv R_2$ then cover(R_1) = cover(R_2). Now assume that R_1 and R_2 were not produced by the same $T_S(e)$. Without loss of generality we may assume that $T_S(e)$ for R_1 was constructed first. The tuples in cover(R_1) are removed from the uncovered target tuples and the e' to build TS(e') for R_2 will be chosen from the remaining uncovered target tuples so $e' \notin$ cover(R_1) but from Lemma 3 we know that $e' \in$ cover(R_2) which contradicts cover(R_1) = cover(R_2), therefore R_1 and R_2 must be produced by the same $T_S(e)$. □

3.6 Choosing the 'Best' Rules

The FORGE algorithm constructs $T_S(e)$ one level at a time. After a new level has been constructed, the set of ground rules resulting from this level of the tree is generalized. As a rule is generalized, it is tested for equivalence with all previously generalized rules and if it is equivalent to some existing rule, then it is discarded. The end results of this process is a set of distinct generalized rules which are next evaluated as to their validity. This evaluation is accomplished by testing consistency of the rule with the given base and target tuples. According to Dietterich's [9] definition of bias, this choice of rules is based on a non-declarative bias.

After eliminating all non-valid rules, there may be no rules remaining, one rule remaining or many rules remaining. If there is more than one rule left, then we must choose the 'best', which may consist of a single rule or a set of rules.

The first step in eliminating valid rules is to remove all rules whose coverage is a subset of any other rule, therefore insuring that the remaining rules have maximal coverage. Rules with identical coverage are chosen on the basis of simplicity. This heuristic is based on the notion that the set of resulting Horn-clauses should be as simple as possible so that they may be easily understood by humans. The preference for simplier rules is a heuristic bias which may be thought of as a declarative bias which has its origins in the form of the representation used for the operationality criterion.

Simplicity of rules may be determined in a variety of ways. First, let's analyze the situation further. We know that the rules being compared result from the same level of the tree, therefore each rule must have the same number of literals and thus simplicity cannot be determined by the number of literals within the rule. Returning to our Horn-clause representation, we propose that non-recursive rules are more easily understood by humans, therefore we prefer non-recursive rules over recursive rules. Carrying this hypothesis further, when comparing two recursive rules, we cast favor to the rule with the least number of recursive calls, ie, the rule with the fewer number of target tuples in the body of the rule. This will resolve all choices except one, the case when choosing between two rules having the same level of recursion, either two non-recursive rules (level of recursion is none), or two recursive rules with the same number of target tuples within the bodies of the rules. To form a heuristic for determining a simplier rule in this situation, we focus on the nature of the tuples in the input.

Each tuple represents a relation between the arguments of the tuple. A tuple with only one argument, for example red(apple), may be thought of as representing a property assigned to the argument. A tuple with two arguments, for example father(tom, bob), represents a binary relation between the two arguments, and in general, a tuple with n arguments represents a n-ary relation between the n arguments of the tuple. We hypothesize that a tuple with fewer arguments is a simplier concept for a human to understand, i.e., a property is a simplier concept than a binary relation. Following this train of thought when faced with choosing between two rules with the same level of recursion, we will choose the rule with the fewer number of arguments within the rule. This leaves a choice between rules with the same number of arguments, the same level of recursion, and the same coverage. At this point one may return to the specific ground rules which gave rise to these rules and calculate the close of each rule. The rule

with the smallest close would be chosen. We have not encountered a situation where calculating the close of the rules would be an advantage, therefore we simply choose the rule which was constructed first.

The next chapter contains the complete FORGE algorithm and and example of an application of the algorithm. It will show how the tree building process is incorporated within the FORGE Algorithm.

Chapter 4

The FORGE Algorithm

4.1 The Algorithm

We now describe our new algorithm FORGE, which learns rules by generalization of ground explanations. The FORGE algorithm utilizes our unique explanation tree, which was examined in detail in Chapter 3, to construct ground explanations which are then generalized. We let S denote the current set of valid rules that have previously been built. $R'(e)$ will denote the set of ground rules produced by $T_S(e)$ and $R(e)$ will be the set of distinct generalized rules resulting from generalizing $R'(e)$. Any new rule added to S cannot be subsumed by any existing rule in S since it covers tuple e which was not covered by any rule in S. A new rule added to S, however, may subsume existing rules in S. When adding a new rule R to S, we remove any rules in S that R subsumes before adding R to the set S. The reduction of set S by rule R is performed in step 8 of the FORGE algorithm and it is denoted by *red*(S, R).

Algorithm FORGE

Input: A finite set of base relations and a target relation, each of which is specified by a listing of its tuples. The target relation will be called "target" within the algorithm.

Output: A complete set of valid rules which together cover all tuples of the target relation and nothing else.

1. [Initialize] Let the current rule set S = ∅.
2. [Choose e] Choose a tuple $e \notin$ cover(S). (We choose e having the simplest structure if possible; see Section 4.3). If there is no such e, then stop; otherwise initialize L = 1, which denotes the length of the explanations of e to be considered next.
3. [Expand $T_S(e)$] Expand the explanation tree $T_S(e)$ in a breadth-first manner as described in Section 3.3 to level L.
4. [Build ground rule set $R'(e)$] Build ground rules $e \leftarrow \sigma(e)$, one rule for each path of length L in $T_S(e)$. If $R'(e)$ is empty, then add the fact "target(e) ←." to S and go to step 2.

5. [Generalize the ground rules $R'(e)$] Generalize each rule in $R'(e)$ by replacing the constants by variables (see Section 3.5). Eliminate from $R'(e)$ all rules that are equivalent, i.e., differ only in regard to the names of variables; call the new rule-set $R(e)$.

6. [Remove non-valid rules from $R(e)$] Obtain the coverage for each rule in $R(e)$ and remove from $R(e)$ all non-valid rules, i.e., rules which cover $\ominus$ tuples. If the resulting $R(e)$ is empty, update L = L + 1 and go to step 3.

7. [Pick the 'best' rules from $R(e)$] Let $B(e) = R(e)$.

   a. [maximal coverage] If R_i and R_j belong to $R(e)$, and cover(R_i) $\subset$ cover(R_j) then remove R_i from $B(e)$. Repeat for all $i \neq j$. The remaining rules in $B(e)$ have maximal coverage.

   b. [simplest rules] (see Section 3.6) If R_i and $R_j \in B(e)$ and cover(R_i) = cover(R_j) then remove the rule with

      i) more recursive calls

      ii) more arguments

8. For all remaining rules R_i in $B(e)$ replace S by red(S, R_i) and add R_i to S.

9. Go to step 2.

The algorithm above is of exponential order. Procedures to reduce the number of rules which are built within $T_S(e)$ by pruning methods are discussed in Chapter 5. Another consideration in reducing the number of rules to be evaluated relies upon domain knowledge about the type of rules which are to be built. If it is known that every argument within the head of a rule is to be linked to at least one literal in the body of the rule, then only full rules need to be evaluated. Evaluating only full rules would greatly reduce the amount of computation required. This heuristic however would eliminate the generation of rules such as the following,

$$\text{sum(Y, X, Y)} \leftarrow \text{zero(X)}.$$

which tells us that zero added to any number Y yields the number Y. If we know prior to constructing the rule set that it will not contain any such 'identity' type rules, then this heuristic would be greatly beneficial.

Another option for step 7 could possibly be to choose the first valid rule produced by the algorithm. This would lead to a non-optimal set of rules in S, but could be remedied by using reduction methods on the set S as it is constructed. One such reduction or generalization method is known as absorption. This generalization method is discussed in detail in the next section.

4.2 Absorption

Absorption is a generalization process described by Muggleton and Buntine [26] which in our situation may be applied to the set of valid rules S. Before presenting the definition of absorption, we introduce some preliminary definitions.

A substitution $\Theta = \{v_1/t_1, \ldots v_n/t_n\}$ when applied to a literal, uniquely maps all v_i to t_i by replacing all occurrences of each v_i with the corresponding t_i. For example, if the literal L = father (X, Y) and Θ = { X/tom, Y/bob} then $L\Theta$ = father (tom, bob).

Given two literals L_1 and L_2 we say L_1 subsumes L_2 if there exists a substitution Θ such that $L_1\Theta = L_2$. For example the literal L_1 = plus (A, B) subsumes the literal L_2 = plus (3, 4) with Θ = {A/3, B/4}.

Let C_1 and C_2 be two clauses and let L_1 be the set of literals from C_1 and L_2 be the set of literals from C_2. If there exists a substitution Θ such that applying Θ to each literal in L_1 yields a subset of L_2 then we say that C_1 Θ-subsumes C_2. The following notation, even though not technically correct, gives us a convenient way of expressing Θ subsumption. If $C_1\Theta \subseteq C_2$ then C_1 Θ-subsumes C_2.

Now we will proceed with our definition of absorption as follows.

Given two clauses:

$$C_1 = T_1 \leftarrow B$$

$$C_2 = T_2 \leftarrow A$$

where B Θ-subsumes A, ie, $\exists$ Θ such that $B\Theta \subseteq A$ and $A' = A - B\Theta$ then the resulting absorption generalization is

$$C_2' = T_2 \leftarrow A'T_1$$

Since FORGE is concerned with a single target concept, the T_i's in the above definition would represent the same target concept in our application. In other words, given two rules for the target concept T:

$$R_1 = T \leftarrow B$$

$$R_2 = T \leftarrow A$$

and given B Θ-subsumes A and $A' = A - B\Theta$, then preforming absorption would yield the rule $R_2' = T \leftarrow A'T$ and the two previous rules would be replaced by

$$R_1 = T \leftarrow B$$

$$R_2' = T \leftarrow A'T$$

The absorption process may be applied to the set of valid rules S whenever S contains at least two rules which fulfill the absorption criterion. As one may note from the above, applying absorption when possible to the set S produces recursive rules which would be more general than the rules of the original set S. Lemma 5 below illustrates that if the basis rules for recursion are produced first by FORGE, then there is no need for absorption.

Lemma 5. Given $R_1 = T \leftarrow B$ is a valid rule belonging to S and $R_2 = T \leftarrow A$ is a valid rule, and B Θ-subsumes A with $A' = A - B\Theta$, then the ground rule which gives rise to the rule $R_2' = T \leftarrow A'T$ will be produced prior to R_2 or at the same level as R_2 within the explanation tree $T_S(e)$.

Proof. Let the length of $A = |A| = n \geq 1$ and $|B| = m \geq 1$. Since B Θ-subsumes A, we know that $|B| \leq |A| \Rightarrow m \leq n$.

Case 1: If $m = n$ then $|A| = |B| \Rightarrow A - B\Theta = \varnothing = A' \Rightarrow A = B\Theta$ so $R_2 = T \leftarrow A \equiv T \leftarrow B\Theta = R_1\Theta$ which implies that R_1 is equivalent ot R_2, the only difference being a variable substitution. Now if $R_1 \equiv R_2$ then we know from Lemma 4 that they were produced by the same tree and furthermore at the same level of the tree, since they have the same length.

Case 2: If $m < n$, then $|A'T| = |A| - |B| + 1 = n - m + 1 \leq n = |A| \Rightarrow |A'T| \leq |A|$. Now if $|A'T| = |A|$ then the length of R_2' = length of R_2 and so R_2' and R_2 will be produced at the same level of the tree and the rule with the best coverage will be chosen. Now if $|A'T| < |A|$ then rule R_2' is shorter than rule R_2 and therefore will be constructed and evaluated first. □

We traditionally call the rule R_1 of Lemma 5 the basis for the recursion in the rule R_2. Lemma 5 tells us that when learning a recursive relation, if we produce the basis rules of the recursion first, then we are assured to derive the recursive rules before any longer rules are derived. This fact assures us that absorption will not be necessary if the basis rules are derived first, which is the rational behind choosing the simplest object in Step 2 [Choose e] of the FORGE algorithm. If the given base relations or target relation contains some type of structure, this domain knowledge may be exploited in an effort to produce rules for the simplest e tuple before any other e. The reasoning behind this being that the simplest tuples should produce the simplest rules or basis rules first. A method of ordering the target tuples according to structure of their arguments within the universe Ω is discussed in detail in the next section.

4.3 Choosing the Linear Ordering on Ω

We describe a simple method for defining a linear ordering on the universe Ω using the information in the base relation tuples. The motivation behind developing a partial ordering of Ω stems from the notion that tuples which contain "simpler" elements will be derived from simpler rules which may be easier to build. Another motivating factor results when building recursive rules which necessarily require a basis rule. The heuristic used here is that basis rules will result from explanations of the simpler tuples in the target relation. We will start by defining the notion of close sets for the arguments in Ω. We illustrate the basic idea by using a simple example. Consider the base relations null(X) and components(X, Y, Z) in the input shown in Table 4.1.

Table 4.1. The Tuples in the Base Relations Null(X) and Components(X, Y, Z).

null(X)	components(X, Y, Z)
⟨()⟩	⟨(b(a)d), b, ((a)d)⟩
	⟨((a)d), (a), (d)⟩
	⟨(d), d, ()⟩
	⟨(a), a, ()⟩
	⟨(f.s) f, s⟩

Expressions like "(b(a)d)" are regarded as indivisible constants, therefore the universe resulting from the base relations given in Table 4.1 is

$$\Omega = \{a, b, d, f, s, (), (a), (d), ((a)d), (b(a)d), (f.s)\}.$$

Assume that we have been told (or we notice it ourselves) that the last two arguments in each tuple of the components relation are uniquely determined by its first argument. In this case we may define the close of an argument to be a subset of arguments from Ω defined as

$$close(e) = \{ Y, Z : components (X, Y, Z) \in \text{ base tuples } \} \cup close(Y) \cup close(Z).$$

We advocate that arguments with smaller close sets will be the simpler arguments. In order for the close of an argument to indicate this notion, the base relations and the target relations must have sufficiently many interrelated tuples to implicitly indicate the intrinsic structure of these relations as well as the relationship between the target relation and the base relations. Without this information it would indeed be impossible to learn the desired rules. In the present example, this means that if the universe Ω contains the object "((a)d)", then Ω must contain also the objects {(a), (d), (), a, d} and hence many other tuples of the null-relation, components-relation, and the list-relation. Table 4.2 contains a list of the close set for each element of Ω. Note the elements which would intuitively be considered simple are the elements with the smaller close sets while the more complicated elements have larger close sets.

Table 4.2. Close(x) for all $x \in \Omega$.

close(a)	$= \varnothing$	close(b)	$= \varnothing$
close(d)	$= \varnothing$	close(f)	$= \varnothing$
close(s)	$= \varnothing$	close(())	$= \varnothing$
close((a))	$= \{a, ()\}$	close((d))	$= \{d, ()\}$
close(((a)d))	$= \{(a), (d), a, d, ()\}$	close((f.s))	$= \{f, s\}$
close((b(a)d))	$= \{((a)d), (a), (d), a, b, d, ()\}$		

We utilize the close relation to determine a partial ordering on the elements of Ω by letting "x < y" if close(x) $\subset$ close(y). Figure 4.1 displays the partial ordering derived for the current Ω. If close(x) $\cap$ close(y) = $\varnothing$ or if (close(x) $\cup$ close(y)) - (close(x) $\cap$ close(y)) $\neq \varnothing$ then the number of elements in each set may be used to determine order. Finally, we extend the partial order to a linear ordering on Ω in an arbitrary way.

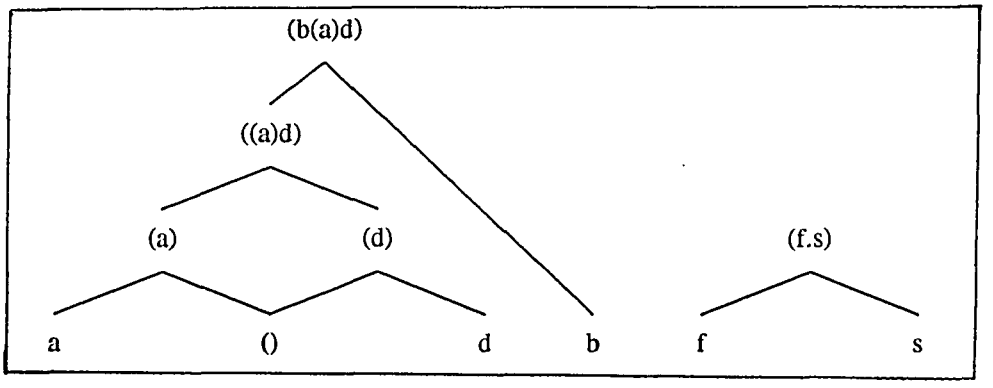

Figure 4.1. Resulting Partial Order for the Input shown in Table 4.1.

We extend our definition of the close for an argument to encompass tuples and explanations in the following manner.

$$close(e) = \bigcup_i close(x_i) \text{ for all arguments } x_i \text{ contained in } e$$

$$close(\sigma(e)) = \bigcup_i close(e_i) \text{ for all } e_i \text{ occurring in } \sigma(e)$$

It is clear that, in general, a smaller $\oplus$ tuple e based on the above ordering on Ω will have fewer explanations, particularly if we restrict the tuples in $\sigma(e)$ to those which have most (or all) of their arguments in the union of the sets close(x), where x is in tuple e. Another advantage of restricting tuples used in $\sigma(e)$ occurs when learning recursive rules. Restricting the tuples used in $\sigma(e)$ to be simpler than e, the tuple that is to be explained, helps to prevent building erroneous recursive rules.

The above method is applicable whenever the objects in the universe Ω have some kind of structure which is relevant for the target concept because the base relations must somehow make that structure implicitly "visible". The rules that involve recursion based on the structure of an object can be discovered by considering the $\oplus$ tuples which are made of the simplest objects. When the objects in Ω contain no structure we cannot use the above method.

Another simple method for defining a linear ordering on Ω is to count the number of occurrences of each constant in Ω in the various base relation tuples and define a linear ordering based on those counts. This method is particularly useful when the objects in Ω contain no structure. For the current example base relations of Table 4.1, we list the counts of occurrences of each constant in Table 4.3 below.

Table 4.3. Counts of Occurrences of Constants within given Base Relations.

Constant	Number of Occurrences
f	1
s	1
(f.s)	1
a	1
b	1
d	1
(b(a)d)	1
(a)	2
(d)	2
((a)d)	2
()	3

We see from Table 4.3 that the constant () would be considered the least simplest constant in this representation and (b(a)d) would be considered one of the more simpler constants. This example shows that simple counts may not produce desired results when dealing with base relations that contain structure of some sort.

Chapter 5

Efficiency Considerations

5.1 Pruning to Improve Efficiency

The FORGE algorithm is of exponential order which in this case means that one representative rule from each equivalent class of rules will be built during the construction of $T_S(e)$. In an effort to reduce the number of rules evaluated, methods of pruning $T_S(e)$ were explored.

Before discussing the pruning method explored, we need to review the effects of adding a literal to a rule. Let R be a rule of length n-1, then adding a literal L_n to R will produce a rule R′ of length n which is an extension of rule R. Now R′ is a more specific rule than R, since it has an added restriction in the literal L_n. The restrictions provided by the additional literal may either (1) have no effect upon the cover of the rule or (2) it may reduce the cover of the rule. If it reduces the cover of the rule it may either reduce the number of ⊕ tuples covered, or reduce the number of ⊖ tuples covered, or it may reduce both the number of ⊕ and ⊖ tuples. There is no possible way for the addition of a literal to increase either the number of ⊕ or ⊖ tuples covered.

Keeping in mind that the cover of a rule under expansion is non-increasing, we may use the cover counts of a rule and its expansion to decide whether the expansion of the rule should be kept for further evaluation and expansion of if it should be discarded. The rules for pruning a tree by using cover counts are as follows. A node (expansion of a rule) is discarded if it satisfies one of the conditions in Figure 5.1.

1.	The ⊕ cover of a rule is reduced to one by the addition of the node while the ⊖ cover remains greater than zero.
2.	The addition of the node has no effects on the ⊕ and ⊖ cover of the rule.
3.	The addition of the node only reduces the ⊕ cover of a rule and doesn't effect the ⊖ cover of the rule

Figure 5.1. Heuristic Rule for Pruning $T_S(e)$ by Cover Counts.

The heuristic rules in Figure 5.1 are aimed at pruning all nodes which do not have a positive effect upon the cover of a rule. To understand the rational behind these heuristic rules it is necessary to recall the meaning of the given tuples. The base and target tuples represent relations or properties which will be accredited to the target tuples in an attempt to explain the target concept.

The reasoning behind heuristic rule (1) is that all rules extending from this node will cover only one $\oplus$ tuple. In this case, we prefer to simply add the tuple e as a fact instead of adding a lengthy rule which covers only one target tuple.

Rule 2 has the effect of discarding any node which does not effect the cover of the rule. The reasoning behind this heuristic is that we are adding a relation or property that is shared by all $\oplus$ and $\ominus$ tuples in the rule's cover therefore the addition of this node does not contribute to any distinction between the $\oplus$ and $\ominus$ tuples in the rule's cover.

Rule 3 discards any node that reduces only the $\oplus$ cover of a rule. The reason being that the node added is a relation or property which holds for the $\ominus$ tuples within the rule's cover and not the $\oplus$ tuples, therefore it is of little use in characterizing the $\oplus$ tuples which is the main goal of a rule.

The pruning described by the three rules of Figure 5.1 is called pruning by count since it uses the counts of the rule cover. This pruning method is included as an option within the FORGE implementation.

Another option provided in the FORGE implementation is to terminate the evaluation of rules after a rule is found that is valid and covers all remaining uncovered $\oplus$ tuples. This option helps reduce the number of rule instantiations which must be performed and does not have any effect on the construction of the explanation tree.

5.2 Pruning to improve Recursive Rules

Another method of reducing the number of nodes added to an explanation tree is to limit the number of tuples within the set C of current facts at the start of the rule building process. This paradigm becomes even move important when building recursive rules. Recalling the tree building algorithm discussed in Chapter 3, we see that all base tuples and all target tuples in Cover(S) make up the set C of

current facts. Whenever we use a tuple from Cover(S) within the tree construction, we are producing a recursive rule. Generally the characteristics of a recursive rule is that it calls itself but usually with parameters that are simpler than the original rule's parameters. If we are dealing with structured arguments we may use some type of linear ordering, such as the close ordering discussed in Section 4.3, on the arguments to determine which ones are simpler than others. The rule head, tuple e, is known before the tree building process is started, therefore we prune the target tuples from the set of current facts C once before the tree building process is begun and we do not need to consider this pruning again until the next e tuple is chosen. This pruning not only limits the size of the tree, but it also helps to eliminate the construction of infinite recursive rules. This pruning method is called pruning by order and is also implemented as an option in the FORGE implementation.

5.3 Saving Maps of Instantiations

Not only is the size of the search tree a limiting factor in this algorithm, but the search involved in finding the cover of each rule is extensive. To find the cover of a general rule R one must find all possible instantiations of the literals within the body of rule R and then substitute these instantiations into the head of R. In an effort to save search time, the set of instantiations for a literal, or the mappings of variables to constants which will map the literal within the body of the rule to the tuples given, is saved with each node in the tree. When a rule is extended, these maps are applied to the new literal in the body of the rule and then it is instantiated to the tuples given in an effort to find the new map for the extended rule. This method saves considerable search time, but it also increases the amount of storage required by the implementation.

Chapter 6

Comparing the EBL Methods of FORGE with the Empirical Methods of FOIL

6.1 An Example of FOIL

Before making a comparison between the methods of FORGE with Quinlan's FOIL algorithm [33], we describe the FOIL algorithm and give a detailed example of its application.

The outermost level of FOIL is summarized in Figure 6.1.

1. Establish a training set consisting of both positive and negative constant tuples of the target relation. Label the positive tuples $\oplus$ and label the negative tuples $\ominus$.
2. Until there are no $\oplus$ tuples left in the training set do the following:
   a) Find a clause that characterizes part of the target relation.
   b) Remove all tuples that satisfy the right hand side of this clause from the training set.

Figure 6.1. Outer Loop of the FOIL Algorithm.

Step 2a from Figure 6.1 constitutes the inner loop of FOIL. Within this inner loop, FOIL Seeks a Prolog clause of the form $T(X_1, X_2, \cdots, X_k) \leftarrow L_1, L_2, \cdots, L_n$ which characterizes some subset of the target relation T. The clause is constructed by adding literals to the right hand side of the clause one at a time. The inner loop is summarized by the steps shown in Figure 6.2.

1. Initialize the local training set to the training set and let i = 1.
2. While T_i contains $\ominus$ tuples do the following:
   a) Find a literal L_i to add to the right hand side of the clause.
   b) Produce a new training set T_{i+1} based on those tuples in T_i that satisfy L_i. Label each tuple in T_{i+1} the same as that of the parent tuple in T_i.
   c) Increment i and continue.

Figure 6.2. Inner Loop of FOIL Algorithm.

If the new literal L_i chosen in step 2a of Figure 6.2 introduces new variables, then the tuples in T_i that satisfy L_i may give rise to several new expanded tuples in T_{i+1}. Quinlan denotes the number of tuples in T_i which give rise to $\oplus$ tuples in T_{i+1} as T_i^{++} and the number of $\oplus$ and $\ominus$ tuples in T_{i+1} as T_{i+1}^{+} and T_{i+1}^{-}, respectively. A good choice for the new literal L_i would be one that covered many of the $\oplus$ tuples

and few of the ⊖ tuples. FOIL uses an information-based heuristic function called the gain function which utilizes T^{+}_{i+1}, T^{-}_{i+1}, and T^{++}_{i} in determining the best candidate for the literal L_i. The gain function is calculated using the formulas shown in Figure 6.3.

1.	$I(T_i)$	=	$-\log_2(T^{+}_{i}/(T^{+}_{i}+T^{-}_{i}))$
2.	$I(T_{i+1})$	=	$-\log_2(T^{+}_{i+1}/(T^{+}_{i+1}+T^{-}_{i+1}))$
3.	$Gain(L_i)$	=	$T^{++}_{i} \times (I(T_i) - I(T_{i+1}))$

Figure 6.3. Formulae to Calculate the Gain Function of FOIL.

The first two formula of Figure 6.3 represent the information from T_i and T_{i+1}, respectively. The difference between $I(T_i)$ and $I(T_{i+1})$ represents the information gained by adding L_i to the clause. Quinlan then multiplies this value by the number of ⊕ tuples to yield the total information gained as represented in formula 3 of Figure 6.3.

To illustrate how this information is utilized when constructing a clause, we present an example application of the FOIL algorithm. The example involves learning the definition of the member relation of a list. The base relations involve the null(X) and components(X, Y, Z) relations. The null(X) relation means that X is the empty list. The components(X, Y, Z) relation means that X is a list with head Y and tail Z. The target relation member(X, Y) means that X is a member of the list Y. The given base and target relation tuples are listed in Table 6.1.

Table 6.1. Given Base and Target Tuples for Learning the Member Relation.

Base Relations	Values	
null	()	
components	(a),a,()	((a)d),(a),(d)
	(d),d,()	(b(a)d),b,((a)d)
Target Relation	**Values**	
member	a,(a)	d,(d)
	b,(b(a)d)	d,(b(a)d)
	d,((a)d)	(a),((a)d)
	(a),(b(a)d)	

The universe for this problem is {a, b, d, (), (a), (d), ((a)d), (b(a)d)} which leads to the initial FOIL training set shown in Table 6.2. The initial training set yields $T^{+}_{1} = 7$ and $T^{-}_{1} = 57$, giving $I(T_1) = -\log_2(7/64) = 3.19$. Table 6.3 contains all candidates for literal L_1 and the corresponding T^{+}_{2}, T^{-}_{2}, $I(T_2)$ and gain values. From Table 6.3 we see that the fifth clause also contains only ⊕ tuples in T_2; therefore

Table 6.2. Initial FOIL Training Set for Learning the Member Relation.

Type	Value of x, y for clause member(x, y).			
⊕:	a,(a)	d,(d)	(a),((a)d)	d,((a)d)
	(a),(b(a)d)	b,(b(a)d)	d,(b(a)d)	
⊖:	(),()	a,()	(a),()	b,()
	(d),()	d,()	((a)d),()	(b(a)d),()
	(),a	a,a	(a),a	b,a
	(d),a	d,a	((a)d),a	(b(a)d),a
	(),(a)	(a),(a)	b,(a)	(d),(a)
	d,(a)	((a)d),(a)	(b(a)d),(a)	(),b
	a,b	(a),b	b,b	(d),b
	d,b	((a)d),b	(b(a)d),b	(),(d)
	a,(d)	(a),(d)	b,(d)	(d),(d)
	((a)d),(d)	(b(a)d),(d)	(),d	a,d
	(a),d	b,d	(d),d	d,d
	((a)d),d	(b(a)d),d	(),((a)d)	a,((a)d)
	b,((a)d)	(d),((a)d)	((a)d),((a)d)	(b(a)d),((a)d)
	(),(b(a)d)	a,(b(a)d)	(d),(b(a)d)	((a)d),(b(a)d)
	(b(a)d),(b(a)d)			

Table 6.3. Candidates for L_1 for the First Iteration of FOIL.

No.	$T(X, Y) \leftarrow L_1$	T_2^+	T_2^-	$I(T_2)$	T_1^{++}	Gain
1.	member(X, Y) ← null(X)	0	8	-	0	0.0
2.	member(X, Y) ← null(Y)	0	8	-	0	0.0
3.	member(X, Y) ← components(X, Y, Z)	0	4	-	0	0.0
4.	member(X, Y) ← components(X, Z, Y)	0	4	-	0	0.0
5.	member(X, Y) ← components(Y, X, Z)	4	0	0.0	4	12.2
6.	member(X, Y) ← components(Z, X, Y)	0	4	-	0	0.0
7.	member(X, Y) ← components(Y, Z, X)	0	4	-	0	0.0
8.	member(X, Y) ← components(Z, Y, X)	0	4	-	0	0.0
9.	member(X, Y) ← components(X, Z, W)	2	30	4.0	2	-1.6
10.	member(X, Y) ← components(Z, X, W)	7	25	2.2	7	7.0
11.	member(X, Y) ← components(Z, W, X)	0	32	-	0	0.0
12.	member(X, Y) ← components(Y, Z, W)	7	25	2.2	7	7.0
13.	member(X, Y) ← components(Z, Y, W)	1	31	5.0	1	-1.8
14.	member(X, Y) ← components(Z, W, Y)	3	29	3.42	3	-0.7

clause 5 is chosen as a rule and the tuples that it covers are removed from the training set. The resulting training set is shown in Table 6.4. FOIL now enters the inner loop of the algorithm for the second time with the updated training set of Table 6.4 which yields the values $T_1^+ = 3$, $T_{1\_} = 57$ and $I(T_1) = 4.32$. Since the updated training set is a subset of the previous training set, it is obvious that the resulting T_2^+ of this iteration of the loop will be less than or equal to the T_1^+ of the previous iteration. Because of this fact, FOIL does not consider any L_1 which previously lead to a T_1^+ of 0. The resulting candidates for L_1 for the second rule and their information and gain values are shown in Table 6.5. Inspection of the gain

Table 6.4. Second FOIL Training Set for Learning Member Relation.

Type	Value of X, Y for clause member(X, Y).			
⊕:	d,((a)d)	(a),(b(a)d)	d,(b(a)d)	
⊖:	(),()	a,()	(a),()	b,()
	(d),()	d,()	((a)d),()	(b(a)d),()
	(),a	a,a	(a),a	b,a
	(d),a	d,a	((a)d),a	(b(a)d),a
	(),(a)	(a),(a)	b,(a)	(d),(a)
	d,(a)	((a)d),(a)	(b(a)d),(a)	(),b
	a,b	(a),b	b,b	(d),b
	d,b	((a)d),b	(b(a)d),b	(),(d)
	a,(d)	(a),(d)	b,(d)	(d),(d)
	((a)d),(d)	(b(a)d),(d)	(),d	a,d
	(a),d	b,d	(d),d	d,d
	((a)d),d	(b(a)d),d	(),((a)d)	a,((a)d)
	b,((a)d)	(d),((a)d)	((a)d),((a)d)	(b(a)d),((a)d)
	(),(b(a)d)	a,(b(a)d)	(d),(b(a)d)	((a)d),(b(a)d)
	(b(a)d),(b(a)d)			

Table 6.5. Candidates for L_1 for the Second Iteration of FOIL.

No.	member(X, Y) ← L_1	T_2^+	T_2^-	$I(T_2)$	T_2^{++}	Gain
1.	member(X, Y) ← components(X, Z, W)	1	30	11.41	1	-7.1
2.	member(X, Y) ← components(Z, X, W)	3	25	3.22	3	3.3
3.	member(X, Y) ← components(Y, Z, W)	3	25	3.22	3	3.3
4.	member(X, Y) ← components(Z, Y, W)	0	31	-	0	0.0
5.	member(X, Y) ← components(Z, W, Y)	1	29	4.91	1	-0.6

column in Table 6.5 shows that clauses 2 and 3 have the same gain value. Further inspection shows that the only ⊕ tuples left are "d,((a)d)", "(a),(b(a)d)", and "d,(b(a)d)". Clause 2 of table 6.5 states that the first member of these tuples is the head of some list. The third clause states that the second member of these tuples is a list with components. Both of these clauses hold that same amount of information about the current training set. At his point FOIL must decide which clause to choose for expansion. If the wrong clause is chosen, the first version of FOIL would have failed to find a correct definition of the member relation. However, more recent versions of FOIL have incorporated backtracking to remedy this problem. For the example at hand, we will assume that FOIL chooses the third clause to expand. The third clause contains the variables x, y, z and w which gives rise to a new training set shown in Table 6.6. Notice that the tuples "a,(a),a,()", "b,(b(a)d),b,((a)d)" and "d,(d),d,()" are not included in the ⊕ tuples of the training set shown in Table 6.6 since they are covered by a previously generated rule. The resulting expansion of the third clause of Table 6.5 is shown in Table 6.7. Notice that at this point, FOIL considers including the target relation as the first literal in an effort to eliminate infinite recursive

Table 6.6. Third FOIL Training Set for Learning the Member Relation.

Type	Value of X, Y, Z, W for clause member(X, Y) ← components(Y, Z, W).			
⊕:	d,((a)d),(a),(d)	(a),(b(a)d),b,((a)d)	d,(b(a)d),b,((a)d)	
⊖:	(),(a),a()	(),(b(a)d),b,((a)d)	(),((a)d),(a),(d)	(),(d),d,()
	(a),(a),a()	a,(b(a)d),b,((a)d)	a,((a)d),(a),(d)	a,(d),d,()
	b,(a),a()	(d),(b(a)d),b,((a)d)	b,((a)d),(a),(d)	(a),(d),d,()
	(d),(a),a()	((a)d),(b(a)d),b,((a)d)	(d),((a)d),(a),(d)	b,(d),d,()
	d,(a),a()	(b(a)d),(b(a)d),b,((a)d)	((a)d),((a)d),(a),(d)	(d),(d),d,()
	((a)d),(a),a()	(b(a)d),((a)d),(a),(d)	((a)d),(d),d,()	(b(a)d),(d),d,()
	(b(a)d),(a),a()			

Table 6.7. Candidates for L_2 at the Third Iteration of FOIL.

No.	member(X, Y) ← L_1, L_2.	T_3^+	T_3^-
1.	member(X, Y) ← components(Y, Z, W), null(Z).	0	0
2.	member(X, Y) ← components(Y, Z, W), null(W).	0	14
3.	member(X, Y) ← components(Y, Z, W), components(z, x, w).	0	0
4.	member(X, Y) ← components(Y, Z, W), member(X, Z).	0	1
5.	member(X, Y) ← components(Y, Z, W), member(X, W).	3	0
6.	member(X, Y) ← components(Y, Z, W), member(Y, Z).	0	0
7.	member(X, Y) ← components(Y, Z, W), member(Y, W).	0	0
8.	member(X, Y) ← components(Y, Z, W), member(Z, W).	0	0
9.	member(X, Y) ← components(Y, Z, W), member(W, Z).	0	0

rules. In addition, FOIL also assumes a partial ordering on the variables within the current clause to further eliminate infinite recursive rules. In the current iteration we have the clause 'member(X, Y) ← components(Y, Z, W)' for which FOIL assumes the partial ordering X < Y, Y < Z and Y < W. This partial ordering implies the X < Z and X < W. We see in Table 6.7 that member(X, Z) and member(X, W) are considered as candidates for L_2 because they are allowed by FOIL's partial ordering, but the clauses member(Z, X) and member(W, X) are eliminated by the partial ordering.

We also see from Table 6.7 that clause 5 covers all remaining ⊕ tuples and therefore is accepted as the second rule. The resulting rule set for the member relation is

member(X, Y) ← components(Y, X, Z).

member(X, Y) ← components(Y, Z, W), member(X, W).

The number of clauses that were constructed and evaluated in this example of FOIL is 28. If, during the second iteration when two clauses were constructed which had the exact same gain value, FOIL would have chosen the second clause instead of the third clause, the algorithm would have built many more

rules in the expansion of the second clause. These extra clauses would have necessarily been discarded during the backtracking process.

6.2 An Example of FORGE

To compare FORGE with FOIL we now present the steps of the FORGE algorithm when learning the same member relation. We again display the base and target tuples for the member relation in Table 6.8 for ease of reference. The universe for this problem is {a, b, d, (), (a), (d), ((a)d), (b(a)d)}. We will let the order of the elements within the universe determine the linear order to be used to prevent infinite recursive rules from being formed.

Table 6.8. Given Base and Target Tuples for Learning the Member Relation.

Base Relations	Values	
null	()	
components	(a),a,()	((a)d),(a),(d)
	(d),d,()	(b(a)d),b,((a)d)
Target Relation	**Values**	
member	a,(a)	d,(d)
	b,(b(a)d)	d,(b(a)d)
	d,((a)d)	(a),((a)d)
	(a),(b(a)d)	

The first step in the FORGE algorithm is to pick a tuple e in the target relation and build the first level of the explanation tree for this e. We choose e = member(a, (a)) and show the explanation tree in Figure 6.4.

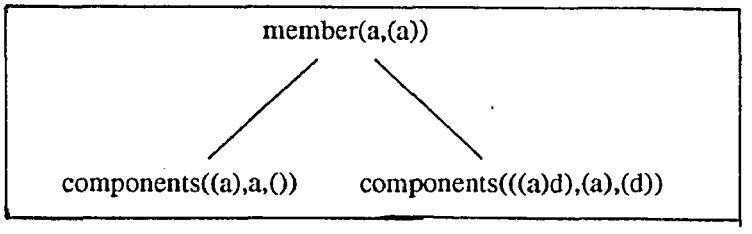

Figure 6.4. Explanation Tree T_S(member(a, (a))).

The generalized rules resulting from the tree of Figure 6.4 and their ⊕ and ⊖ cover values are shown in Table 6.9.

Table 6.9. Resulting General Rules from T_S(member(a, (a))).

No.	Rule	⊕ Cover	⊖ Cover
1.	member(X, Y) ← components(Y, X, Z)	4	0
2.	member(X, Y) ← components(X, Y, W)	1	31

From Table 6.9 we see that Rule 1 is a valid rule and should be added to the set S of valid rules. Since S is empty, we simply add Rule 1 to S without considering reducing S. We now remove all ⊕ tuples covered by S from our set of target tuples and add them as a given base relation since we now have a rule which defines them. The resulting base and target relation tuples are shown in Table 6.10.

Table 6.10. Updated FORGE Base and Target Tuples for Second Iteration.

Base Relations	Values	
null	()	
components	(a),a,()	((a)d),(a),(d)
	(d),d,()	(b(a)d),b,((a)d)
member	a,(a)	b,(b(a)d)
	d,(d)	(a),((a)d)
Target Relation	**Values**	
member	d,((a)d)	d,(b(a)d)
	(a),(b(a)d)	

We choose the next *e* as member(d,((a)d)) and build its explanation tree as shown in Figure 6.5. The tree in Figure 6.5 is shown in a horizontal fashion instead of the usual vertical fashion to save space in the figure. Notice that the nodes are added in the order of their links to the arguments of the target tuple. In Figure 6.5 we see that all tuples that are linked to the target "member(d,((a)d))" by the argument "d" are listed first followed by all the tuples linked to "((a)d)". We also notice that the tuple "member((a),((a)d)" is not considered as a valid extension to the rule. The heuristic used in this case is that at least one corresponding argument of the tuple considered for addition must be less than the corresponding argument of the target tuple before it may be added to the rule. Since no corresponding argument of the tuple "member((a),((a)d)" is less than the corresponding arguments of the target tuple, i.e., (a) is not less than d and ((a)d) is not less than ((a)d), this tuple is not added to the rule.

The resulting general rules from the tree of Figure 6.5 are shown in Table 6.11 along with their coverage values.

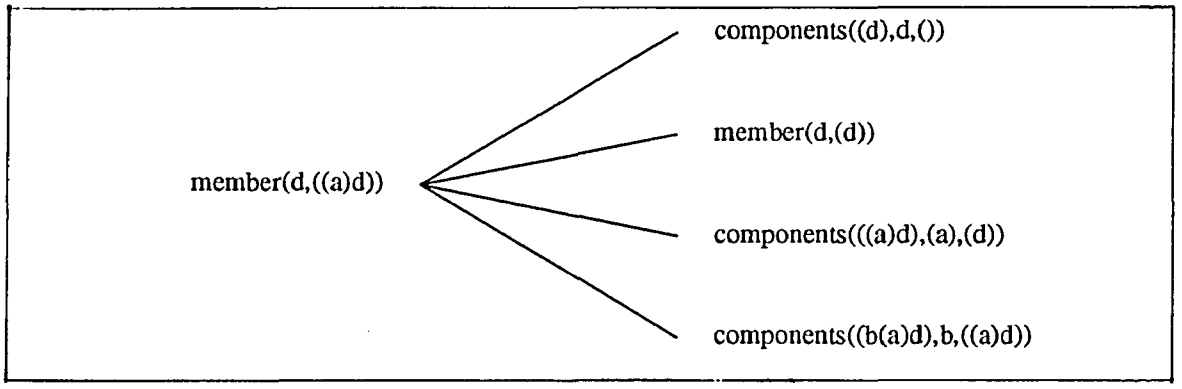

Figure 6.5. Level One of T_S(member(d, ((a)d))).

Table 6.11. General Rules from the First Level of T_S(member(d, ((a)d))).

No.	Generalized Rule	⊕ Cover	⊖ Cover
1.	member(X, Y) ← components(Z, X, W)	3	25
2.	member(X, Y) ← member(X, Z)	3	26
3.	member(X, Y) ← components(Y, Z, W)	3	25
4.	member(X, Y) ← components(Z, W, Y)	1	21

Notice that the ⊕ coverage is the number of ⊕ target tuples covered by this rule and not covered by rules in the set S. The ⊕ and ⊖ cover values in Table 6.11 show that there are no valid rules in this level of the tree. FORGE expands the tree to the next level as shown in Figure 6.6. If we were pruning the tree by cover counts we would not expand the fourth branch of the tree since the rule it produced covers only one ⊕ tuple and many ⊖ tuples. For completeness we will not consider pruning in this example.

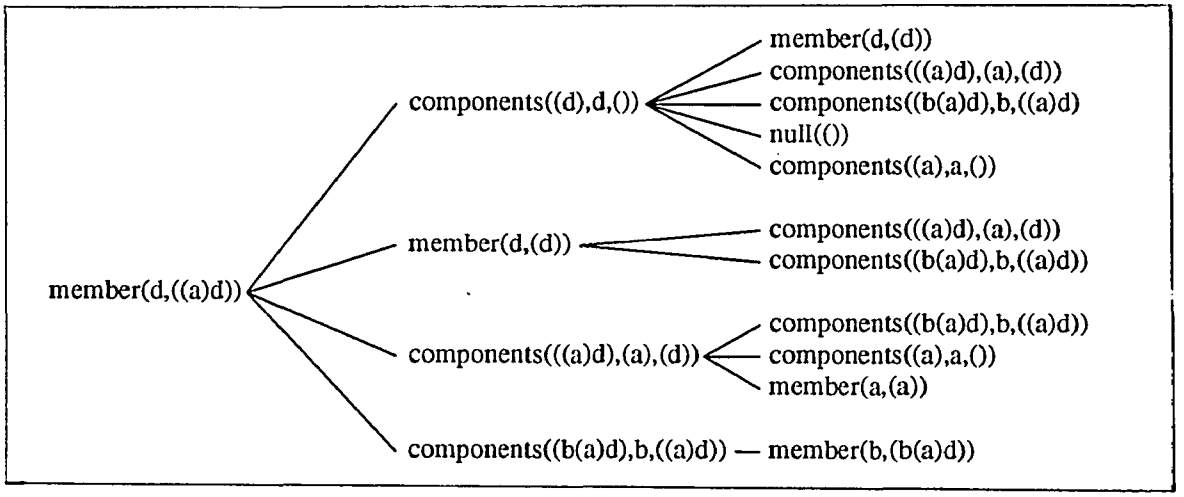

Figure 6.6. T_S(member(d, ((a)d))) Expanded to the Second Level.

The tree in Figure 6.6 yields eleven general rules which are listed in Table 6.12.

Table 6.12. General Rules Resulting from Expansion of T_S(member(d, ((a)d))).

No.	Generalized Rule	⊕ Cover	⊖ Cover
1.	member(X, Y) ← components(Z, X, W), member(X, Z)	3	25
2.	member(X, Y) ← components(Z, X, W), components(Y, V, Z)	2	0
3.	member(X, Y) ← components(Z, X, W), components(V, U, Y)	1	9
4.	member(X, Y) ← components(Z, X, W), null(W)	2	12
5.	member(X, Y) ← components(Z, X, W), components(V, U, W)	3	25
6.	member(X, Y) ← member(X, Z), components(Y, V, Z)	3	0
7.	member(X, Y) ← member(X, Z), components(V, U, Y)	1	9
8.	member(X, Y) ← components(Y, Z, V), components(U, W, Y)	1	13
9.	member(X, Y) ← components(Y, Z, V), components(Z, U, W)	1	6
10.	member(X, Y) ← components(Y, Z, V), member(U, Z)	1	6
11.	member(X, Y) ← components(Z, V, Y), member(V, Z)	1	14

As shown by the ⊕ and ⊖ cover counts in Table 6.12, we see that rules 2 and 6 are valid, but rule 6 has the best coverage so rule 6 is chosen to be added to S. Since there remain no uncovered target tuples, the algorithm terminates with the set S of valid rules containing the following two rules.

member(X, Y) ← components(Y, X, Z).

member(X, Y) ← member(X, Z), components(Y, V, Z).

Since we are considering the order of the literals within a rule to be nonsignificant, we consiser the second rule of above to be the same as "member(X, Y) ← components(Y, V, Z), member(X, Z)".

6.3 Comparison of FORGE and FOIL

The resulting set of rules for FORGE and FOIL are the same, however the number of rules constructed and evaluated by each are not the same. The FOIL algorithm constructed and evaluated 28 rules while FORGE constructed and evaluated only 17 rules. If we had not chosen the correct clause to expand within the second iteration of FOIL, when two clauses had the exact same gain value, FOIL would have constructed and evaluated many more than 28 rules. On the other hand, FORGE could possibly have evaluated less than 17 rules if we had chosen the option of pruning the tree.

Other examples of FOIL generating more rules than FORGE are taken from Quinlan [33] and include learning the definition of the list relation. The input for this problem is found in Table 6.13 and resulting rules evaluated by FOIL and FORGE are listed in Tables 6.14 and 6.15, respectively.

Table 6.13. Input Base and Target Realtion for Learning the List Relation.

Base Relations	Values	
null	()	
components	(a),a,()	((a)d),(a),(d)
	(d),d,()	(b(a)d),b,((a)d)
	(e.f),e,f	
Target Relation	Values	
list	()	(a)
	(d)	((a)d)
	(b(a)d)	

We see from Table 6.14 that FOIL again must make a choice between rules No. 2 and No. 4 in the first iteration for expansion within the second iteration. If rule No. 4 from the first iteration had been chosen for expansion, FOIL would have had to backtrack after exploring all of the expansions of rule No. 4. We assume FOIL will choose rule No. 2 from the first iteration to be expanded. FOIL then produces and evaluates 31 more rules before finding a suitable set of rules.

From Table 6.15 we see that FORGE produces a total of 9 rules compared to a total of 35 rules produced by FOIL. If we consider pruning the second explanation tree produced by FORGE, rule No. 2 would have been discarded and subsequently rules 6 and 7 would not have been produced leaving a total of 7 rules produced and evaluated by FORGE.

Table 6.14. Rules Evaluated by FOIL when Learning the List Relation.

No.	Rules from the First Iteration	Cover ⊕	Cover ⊖
1.	list(X) ← null(X).	1	0
2.	list(x) ← components(X, Y, Z)	4	1
3.	list(x) ← components(Y, X, Z)	1	4
4.	list(x) ← components(Y, Z, X)	4	1

No.	Rules from the Second Iteration	Cover ⊕	Cover ⊖
1.	list(X) ← components(X, Y, Z), null(Y)	0	0
2.	list(X) ← components(X, Y, Z), null(Z)	2	0
3.	list(X) ← components(X, Y, Z), components(X, Y, W)	4	1
4.	list(X) ← components(X, Y, Z), components(Y, X, W)	0	0
5.	list(X) ← components(X, Y, Z), components(X, W, Y)	0	0
6.	list(X) ← components(X, Y, Z), components(Y, W, X)	0	0
7.	list(X) ← components(X, Y, Z), components(W, X, Y)	0	0
8.	list(X) ← components(X, Y, Z), components(W, Y, X)	0	0
9.	list(X) ← components(X, Y, Z), components(X, W, Z)	4	1
10.	list(X) ← components(X, Y, Z), components(Z, W, x)	0	0
11.	list(X) ← components(X, Y, Z), components(W, X, Z)	0	0
12.	list(X) ← components(X, Y, Z), components(W, Z, X)	0	0
13.	list(X) ← components(X, Y, Z), components(X, Z, W)	4	1
14.	list(X) ← components(X, Y, Z), components(Z, X, W)	1	0
15.	list(X) ← components(X, Y, Z), components(W, Y, Z)	4	1
16.	list(X) ← components(X, Y, Z), components(W, Z, Y)	0	0
17.	list(X) ← components(X, Y, Z), components(Y, W, Z)	0	0
18.	list(X) ← components(X, Y, Z), components(Z, W, Y)	0	0
19.	list(X) ← components(X, Y, Z), components(Y, Z, W)	0	0
20.	list(X) ← components(X, Y, Z), components(Z, Y, W)	0	0
21.	list(X) ← components(X, Y, Z), components(X, W, V)	4	1
22.	list(X) ← components(X, Y, Z), components(W, X, V)	1	0
23.	list(X) ← components(X, Y, Z), components(W, V, X)	2	0
24.	list(X) ← components(X, Y, Z), components(Y, W, V)	1	0
25.	list(X) ← components(X, Y, Z), components(W, Y, V)	4	1
26.	list(X) ← components(X, Y, Z), components(W, V, Y)	0	0
27.	list(X) ← components(X, Y, Z), components(Z, W, V)	2	0
28.	list(X) ← components(X, Y, Z), components(W, Z, V)	0	0
29.	list(X) ← components(X, Y, Z), components(W, V, Z)	6	1
30.	list(X) ← components(X, Y, Z), list(Y)	1	0
31.	list(X) ← components(X, Y, Z), list(Z)	4	0

Table 6.15. Rules Evaluated by FORGE when Learning the List Relation.

No.	Rules from First Explanation Tree	Cover ⊕	Cover ⊖
1.	list(X) ← null(X)	1	0
2.	list(X) ← components(Y, Z, X)	4	1

No.	Rules from Second Explanation Tree	Cover ⊕	Cover ⊖
1.	list(X) ← components(X, Y, Z)	4	1
2.	list(X) ← components(Y, X, Z)	1	4
3.	list(X) ← components(X, Y, Z), null(Z)	2	0
4.	list(X) ← components(X, Y, Z), list(Z)	4	0
5.	list(X) ← components(X, Y, Z), components(W, V, Z)	4	1
6.	list(X) ← components(Y, X, Z), components(W, V, Y)	1	1
7.	list(X) ← components(Y, X, Z), components(Z, W, V)	1	1

Both FOIL and FORGE produce the same set of rules for the list relation. These rules are listed in Table 6.16. We observe however, that FORGE produced and evaluated a total of 9 rules while FOIL produced and evaluated a total of 35 rules.

Table 6.16. Rules for List Relation from FOIL and FORGE.

list(X) ← null(X).
list(X) ← components(X, Y, Z), list(Z).

In the examples shown in this chapter, we see that FOIL created and evaluated more rules than FORGE. By the nature of the addition of literals to the clauses in FOIL we can see that FOIL will construct literals which contain every possible permutation for the variables involved. In effect, FORGE only adds literals to clauses if there is at least one tuple in the target relation that is satisfied by this additional literal. FOIL adds literals to clauses even if there are no examples in the data which will satisfy the resulting clause causing FOIL to evaluate rules which do not cover any given tuples. All rules created by FORGE, on the other hand, will cover at least one of given target tuples. This difference is due to the fact that FORGE utilizes knowledge from the given input tuples while constructing rules. This EBL technique guides FORGE in the selection of literals to be added to clauses, thus resulting in the smaller number of rules generated by FORGE.

There are cases when FORGE will produce more rules than FOIL. These cases are characterized by input base and target tuples which contain many common constant arguments and a target relation which has few arguments. In this case, since the target relation has few arguments, FOIL will have

fewer arguments to permute. FORGE on the other hand will have many links in common with the target example and will construct more branches in the explanation tree.

The partial ordering adopted by FOIL in an attempt to prevent the construction of infinite recursive rules must be derived for each different set of clauses that are considered. FORGE adopts a partial ordering of the elements of the universe once at the initial stage of the algorithm and does not have to re-define the ordering for any rule.

Both the FORGE and FOIL algorithms are exponential in search time. FOIL searches one path at a time in a depth-first fashion while FORGE searches all paths at the same time in a breadth-first fashion. The reason for implementing a breadth-first search in FORGE was in an effort to find the shortest rules first. This decision was supported by the heuristic that shorter rules are simpler rules. The differences in the search procedures of the two algorithms produce the need for backtracking in the FOIL algorithm, while producing larger storage requirements in the FORGE algorithm.

Another implementation of FORGE could implement a depth-first search in place of the breadth-first search. This would reduce the memory requirements to be the same as the memory requirements of FOIL, but would remove the assurance that shorter rules would be found before longer rules.

Chapter 7

Example Runs of FORGE

In this chapter, we present some example runs of the FORGE implementation and examine some effects of different input and options on the output.

7.1 Example 1 of Canreach Relation

The first example presented is a graph example. The input consisted of a graph with five nodes and five forward arcs or links in the graph. The target relation is called canreach and denotes the first node being linked by a path of one or more forward arcs to the second node. The example input and output follows.

INPUT:

```
(link 0 1) (link 1 2) (link 0 3) (link 2 5) (link 3 2)
*end*
(obj-order (0 1) (1 2) (2 3) (3 5))
(reln-order link canreach)
(canreach 0 1) (canreach 0 2) (canreach 0 3)
(canreach 0 5) (canreach 1 2) (canreach 1 5)
(canreach 2 5) (canreach 3 2) (canreach 3 5)
*end*
(rules-subsumed nil)
(count-tree-nodes t)
(rules-evaluated nil)
(prune-by-count nil)
(prune-by-order nil)
```

OUTPUT:

```
The linear order derived from obj-order is (0 1 2 3 5)

Universe is (0 1 2 3 5)

INPUT BASE TUPLES:
((link 0 1) (link 0 3) (link 1 2) (link 2 5) (link 3 2))

INPUT TARGET TUPLES:
((canreach 0 1) (canreach 0 2) (canreach 0 3) (canreach 0 5) (canreach 1 2)
 (canreach 1 5) (canreach 2 5) (canreach 3 2) (canreach 3 5))
```

**** ITERATION 1 ****
The next tuple chosen for e is: (canreach 0 1)

Valid rules found are:
Rule N2: ((canreach 0 1) (link 0 1))
Cover: ((canreach 0 1) (canreach 0 3) (canreach 1 2)
(canreach 2 5) (canreach 3 2))

Adding to S: ((canreach Y1 Y2) (link Y1 Y2))

Uncovered target tuples:
((canreach 0 2) (canreach 0 5) (canreach 1 5) (canreach 3 5))

**** ITERATION 2 ****
The next tuple chosen for e is: (canreach 0 2)

Valid rules found are:
Rule N19: ((canreach 0 2) (link 0 1) (link 1 2))
Cover: ((canreach 0 2) (canreach 1 5) (canreach 3 5))

Rule N22: ((canreach 0 2) (link 0 1) (canreach 1 2))
Cover: ((canreach 0 2) (canreach 0 5) (canreach 1 5) (canreach 3 5))

Rule N34: ((canreach 0 2) (canreach 0 1) (link 1 2))
Cover: ((canreach 0 2) (canreach 0 5) (canreach 1 5) (canreach 3 5))

Rule N37: ((canreach 0 2) (canreach 0 1) (canreach 1 2))
Cover: ((canreach 0 2) (canreach 0 5) (canreach 1 5) (canreach 3 5))

Rules selected from valid rules are: (N22)

Adding to S: ((canreach Y1 Y3) (link Y1 Y2) (canreach Y2 Y3))

Uncovered target tuples: nil

Rule: N2
Ground form of rule:
((canreach 0 1) (link 0 1))
General rule:
((canreach Y1 Y2) (link Y1 Y2))

Rule: N22
Ground form of rule:
((canreach 0 2) (link 0 1) (canreach 1 2))
General rule:
((canreach Y1 Y3) (link Y1 Y2) (canreach Y2 Y3))

Count of nodes constructed:
(0 1)(1 3)
(0 1)(1 10)(2 45)

Notice the object order defined within the input file corresponds to our usual ordering of integers. In this case the algorithm found the usual two rules for defining the relation canreach. The last thing listed in the output is the count of nodes constructed. Each line represents an iteration of the algorithm or the construction of one explanation tree. The first element of each ordered pair represents the level of the tree and the second element of the ordered pairs represents the number of nodes added to that level of the tree. Each tree starts off with one root node at level zero. It took two iterations to find the two rules and a total of three nodes were added to the explanation tree in finding the first rule. A total of 55 nodes were added to the tree when constructing the second rule. No pruning options were considered in this example.

The rules produced by the algorithm immediately precede the count of nodes constructed. In this case the usual two rules defining the canreach relation were constructed.

The proper recursive rule was constructed on the second iteration of this example because the basis rule for the recursion was already constructed by the first iteration. This illustrates the lack of need for absorption in this case and is supported by Lemma 5 of Section 4.2.

7.2 Example 2 of Canreach Relation

The next example uses the same input as the previous example, except that the object order has been changed. In this case the odering is just the opposite of our usual ordering of integers. The target concept is also the same as in the previous example. The example input and output is listed next.

INPUT:

```
(link 0 1) (link 1 2) (link 0 3) (link 2 5) (link 3 2)
*end*
(obj-order (5 3)(3 2)(2 1)(1 0))
(reln-order link canreach)
(canreach 0 1) (canreach 0 2) (canreach 0 3) (canreach 0 5)
(canreach 1 2) (canreach 1 5) (canreach 2 5) (canreach 3 2)
(canreach 3 5)
*end*
(rules-subsumed nil)
(count-tree-nodes t)
(rules-evaluated nil)
(prune-by-count nil)
(prune-by-order nil)
```

OUTPUT:

```
The linear order derived from obj-order is (5 3 2 1 0)

Universe is (5 3 2 1 0)

INPUT BASE TUPLES:
((link 3 2) (link 2 5) (link 1 2) (link 0 3) (link 0 1))

INPUT TARGET TUPLES:
((canreach 3 5) (canreach 3 2) (canreach 2 5) (canreach 1 5) (canreach 1 2)
 (canreach 0 5) (canreach 0 3) (canreach 0 2) (canreach 0 1))

**** ITERATION 1 ****
The next tuple chosen for e is: (canreach 3 5)

Valid rules found are:
Rule N6: ((canreach 3 5) (link 3 2) (link 2 5))
Cover: ((canreach 3 5) (canreach 1 5) (canreach 0 2))

Adding to S: ((canreach Y2 Y1) (link Y2 Y3) (link Y3 Y1))

Uncovered target tuples: ((canreach 3 2) (canreach 2 5) (canreach 1 2)
                          (canreach 0 5) (canreach 0 3) (canreach 0 1))

**** ITERATION 2 ****
The next tuple chosen for e is: (canreach 3 2)

Valid rules found are:
Rule N12: ((canreach 3 2) (link 3 2))
Cover: ((canreach 3 2) (canreach 2 5) (canreach 1 2) (canreach 0 3)
        (canreach 0 1))

Rules selected from valid rules are: (N12)

Adding to S: ((canreach Y2 Y3) (link Y2 Y3))

Uncovered target tuples: ((canreach 0 5))

**** ITERATION 3 ****
The next tuple chosen for e is: (canreach 0 5)

Valid rules found are:
Rule N33: ((canreach 0 5) (link 0 3) (canreach 3 5))
Cover: ((canreach 3 5) (canreach 1 5) (canreach 0 5) (canreach 0 2))

Rule N49: ((canreach 0 5) (canreach 0 2) (link 2 5))
Cover: ((canreach 3 5) (canreach 1 5) (canreach 0 2) (canreach 0 5))

Rules selected from valid rules are: (N33)

Removing from S: ((canreach Y2 Y1) (link Y2 Y3) (link Y3 Y1))
```

```
Adding to S: ((canreach Y5 Y1) (link Y5 Y2) (canreach Y2 Y1))

Uncovered target tuples: nil

Rule: N12
  Ground form of rule:
   ((canreach 3 2) (link 3 2))
  General rule:
   ((canreach Y2 Y3) (link Y2 Y3))

Rule: N33
  Ground form of rule:
   ((canreach 0 5) (link 0 3) (canreach 3 5))
  General rule:
   ((canreach Y5 Y1) (link Y5 Y2) (canreach Y2 Y1))

Count of nodes constructed:
(0 1)(1 3)(2 6)
(0 1)(1 6)
(0 1)(1 9)(2 60)
```

We notice that in this example, the basis rule for the recursion was not constructed as the first rule. The first tuple chosen for *e* was an example of two nodes that were linked by a path of length 2. The algorithm added a total of 9 nodes to the tree in finding the first rule which covers each tuple which represents a path of length 2. The second iteration chooses and *e* that is and example of two nodes linked by a path of length 1. The basis rule for the recursion was found in the second iteration which required a total of 6 nodes to be added to the tree. Notice that in the first example only 3 nodes were added to the tree when constructing the basis rule. In this case the additional 3 nodes result from the nodes covered by the first rule already constructed. This illustrates that the use of previously covered target tuples in the construction of new rules increases the searching done by the algorithm, but also allows it to discover the important recursive rules.

Even though the basis rule was not discovered first, the algorithm does find the usual set of two rules to define the concept of canreach. The third rule constructed covers all the tuples that the first rule covered in addition to the remaining uncovered tuples. In this case the reduction of the set S of valid rules causes the removal of the first rule constructed before the third rule is added to S. The output of iteration 3 shows that the first rule is removed before the third rule is added to the set S. The reduction process helps in producing the smallest set of valid rules for S.

The reduction of S in this example was possible since a tuple which represented a path of length one was chosen before a tuple which represented a longer path. If, however, we had chosen the tuples in order such that the longest path was represented first, followed by the next longest path and so forth, we would have constructed a rule for each tuple chosen and no reduction in S could have been accomplished. The resulting set S would have been a valid set of rules which covered the input target tuples, but it would have not been the smallest possible set of valid rules to cover the input target tuples.

7.3 Example of Member Relation

We now present the member example of Chapter 6. In this example we choose the special options of pruning by order and stopping once the rule evaluated is valid and covers all remaining $\oplus$ tuples to limit the growth of the tree and to limit the number of rules evaluated. The input base relation consists of the null relation denoted by (null X) which implies that X is nil and the components relation denoted by (comp X Y Z) which implies that X is a list with head Y and tail Z. The target relation is (member X Y) which implies that X is a member of the list Y.

INPUT:

```
(null ())          (comp (a) a ())       (comp (d) d ())
(comp ((a)d) (a) (d))  (comp (b(a)d) b ((a)d))
*end*
(obj-order (a b) (b d) (() (a))((a) (b))((b) (d))((d) ((a)))
       (((a)) ((a)d)) (((a)d) (b(a)))((b(a)) (b(a)d)))
(reln-order null comp member)
(member a (a))     (member b (b(a)d))   (member d ((a)d))
(member (a) (b(a)d)) (member d (d))      (member d (b(a)d))
(member (a) ((a)d))
*end*
(rules-subsumed t)
(count-tree-nodes t)
(rules-evaluated t)
(prune-by-count nil)
(prune-by-order t)
```

OUTPUT:

```
The linear order derived from obj-order is
(a b d nil (a) (b) (d) ((a)) ((a) d) (b (a)) (b (a) d))

Universe is
(a b d nil (a) (b) (d) ((a)) ((a) d) (b (a)) (b (a) d))
```

INPUT BASE TUPLES:
((null nil) (comp (a) a nil) (comp (d) d nil) (comp ((a) d) (a) (d))
(comp (b (a) d) b ((a) d)))

INPUT TARGET TUPLES:
((member a (a)) (member b (b (a) d)) (member d (d)) (member d ((a) d))
(member d (b (a) d)) (member (a) ((a) d)) (member (a) (b (a) d)))

**** ITERATION 1 ****
The next tuple chosen for e is: (member a (a))

Valid rules found are:
Rule N2: ((member a (a)) (comp (a) a nil))
Cover: ((member a (a)) (member d (d)) (member (a) ((a) d)) (member b (b (a) d)))

Adding to S: ((member Y1 Y5) (comp Y5 Y1 Y4))

Uncovered target tuples:
((member d ((a) d)) (member d (b (a) d)) (member (a) (b (a) d)))

**** ITERATION 2 ****
The next tuple chosen for e is: (member d ((a) d))

Excluding from Current Facts: (member (a) ((a) d))

Valid rules found are:
Rule N14: ((member d ((a) d)) (member d (d)) (comp ((a) d) (a) (d)))
Cover: ((member d ((a) d)) (member d (b (a) d)) (member (a) (b (a) d)))

Rules selected from valid rules are: (N14)

Adding to S: ((member Y3 Y9) (member Y3 Y7) (comp Y9 Y5 Y7))

Uncovered target tuples: nil

Rule: N2
  Ground form of rule:
   ((member a (a)) (comp (a) a nil))
  General rule:
   ((member Y1 Y5) (comp Y5 Y1 Y4))

Rule: N14
  Ground form of rule:
   ((member d ((a) d)) (member d (d)) (comp ((a) d) (a) (d)))
  General rule:
   ((member Y3 Y9) (member Y3 Y7) (comp Y9 Y5 Y7))

Count of nodes constructed:
(0 1)(1 2)
(0 1)(1 4)(2 11)

List of rules evaluated:
N2: ((member a (a)) (comp (a) a nil))
N3: ((member a (a)) (comp ((a) d) (a) (d)))
N5: ((member d ((a) d)) (comp (d) d nil))
N6: ((member d ((a) d)) (member d (d)))
N7: ((member d ((a) d)) (comp ((a) d) (a) (d)))
N8: ((member d ((a) d)) (comp (b (a) d) b ((a) d)))
N9: ((member d ((a) d)) (comp (d) d nil) (member d (d)))
N10: ((member d ((a) d)) (comp (d) d nil) (comp ((a) d) (a) (d)))
N11: ((member d ((a) d)) (comp (d) d nil) (comp (b (a) d) b ((a) d)))
N12: ((member d ((a) d)) (comp (d) d nil) (null nil))
N13: ((member d ((a) d)) (comp (d) d nil) (comp (a) a nil))
N14: ((member d ((a) d)) (member d (d)) (comp ((a) d) (a) (d)))

We notice that this output also includes a list of rules that have been evaluated. In this case we see that 12 of the total 17 rules have been evaluated. The rule evaluation was stopped when a valid rule was found that covered all remaining $\oplus$ tuples. The resulting set of rules for S is the expected set of rules to define the member relation. Also notice that the tuple "member((a) ((a) d))" has been excluded from the current facts when constructing rules for the tuple "member(d ((a)d))". This is because it does not have at least one argument that is less than the corresponding target tuple argument.

7.4 Example of Append Relation

In the append relation example, we see how the proposed heuristics help in pruning the explanation tree and we also see how the reduction of the set S of valid rules helps in obtaining a minimum set of valid rules. The following is a listing of the output to learn the append relation for lists. In all 2472 nodes were constructed, but only 1044 rules were evaluated. All of the pruning and evaluation options were taken in this run to reduce the size of the problem. The prune by count option allowed 67 nodes to be pruned during iteration 5 and 102 nodes during iteration 6. Each of these nodes were pruned because the addition of the node did not improve the coverage of the resulting rule. The nodes pruned are listed in the output of each iteration and an (=) indication is placed after the node number indicating that the cover before and after the addition of the node was equal. The two resulting rules are listed toward the end of the output. The first rule is the basis rule for the recursion and the second rule is the usual recursive definition of append. A total of six rules were found, but through the reduction process of the set S, four of the constructed rules were discarded. The input base relations consists of (null X) which implies that X is nil and (comp X Y Z) that implies that X is a list with head Y and tail Z. The target relation is

represented by (append X Y Z) which implies that X appended to list Y yields list Z. The example input and output for this example is listed below.

INPUT:

```
(null ())                (comp (b(a)d) b ((a)d))
(comp ((a)d) (a) (d))    (comp (b(a)) b ((a)))
(comp ((a)) (a) ())      (comp (a)  a  ())
(comp (b)  b  ())        (comp (d)  d  ())
(comp () () ())
*end*

(obj-order (a b)        (b d)        (() (a))
       ((a) (b))     ((b) (d))      ((d) ((a)))
       (((a)) ((a)d)) (((a)d) (b(a))) ((b(a)) (b(a)d)))
(reln-order null comp append)

(append () (b(a)d) (b(a)d))    (append (b) ((a)d) (b(a)d))
(append (b(a)) (d) (b(a)d))    (append (b(a)d) () (b(a)d))
(append () ((a)d) ((a)d))      (append ((a)) (d) ((a)d))
(append ((a)d) () ((a)d))      (append () (b(a)) (b(a)))
(append (b) ((a)) (b(a)))      (append (b(a)) () (b(a)))
(append () ((a)) ((a)))        (append ((a)) () ((a)))
(append () (a) (a))            (append (a)  () (a))
(append () (b) (b))            (append (b)  () (b))
(append () (d) (d))            (append (d)  () (d))
(append () () ())
*end*

(rules-subsumed t)
(count-tree-nodes t)
(rules-evaluated t)
(prune-by-count t)
(prune-by-order t)
```

OUTPUT:

```
The linear order derived from obj-order:
(a b d nil (a) (b) (d) ((a)) ((a) d) (b (a)) (b (a) d))

UNIVERSE:
(a b d nil (a) (b) (d) ((a)) ((a) d) (b (a)) (b (a) d))

INPUT BASE TUPLES:
((null nil)     (comp nil nil nil)  (comp (a) a nil)
 (comp (b) b nil) (comp (d) d nil)    (comp ((a)) (a) nil)
 (comp ((a) d) (a) (d))              (comp (b (a)) b ((a)))
 (comp (b (a) d) b ((a) d)))
```

INPUT TARGET TUPLES:
((append nil nil nil) (append nil (a) (a))
(append nil (b) (b)) (append nil (d) (d))
(append nil ((a)) ((a))) (append nil ((a) d) ((a) d))
(append nil (b (a)) (b (a))) (append nil (b (a) d) (b (a) d))
(append (a) nil (a)) (append (b) nil (b))
(append (b) ((a)) (b (a))) (append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)) (append ((a) d) nil ((a) d))
(append (b (a)) nil (b (a))) (append (b (a)) (d) (b (a) d))
(append (b (a) d) nil (b (a) d)))

**** ITERATION 1 ****
The next tuple chosen for e is: (append nil nil nil)

Valid rules found are:
Rule N2: ((append nil nil nil) (null nil))
Cover: ((append nil nil nil))

Rule N3: ((append nil nil nil) (comp nil nil nil))
Cover: ((append nil nil nil))

Rules selected from valid rules are: (N2)

Adding to S: ((append Y4 Y4 Y4) (null Y4))

Uncovered target tuples: ((append nil (a) (a))
(append nil (b) (b))
(append nil (d) (d))
(append nil ((a)) ((a)))
(append nil ((a) d) ((a) d))
(append nil (b (a)) (b (a)))
(append nil (b (a) d) (b (a) d))
(append (a) nil (a))
(append (b) nil (b))
(append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d))
(append ((a)) nil ((a)))
(append ((a)) (d) ((a) d))
(append ((a) d) nil ((a) d))
(append (b (a)) nil (b (a)))
(append (b (a)) (d) (b (a) d))
(append (b (a) d) nil (b (a) d)))

**** ITERATION 2 ****
The next tuple chosen for e is: (append nil (a) (a))

Valid rules found are:
Rule N18: ((append nil (a) (a)) (null nil) (comp (a) a nil))
Cover: ((append nil nil nil) (append nil (a) (a))
(append nil (b) (b)) (append nil (d) (d))
(append nil ((a)) ((a))))

Rule N24: ((append nil (a) (a)) (comp nil nil nil)
(comp (a) a nil))
Cover: ((append nil nil nil) (append nil (a) (a))
(append nil (b) (b)) (append nil (d) (d))
(append nil ((a)) ((a))))

Rule N32: ((append nil (a) (a)) (comp (a) a nil)
(comp ((a)) (a) nil))
Cover: ((append nil nil nil) (append nil (a) (a)))

Rule N33: ((append nil (a) (a)) (comp (a) a nil)
(append nil nil nil))
Cover: ((append nil nil nil) (append nil (a) (a))
(append nil (b) (b)) (append nil (d) (d))
(append nil ((a)) ((a))))

Rule N34: ((append nil (a) (a)) (comp (a) a nil)
(comp ((a) d) (a) (d)))
Cover: ((append nil nil nil) (append nil (a) (a)))

Rules selected from valid rules are: (N18)

Removing from S: ((append Y4 Y4 Y4) (null Y4))

Adding to S: ((append Y4 Y5 Y5) (null Y4) (comp Y5 Y1 Y4))

Uncovered target tuples: ((append nil ((a) d) ((a) d))
(append nil (b (a)) (b (a)))
(append nil (b (a) d) (b (a) d))
(append (a) nil (a))
(append (b) nil (b))
(append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d))
(append ((a)) nil ((a)))
(append ((a)) (d) ((a) d))
(append ((a) d) nil ((a) d))
(append (b (a)) nil (b (a)))
(append (b (a)) (d) (b (a) d))
(append (b (a) d) nil (b (a) d)))

**** ITERATION 3 ****
The next tuple chosen for e is: (append nil ((a) d) ((a) d))

Valid rules found are:
Rule N73: ((append nil ((a) d) ((a) d)) (null nil)
(comp ((a) d) (a) (d)))

Cover: ((append nil nil nil) (append nil (a) (a))
(append nil (b) (b)) (append nil (d) (d))
(append nil ((a)) ((a)))
(append nil ((a) d) ((a) d))
(append nil (b (a)) (b (a)))
(append nil (b (a) d) (b (a) d)))

Rule N74: ((append nil ((a) d) ((a) d)) (null nil)
(comp (b (a) d) b ((a) d)))
Cover: ((append nil nil nil) (append nil (d) (d))
(append nil ((a)) ((a)))
(append nil ((a) d) ((a) d)))

Rule N84: ((append nil ((a) d) ((a) d)) (comp nil nil nil)
(comp ((a) d) (a) (d)))
Cover: ((append nil nil nil) (append nil (a) (a))
(append nil (b) (b)) (append nil (d) (d))
(append nil ((a)) ((a)))
(append nil ((a) d) ((a) d))
(append nil (b (a)) (b (a)))
(append nil (b (a) d) (b (a) d)))

Rule N85: ((append nil ((a) d) ((a) d)) (comp nil nil nil)
(comp (b (a) d) b ((a) d)))
Cover: ((append nil nil nil) (append nil (d) (d))
(append nil ((a)) ((a)))
(append nil ((a) d) ((a) d)))

Rule N94: ((append nil ((a) d) ((a) d)) (comp (a) a nil)
(comp ((a) d) (a) (d)))
Cover: ((append nil nil nil) (append nil ((a)) ((a)))
(append nil ((a) d) ((a) d)))

Rule N126: ((append nil ((a) d) ((a) d)) (append nil nil nil)
(comp ((a) d) (a) (d)))
Cover: ((append nil nil nil) (append nil (a) (a))
(append nil (b) (b)) (append nil (d) (d))
(append nil ((a)) ((a)))
(append nil ((a) d) ((a) d))
(append nil (b (a)) (b (a)))
(append nil (b (a) d) (b (a) d)))

Rule N127: ((append nil ((a) d) ((a) d)) (append nil nil nil)
(comp (b (a) d) b ((a) d)))
Cover: ((append nil nil nil) (append nil (d) (d))
(append nil ((a)) ((a)))
(append nil ((a) d) ((a) d)))

Rule N131: ((append nil ((a) d) ((a) d)) (append nil (a) (a))
(comp ((a) d) (a) (d)))
Cover: ((append nil nil nil) (append nil ((a)) ((a)))
(append nil ((a) d) ((a) d)))

Rule N132: ((append nil ((a) d) ((a) d)) (append nil (a) (a))
(comp (b (a) d) b ((a) d)))
Cover: ((append nil nil nil) (append nil (d) (d))
(append nil ((a)) ((a)))
(append nil ((a) d) ((a) d)))

Rule N135: ((append nil ((a) d) ((a) d)) (append nil (b) (b))
(comp ((a) d) (a) (d)))
Cover: ((append nil nil nil) (append nil (a) (a))
(append nil (b) (b)) (append nil (d) (d))
(append nil ((a)) ((a)))
(append nil ((a) d) ((a) d))
(append nil (b (a)) (b (a)))
(append nil (b (a) d) (b (a) d)))

Rules selected from valid rules are: (N73)

Removing from S: ((append Y4 Y5 Y5) (null Y4) (comp Y5 Y1 Y4))

Adding to S: ((append Y4 Y9 Y9) (null Y4) (comp Y9 Y5 Y7))

Uncovered target tuples: ((append (a) nil (a))
(append (b) nil (b))
(append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d))
(append ((a)) nil ((a)))
(append ((a)) (d) ((a) d))
(append ((a) d) nil ((a) d))
(append (b (a)) nil (b (a)))
(append (b (a)) (d) (b (a) d))
(append (b (a) d) nil (b (a) d)))

**** ITERATION 4 ****
The next tuple chosen for e is: (append (a) nil (a))

Valid rules found are:
Rule N149: ((append (a) nil (a)) (append nil (a) (a)))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (d) nil (d))
(append ((a)) nil ((a)))
(append ((a) d) nil ((a) d))
(append (b (a)) nil (b (a)))
(append (b (a) d) nil (b (a) d)))

Rules selected from valid rules are: (N149)

Adding to S: ((append Y5 Y4 Y5) (append Y4 Y5 Y5))

Uncovered target tuples: ((append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append ((a)) (d) ((a) d))
(append (b (a)) (d) (b (a) d)))

**** ITERATION 5 ****
The next tuple chosen for e is: (append (b) ((a)) (b (a)))
PRUNING NODE: (=) N179 (=) N182 (=) N183 (=) N193 (=) N200
(=) N201 (=) N202 (=) N204 (=) N205 (=) N206
(=) N207 (=) N208 (=) N209 (=) N219 (=) N220
(=) N221 (=) N223 (=) N224 (=) N228 (=) N229
(=) N230 (=) N231 (=) N238 (=) N243 (=) N251
(=) N252 (=) N253 (=) N254 (=) N255 (=) N256
(=) N259 (=) N260 (=) N261 (=) N263 (=) N264
(=) N265 (=) N266 (=) N267 (=) N268 (=) N274
(=) N275 (=) N276 (=) N278 (=) N279 (=) N283
(=) N284 (=) N285 (=) N286 (=) N287 (=) N288
(=) N289 (=) N290 (=) N292 (=) N293 (=) N294
(=) N295 (=) N296 (=) N297 (=) N301 (=) N302
(=) N303 (=) N305 (=) N306 (=) N310 (=) N311
(=) N312 (=) N313

Valid rules found are:
Rule N316: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(append nil (b) (b)) (comp (b (a)) b ((a))))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N336: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(append (b) nil (b)) (comp (b (a)) b ((a))))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N355: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp ((a)) (a) nil) (comp (b (a)) b ((a))))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N375: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp (b (a)) b ((a))) (append nil ((a)) ((a))))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N376: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp (b (a)) b ((a))) (append ((a)) nil ((a))))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N377: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp (b (a)) b ((a)))
(append nil (b (a)) (b (a))))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N378: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp (b (a)) b ((a)))
(append (b (a)) nil (b (a))))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N380: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp (b (a)) b ((a))) (null nil))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N381: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp (b (a)) b ((a))) (comp nil nil nil))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N384: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp (b (a)) b ((a))) (append nil nil nil))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N385: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp (b (a)) b ((a))) (append nil (a) (a)))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rule N389: ((append (b) ((a)) (b (a))) (comp (b) b nil)
(comp (b (a)) b ((a))) (append (a) nil (a)))

Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d)))

Rules selected from valid rules are: (N380)

Adding to S: ((append Y6 Y8 Y10) (comp Y6 Y2 Y4)
(comp Y10 Y2 Y8) (null Y4))

Uncovered target tuples: ((append (b (a)) (d) (b (a) d)))

**** ITERATION 6 ****
The next tuple chosen for e is: (append (b (a)) (d) (b (a) d))
PRUNING NODE: (=) N1073 (=) N1074 (=) N1075 (=) N1085
(=) N1086 (=) N1087 (=) N1088 (=) N1089
(=) N1090 (=) N1100 (=) N1101 (=) N1102
(=) N1105 (=) N1106 (=) N1107 (=) N1108
(=) N1109 (=) N1110 (=) N1114 (=) N1124
(=) N1125 (=) N1126 (=) N1129 (=) N1130
(=) N1134 (=) N1135 (=) N1136 (=) N1137
(=) N1147 (=) N1148 (=) N1149 (=) N1150
(=) N1151 (=) N1152 (=) N1156 (=) N1163
(=) N1164 (=) N1165 (=) N1175 (=) N1176
(=) N1177 (=) N1182 (=) N1183 (=) N1185
(=) N1188 (=) N1189 (=) N1194 (=) N1195
(=) N1196 (=) N1199 (=) N1200 (=) N1201
(=) N1202 (=) N1203 (=) N1204 (=) N1208
(=) N1213 (=) N1214 (=) N1215 (=) N1218
(=) N1219 (=) N1223 (=) N1224 (=) N1225
(=) N1226 (=) N1231 (=) N1232 (=) N1233
(=) N1234 (=) N1235 (=) N1236 (=) N1237
(=) N1238 (=) N1239 (=) N1240 (=) N1241
(=) N1242 (=) N1243 (=) N1244 (=) N1245
(=) N1246 (=) N1249 (=) N1250 (=) N1251
(=) N1252 (=) N1253 (=) N1254 (=) N1258
(=) N1259 (=) N1260 (=) N1261 (=) N1264
(=) N1265 (=) N1269 (=) N1270 (=) N1271
(=) N1272 (=) N1273 (=) N1274 (=) N1275
(=) N1276 (=) N1277

Valid rules found are:
Rule N1355: ((append (b (a)) (d) (b (a) d))
(comp (b (a)) b ((a))) (append ((a)) (d) ((a) d))
(comp (b (a) d) b ((a) d)))
Cover: ((append nil nil nil) (append (a) nil (a))
(append (b) nil (b)) (append (b) ((a)) (b (a)))
(append (b) ((a) d) (b (a) d))
(append (d) nil (d)) (append ((a)) nil ((a)))
(append ((a)) (d) ((a) d))
(append ((a) d) nil ((a) d))
(append (b (a)) nil (b (a)))
(append (b (a)) (d) (b (a) d))
(append (b (a) d) nil (b (a) d)))

Rules selected from valid rules are: (N1355)

Removing from S: ((append Y5 Y4 Y5) (append Y4 Y5 Y5))

Removing from S: ((append Y6 Y8 Y10) (comp Y6 Y2 Y4)
(comp Y10 Y2 Y8) (null Y4))

Adding to S: ((append Y10 Y7 Y11) (comp Y10 Y2 Y8)
(append Y8 Y7 Y9) (comp Y11 Y2 Y9))

Uncovered target tuples: nil

Rule: N73
Ground form of rule:
((append nil ((a) d) ((a) d)) (null nil)
(comp ((a) d) (a) (d)))
General rule:
((append Y4 Y9 Y9) (null Y4) (comp Y9 Y5 Y7))

Rule: N1355
Ground form of rule:
((append (b (a)) (d) (b (a) d)) (comp (b (a)) b ((a)))
(append ((a)) (d) ((a) d)) (comp (b (a) d) b ((a) d)))
General rule:
((append Y10 Y7 Y11) (comp Y10 Y2 Y8) (append Y8 Y7 Y9)
(comp Y11 Y2 Y9))

Count of nodes constructed:
(0 1)(1 6)
(0 1)(1 8)(2 32)
(0 1)(1 13)(2 82)
(0 1)(1 15)
(0 1)(1 9)(2 143)(3 745)
(0 1)(1 13)(2 205)(3 1201)

The final output from our FORGE algorithm includes only two rules for the append relation. The resulting two rules from the FORGE algorithm are listed in Table 7.1.

Table 7.1. Resulting Rules from FORGE for Append Relation.

No.	Rule
1.	append(X, Y, Y) ← null(X), comp(Y, Z, W).
2.	append(X, Y, Z) ← comp(X, W, V), append(V, Y, U), comp(Z, W, U).

We see from Table 7.1 that the second rule is a recursive rule that defines append in terms of smaller elements of X. Rule 1 is the basis rule for the recursion. The two rules constructed by FORGE are the usual rules used to define the append relation in Prolog type programs.

The output from the FOIL algorithm [33] is listed in Table 7.2. We see that the FOIL algorithm does not produce the smallest set of rules to define the append relation. FORGE used the reduction of the set S of valid rules to produce a smaller set of rules than the FOIL algorithm.

Table 7.2. Resulting Rules from FOIL for Append Relation.

No.	Rule
1.	append(X, Y, Z) ← X = Z, null(Y).
2.	append(X, Y, Z) ← Y = Z, null(X).
3.	append(X, Y, Z) ← comp(Z, Y, Y), comp(X, U, V), null(V).
4.	append(X, Y, Z) ← comp(Z, W, U), comp(X, W, V), append(V, Y, U).

The set of rules produced by FOIL is a superset of the set of rules produced by FORGE. The resulting rules from FORGE include the fact that Y in append(X, Y, Z) should be a list by use of the components relation. The rules resulting from FOIL do not include this necessary restriction. Also we see that rules 1 and 3 of the FOIL rules are not necessary when defining the append relation. Not only does that append example illustrate the success of the tree pruning heuristics, but it gives an example where FORGE uses its reduction of the set S to produce a more compact set of rules than what is produced by the FOIL algorithm.

Chapter 8

Future Directions and Summary

Future expansions of FORGE are intended to include the negation of literals within a rule. FOIL has the capability of including negated literals. Including negated literals in the rule building process greatly increases the number of possible extensions of each rule in FOIL. Using EBL techniques when considering negated literals should also reduce the number of rule extensions considered.

Another area which appears promising is to extend FORGE's capabilities to learning targets which consist of more than a single concept. For example, in our graph examples of Chapter 7, we may consider learning the target concept which consists of being linked by two forward arcs of linked by one backward arc.

The extensions of the FORGE algorithm into different areas may provide better insight to the appropriateness of the tree pruning heuristics used within the algorithm. The heuristics used in selecting the best set of valid rules also need further investigation. Techniques employed by C. Rouveirol in [37] need to be further explored to determine their appropriateness in selecting the best set of valid rules.

Another possible avenue for exploration would be the inclusion of absorption within the reduction of the set S of valid rules. Absorption may help in eliminating the need to use heuristics in selecting the best set of valid rules.

The current implementation is written in Franz Lisp [14] on an Encore Multimax 320 computer running the UMAX 4.3 (BSD) operating system. The memory requirements for the implementation are extensive and another future objective is to trim the implementations use of memory.

Since the FORGE algorithm is exponential in nature, implementing parallelism in the algorithm would be very promising. The explanation tree employed by the algorithm easily lends itself to parallel construction. Each branch of the tree may be extended independently of the other branches of the tree. Construction of the thee would be halted once a valid rule was discovered by a parallel process. Another option to using parallelism is to construct rules for each target tuple in parallel. Once a valid rule has been found for a subset of the target tuples, the processes working on these tuples may be

halted. The remaining processes would update their progress by including the covered target subset in the base tuples.

We have presented the FORGE algorithm and discussed the EBL techniques employed within the algorithm. We have compared the EBL based FORGE algorithm with the empirical based FOIL algorithm and have shown the advantages of using the EBL techniques within this limited knowledge-based framework. The significance of FORGE is twofold. First we have illustrated with this algorithm the advantages of employing explanation-based learning techniques over empirical techniques within the same framework. Second we have illustrated that explanation-based learning techniques are successful even when the amount of background knowledge is limited. Most of the background knowledge employed within the FORGE algorithm is based upon the representation used for the problem and not specific information from the problem itself. This allows the algorithm to be applied to more and various problem domains.

References

1. A. Barr, P. R. Cohen and E. A. Feigenbaum, **The Handbook of Artificial Intelligence, Vol IV**, Addison-Wesley, Reading, MA, 1989.

2. F. Bergadano, "The problem of Induction and Machine Learning", *12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, pp 1073-1078, 1991.

3. I. Bratko, **Prolog Programming for Artificial Intelligence**, Addison-Wesley, Wokinghan, England, 1986.

4. J. G. Carbonell, R. S. Michalski and T. M. Mitchell, "An overview of Machine Learning", in **Machine Learning: An Artificial Intelligence Approach**, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga Publishing Company, Palo Alto, CA, 1983.

5. E. Charniak and D. McDermott, **Introduction to Artificial Intelligence**, Addison-Wesley, Reading, MA, 1987.

6. J. Cheng, U. M. Fayyad, K. B. Irani, and Z. Qian, "Improved Decision Trees: A generalized Version of ID3", *Machine Learning*, Vol 6, 1991.

7. W. F. Clocksin and C. S. Mellish, **Programming in Prolog**, Springer-Verlag, Berlin, 1984.

8. W. W. Cohen, "Abductive Explanation-Based Learning: A Solution to the Multiple Inconsistent Explanation Problem", *Machine Learning*, Vol 7, pp 168-219, 1992.

9. T. G. Dietterich, "Learning at the Knowledge Level", *Machine Learning*, Vol 1, No 3, pp 287-316, 1986.

10. T. Ellman, "Explanation-Based Learning: A survey fo Programs and Perspectives", *ACM Computing Surveys*, Vol 21, No 2, pp 163-216, 1989.

11. C. Feng and S. Muggleton, "Towards Inductive Generalization in Higher Order Logic", *Proceedings of AAAI '92*, pp 154-162, 1992.

12. R. E. Fikes, P. E. Hart and N. J. Nilson, "Learning and Executing Generalized Robot Plans", *Artificial Intelligence*, Vol 3, No 4, pp 251-288, 1972.

13. M. W. Firebaugh, **Artificial Intelligence, A Knowledge Based Approach**, Boyd and Fraser Publishing Company, Boston, MA, 1988.

14. J. K. Foderaro, K. L. Sklower and K. Layer, **The FRANZ LISP Manual**, Regents, University of California, 1983.

15. J. Genest, S. Matwin and B. Plante, "Explanation-Based Learning with Incomplete Theories: A Three Step Approach", *Proceedings of the 7th International Conference on Machine Learning*, pp 286-293, 1990.

16. R. Johnson and O. Wichern, **Applied Multivariate Statistical Analysis**, Prentice Hall, Englewood Cliffs, NJ, 1988.

17. C. A. Knoblock, S. Minton and O. Etzioni, "Integrating Abstraction and Explanation-Based Learning in PRODIGY", *Proceedings of the 9th National Conference on Artificial Intelligence*, MIT Press, pp 541-546, 1991.

18. S. Lapointe and S. Matwin, "Sub-unification: A Tool for Efficient Induction of Recursive Programs", *Proceedings of AAAI '92*, pp 273-281, 1992.

19. J. W. LLoyd, **Foundations of Logic Programming**, Springer-Verlag, New York, 1984.

20. R. S. Michalski, "Understanding the Nature of Learning: Issues and Research Directions", in **Machine Learning: An Artificial Intelligence Approach, Vol. II**, R. S. Michalski, J. G. Carbonell, and T. M. Michell (ed.), Morgan Kaufman Inc., Los Altos, CA, pp 3-26, 1986.

21. S. Minton, J. Carbonell, C. Knoblock, D. Kuokka, O. Etzioni, and Y. Gil, "Explanation Based Learning: A Problem Solving Perspective", *Artificial Intelligence*, Vol 40, pp 63-118, 1989.

22. T. M. Mitchell, "The need for Biases in Learning Generalizations", Technical Report, CBM-TR-117, Rutgers University, New Brunswick, NJ, 1980.

23. T. M. Mitchell, "Generalization as Search", *Artificial Intelligence*, Vol 18, No 2, pp 203-226, 1982.

24. T. M. Mitchell, R. Keller, and S. Kedar-Cabelli, "Explanation-Based Generalization: A unifying view", *Machine Learning*, Vol 1, pp 47-80, 1986.

25. R. Mooney, "A General Explanation-Based Learning Mechanism and Its Application to Narrative Understanding", Technical Report, AITR88-66, University of Texas at Austin, Austin, TX, 1988.

26. S. Muggleton and W. Buntine, "Machine Invention of First Order Predicates by Inverting Resolution", *Proceedings of the Fifth International Machine Learning Workshop*, Morgan Kaufman, pp 337-352, 1988.

27. S. Muggleton and C. Feng, "Efficient Induction of Logic Programs", *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, Japan, Japanese Society for Artificial Intelligence, pp 368-381, 1990.

28. G. Pagallo and O. Haussler, "Boolean Feature Discovery in Empirical Learning", *Machine Learning*, Vol 5, pp 71-99, 1990.

29. M. J. Pazzani, C. A. Brunk and G. Silverstein, "A Knowledge-Intensive Approach to Relational Concept Learning", *Proceedings of the Eighth International Workshop on Machine Learning*, pp 432-436, 1991.

30. M. J. Pazzani and D. Kibler, "The Utility of Knowledge in Inductive Learning", to appear in *Machine Learning*, 1992.

31. J. R. Quinlan, "Learning Efficient Classification Procedures and their application to Chess End Games", in **Machine Learning: An Artificial Intelligence Approach**, R. S. Michalski, J. G. Carbonel and T. M. Mitchell (ed.), Tioga Publishing Company, Palo Alto, CA, pp 463-487, 1983.

32. J. R. Quinlan, "Induction of Decision Trees", *Machine Learning*, Vol 1, pp 81-106, 1986.

33. J. R. Quinlan, "Learning Logical Definitions from Relations", *Machine Learning*, Vol 5, pp 239-266, 1990.

34. B. L. Richards and R. J. Mooney, "Learning Relations by Pathfinding", *Proceedings of AAAI '92*, pp 50-55, 1992.

35. J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *Journal of the ACM*, Vol 12, No 1, pp 23-41, 1965.

36. C. Rouveirol and J. F. Puget, "Beyond Inversion of Resolution", *Proceedings of the Fifth International Conference on Machine Learning*, Morgan Kaufman, pp 122-130, 1990.

37. C. Rouveirol, "Semantic Model for Induction of First Order Theories", *Proceedings of the Ninth Machine Learning Workshop*, pp 685-690, 1992.

38. G. A. F. Seber, **Multivariate Observations**, John Wiley and Sons, Inc., New York, 1984.

39. S. A. Vere, "Induction of Concepts in the Predicate Calculus", *The Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, USSR, pp 281-287, 1975.

40. S. A. Vere, "Inductive Learning of Relational Productions", *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, Morgan Kaufman, Los Altos, CA, pp 349-355, 1977.

41. L. Watanabe and L. Rendell, "Learning Structural Decision Trees from Examples", *12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, J. Mylopoulos and R. Reiter (ed.), pp 770-776, 1991.

42. R. Wilensky, **Common LISPcraft**, W. W. Norton and Company, New York, 1986.

43. P. H. Winston, **Artificial Intelligence**, Addison-Wesley, Reading, MA, 1984.

44. Y. Wu, S. Wang, and Q. Zhu, "An Integrated Framework of Inducing Rules from Examples", *Proceedings of the Seventh International Conference on Machine Learning*, pp 357-365, 1990.

Appendix

Pseudocode for FORGE Implementation

Function: ESTABLISH-ROOT (rootvalue)

Input: Rootvalue is the target tuple that has been choosen as 'e'.

Output: The node produced for the root is returned.

Note: This function is called by BUILD-S. The tuples to be used in the rule building process (set G and G0 for the root node) are set in this function. This set is limited if the *prune-by-order* option is non-nil. The root node is given a value and its ground rule is set to the value of the tuple e. Its tuple cover is set to (1 0) which implies it covers itself only (one positive and no negative tuples).
A minus B is interpreted as the set difference between set A and set B.

Method:

```
1)  *root* ← GENERATE-LABEL
2)  SET-VALUE (*root*, rootvalue)
3)  currentfacts ← *basetuples*
4)  If *S* ≠ nil then
5)      If *prune-by-order* then
6)          currentfacts ← append (currentfacts,
7)              SIMPLER (GET-TCOVER('*S*), TUPLE-CLOSE rootvalue))
8)      else
9)          currentfacts ← append (currentfacts, GET-TCOVER('*S*))
10) SET-G (*root*, GENERATE-G (tail (rootvalue), currentfacts))
11) SET-G0 (*root*, currentfacts minus GET-G (*root*))
12) SET-GR-RULE (*root* (list rootvalue))
13) If *prune-by-count* then
14)     SET-TCOVER (*root* '(1, 0))
15) If *count-tree-nodes* then
16)     Print (*root*, '(0 1)) to *count-tree-nodes*
17)     *count-tree-nodes ← list (*count-tree-nodes*, 0)
18) Return (*root*)
```

Function: SIMPLER (cov-tar-tuples curr-target)

Input: cov-tar-tuples is a list of the tuples in cover(*S*). curr-target is the current target tuple.

Output: A list of tuples from cov-tar-tuples that are simpler than curr-target. A tuple is simpler than curr-target if at least one of its arguments is simpler than the corresponding argument of curr-target.

Note: This function is called by ESTABLISH-ROOT only if the *prune-by-count* option is non-nil. This function restricts the defined target tuples which will be included in the set of currect facts to begin building the explanation tree for curr-target.

Method:

```
1)  Remove (nil,
1)      For each tuple2 in cov-tar-tuples do
2)          If (apply 'or
3)              For each argtup1 and argtup2 in curr-target and tuple2 do
3)                  If member (argtup1 (tail (member (argtup2 *universe*)))) then
4)                      Return ('t)          ;simpler argument
5)                  else
6)                      Return (nil)
7)          then Return (tuple2)
8)          else Return (nil)
```

Function: BUILD-FAMILY (parent)

Input: parent is the tree node for which children are to be created.

Output: The list of children created for parent is returned. If parent has no children,then nil is returned.

Note: This function is called by EXTEND-TREE to produce all the children nodes for one parent. It produces one node for each element of the set G of parent. It also adds the value of the node, the ground rule associated with the node, and the correct G and G0 sets to the property list of each child node. The previous cover is set if *prune-by-count* option is non-nil and the mappings are also set. A ⊕ B is interpreted as (append A (list B)).

Method:

```
1)  G ← GET-G (parent)
2)  If G = nil then Return (nil)
3)  G0 ← GET-G0 (parent)
4)  childlist ← nil
5)  Do while G ≠ nil
6)      childvalue ← head (G)
7)      childlist ← childlist ⊕ (newnode ← GENERATE-LABEL)
8)      SET-VALUE (newnode, childvalue)
9)      SET-G (newnode, append (tail (G),
                GENERATE-G (REMOVE-DUPLICATES (tail (childvalue)), G0)))
10)     SET-G0 (newnode, G0 minus GET-G (newnode))
11)     SET-PARENT (newnode, parent)
12)     If *prune-by-count* then
13)       SET-PCOVER (newnode, GET-COVER (parent))
14)     G ← tail (G)
15) Return (childlist)
```

Function: EXTEND-TREE (parent-list)

Input: parent-list is a list of tree nodes for which children are to be created.

Output: A list of children created by the function, which is the next level of the explanation tree, is returned. If no children are created, then nil is returned.

Note: This function is used to extend the tree by one level. It is called by FIND-VALID-RULES. It calls BUILD-FAMILY to actually create the new nodes. After a family has been created the parent node is released.

Method:

1) If parent-list = nil then Return (nil)
2) parents ← parent-list
3) childlist ← nil
4) Do while parents ≠ nil
5) childlist ← append (childlist, BUILD-FAMILY (head (parents)))
6) REDUCE-PROPLST (head (parents))
7) parents ← tail (parents)
8) Return (childlist)

Algorithm GENERATE-G (arguments, tuples)

Input: Auguments is a list of distinct constants appearing the latest node added to the tree. Tuples is a list of tuples that have not yet been used in the construction of this branch of the tree, i.e., the set G0 from the parent node of the latest node added to the tree.

Output: The subset of tuples in which at least one of arguments appears in the tuple, i.e. the set of elements from tuples that have at least one link to arguments. The subset will be in order of its links to arguments, that is all the tuples linked to the first argument will be first followed by all the tuples linked to the second argument and so forth.

Note: This function is called by ESTABLISH-ROOT. It is used to create the new set G for a current node. This function and all functions used by this one is order preserving.
A ⊕ B is interpreted as (append A (list B)).

Method:

```
1)  G ← nil
2)  For each argument in arguments do
3)      For each tuple in tuples do
4)          if submember (argument, list(tuple)) then
5)              G ← G ⊕ tuple
6)  Return (remove-duplicates (apply 'append G))
```

Algorithm BUILD-S (infile bug)

Input: infile is the file with the input information. This file is read by function SETUP. If bug is non-nill then the debugging print will print each function's name and parameters each time the function is called.

Output: The set of valid rules that cover the list of *targettuples* is returned. *S* is a global variable that is the current set of valid rules, that is, it is a list of rule pointers.

Note: This function is the main driver of the algorithm. It establishes a root and builds the tree one level at a time until a valid rule is found or until the tree ends. If any valid rules are found, it then picks that best of them and adds them to *S* and removes any rule from *S* which may be subsumed these rules. If no rules are valid, it adds the target tuple as a fact rule and starts the loop over until all of *targettuples* are covered.

Method:

```
1)   *pdebug* ← bug
2)   SETUP (infile)
3)   uncovered-target-tuples ← *targettuples*
4)   Do while uncovered-target-tuples ≠ nil
5)     print('Next e is', rootvalue ← head (uncovered-target-tuples))
6)     newrules ← FIND-VALID-RULES (list (ESTABLISH-ROOT (rootvalue)),
                                          uncovered-target-tuples)
7)     If newrules ≠ nil then
9)       print each rule in newrules and its cover
10)    Case
11)      newrules = nil:
12)        ADD-FACT-TO-*S* (*root*, list (rootvalue))
13)      (*S* = nil) and (length (newrules) = 1):
14)        ADD-TO-*S* (head (newrules), GET-COVER (head (newrules)))
15)      ELSE:
16)        If length(newrules) > 1 then
17)          newrules ← FIND-SIMPLEST-RULES (newrules, uncovered-target-tuples)
18)        print('Valid rules selected are', newrules)
19)        rules-to-remove ← nil
20)        For each rule in newrules do
21)          For each S-rule in *S* do
22)            if S-rule ∉ rules-to-remove then
23)              Case
24)                length(S-rule) = 1:
25)                  if tcover(S-rule) ∈ tcover(rule) then
26)                    rules-to-rmeove ← rules-to-remove ⊕ S-rule
27)                Else:
28)                  if tcover(S-rule) ⊂ tcover(rule) or
                       RULE1-SUBSUMES-RULE2( rule, S-rule) then
```

```
29)                         rules-to-remove ← rules-to-remove ⊕ S-rule
30)                     End Case
31)                 End For
32)             End For
33)     Case
34)     If rules-to-remove ≠ nil then
35)       For each rule in rules-to-remove do
36)         REMOVE-FROM-*S* (rule)
37)     For each rule in newrules do
38)       ADD-TO-*S* (rule)
39)     uncovered-target-tuples ← uncovered-target-tuples – tcover(*S*)
40)     print("Remaining uncovered tuples", uncovered-target-tuples)
41)   End DO
```

Function: FIND-SIMPLEST-RULES (newrules, uncov-tar-tup)

Input: newrules is a list of pointers to all valid rules produced at the current level of the explanation tree. uncov-tar-tup is a list of target tuples that are not in cover(S).

Output: A list of simplest rules from newrules.

Note: This function is called by BUILD-S.

Method:

```
1)  rules-to-remove ← nil
2)  rest-newrules ← nil
3)  For each rule1 in newrules do
4)    If rule1 ∉ rules-to-remove then
5)      rest-newrules ← tail (member (rule1, newrules))
6)      If rest-newrules ≠ nil then
7)        rules-to-remove ← For each rule2 in rest-newrules do
8)          if rule2 ∉ rules-to-remove then
9)            RULE-2-REMOVE (rule1, rule2, uncov-tar-tup)
10) Return (newrules – rules-to-remove)
```

Function: RULE-2-REMOVE (rule1, rule2, uncov-tup)

Input: rule1 and rule2 are two valid rules from the same level of the same explanation tree. uncov-tup is a list of target tuples not in cover(S).

Output: If the cover of the two rules is disjoint then nil is returned. If the cover of one rule is contained within the cover of the other rule, then the rule with the smaller cover is returned. If the rules have the exact same cover, then the least simple rule is returned. Simpler is first determined by the least recursive calls in a rule, and next by the least number of arguments. If the two rules are the same on all these characteristics, then rule2 is returned as the rule to remove.

Note: This function is called by FIND-SIMPLEST-RULES.

Method:

```
1)  cov1 ← tcover (rule1) ∩ uncov-tup
2)  cov2 ← tcover (rule2) ∩ uncov-tup
3)  r ← nil
4)  If cov1 ⊆ cov2 then
5)    If cov2 ⊆ cov1 then
6)      If (r ← MORE-RECURSIVE (rule1, rule2)) = nil then
7)        If (r ← MORE-ARGS (rule1, rule2)) = nil then
8)          Return (rule2)
9)        Else
10)         Return (r)
11)     Else
12)       Return (r)
13)   Else
14)     Return (rule1)
15) Else
16)   If cov2 ⊆ cov1 then
17      Return (rule2)
18    Else
19      Return (nil)
```

Function: FIND-VALID-RULES (current-rule-set)

Input: current-rule-set is a list of rule pointers for the current set of rules. If the tree is being built one level at a time then the current-rule-set is the list of leaves of the current level of the tree.

Output: The list of valid rule pointers. This list may be nil if no valid rules are found, or it may contain only one rule ptr or it may be a list of two or more rule pointers.

Note: A rule pointer is a symbol that is a leaf node of a tree. It has a ground rule associated with it. A rule value is a list whose head is the rule pointer and whose tail is the (generalized) rule. This function is used to extend the tree one level and then evaluate the rules. If any are valid, they are returned. If none are valid then the function is called recursively. This function is called by BUILD-S.

Method:

```
1)  allnewrule-ptrs ← current-rule-set
2)  Do while allnewrule-ptrs ≠ nil
3)    allnewrule-vals ← nil
4)    newrules ← nil
5)    valid ← nil
6)    allnewrule-ptrs ← EXTEND-TREE (allnewrule-ptrs)
7)    If *count-tree-nodes* ≠ nil then
8)      CT-TREE-NODES (length (allnewrule-ptrs))
9)    Case
10)     allnewrule-ptrs = nil:
11)       Return (nil)
12)     Else:
13)       newrules ← DISTINCT-RULES (allnewrule-ptrs)
14)       If *rules-evaluated* or *first-valid-rule* then
15)         valid ← EVAL-RULES (newrules uncov-tar-tup)
16)       Else
17)         valid ← (For each rule in newrules do
                          SET-COVER-RETURN-IF-VALID (rule))
18)       If valid ≠ nil then
19)         MARKVALIDPATH (valid)
20)         REMOVE-NONVALID (allnewrule-ptrs)
21)         RETURN (valid)
22)       Else
23)         SET-CURR-MAP-COVER (allnewrule-ptrs – newrules)
24)         If *prune-by-count* ≠ nil then
25)           allnewrle-ptrs ← PRUNE-TREE (allnewrule-ptrs)
26)   End Case
27) End DO
```

Function: DISTINCT-RULES (allnewrule-ptrs)

Input: allnewrule-ptrs is a list of pointers to the current rules (or leaf nodes in the explanation tree).

Output: A list of rule pointers that point to distinct rules. Rules pointed to by allnewrule-ptrs may be equivalent. This function removes the pointers to equivalent rules so that the remaining set of rule pointers is the largest set such that no two rules are equivalent.

Note: This function is called by FIND-VALID-RULES.

Method:

1) dist-rules ← nil
2) all-ptrs ← allnewrule-ptrs
3) first-rule ← head (allnewrule-ptrs)
4) rest-rules ← tail (allnewrule-ptrs)
5) Do while all-ptrs ≠ nil
6) all-ptrs ← REMOVE-SAME-RULES (first-rule, rest-rules)
7) dist-rules ← dist-rules ⊕ first-rule
8) first-rule ← head (all-ptrs)
9) rest-rule ← tail (all-ptrs)
10) End Do
11) Return (dist-rules)

Function: REMOVE-SAME-RULES (rule-ptr, rest-rules)

Input: rule-ptr is a pointer to a rule which is to be kept. rest-rules is a list of rule pointers which may or may not point to rules that are equivalent to the rule of rule-ptr.

Output: A list of rule pointers that point to rules that are distinct from the rule of rule-ptr.

Note: This function is called by DISTINCT-RULES and is used to remove all rules equivalent to the rule to which rule-ptr points from the list rest-rules.

Method:

```
1)   rule ← GET-GENERAL-RULE (rule-ptr)
2)   map ← nil
3)   For each r-ptr in rest-rules do
4)     if (map ← (RULE1-SUBSUMES-RULE2 (rule, GET-GR-RULE (r-ptr))) = nil
       and GET-GR-RULE (r-ptr) ⊆ SUBPAIRS (map, rule) then
5)       SET-CURRMAP (r-ptr (list (rule-ptr, map)))
6)       If *prune-by-count* ≠ nil then
7)         (setproperty r-ptr 'cover rule-ptr)
8)       If *rules-subsumed* then
9)         print ('Rules equivalent', r-ptr, rule-ptr)
10)      Return (nil)
11)    Else
12)      Return (r-ptr)
```

Function: SET-COVER-RETURN-IF-VALID (rule-ptr)

Input: rule-ptr is a pointer to a distinct rule from the set of rule pointers of the current level of the tree.

Output: Nil if rule is not valid otherwise the rule pointer is returned.

Note: This function is used to evaluate the coverage of rules. It determines if a rule is valid or not. It also adds the tuples covered to the property list of the rule pointer. It is called by FIND-VALID-RULES and EVAL.-RULES. A – B is interpreted as the set difference between set A and set B.

Method:

```
1)  rule ← GET-GENERAL-RULE (rule-ptr)
2)  rule-head ← head (rule)
3)  literal ← head (last (rule))
4)  tuples ← If head (literal) = head (head (*targettuples*)) then
5)     *targettuples*
6)  else
7)     *basetuples*
8)  parent-map ← GET-MAPPINGS (GET-PARENT (rule-ptr))
9)  curr-map ← INSTANTIATE-LITERAL-TO-TUPLES (literal, parent-map, tuples)
10) mappings ← PRODUCEMAP (parent-map, curr-map)
11) cover-rule ← nil
12) cover-r ← nil
13) cover-rule ← remove-duplicates (For each map in mappings do
14)       Return ( SUBINLIST (map, rule-head)))
15) cover-r ← apply 'append (For each cover in cover-rule do
16)       If CONTAINS-VAR (cover) then
17)            Return (ALL-INSTANTIATIONS (cover)))
18)       Else
19)            Return (cover)
20) If *prune-by-count* then
21)       SET-COVER (rule-ptr, cover-r)
22) SET-CURRMAP (rule-ptr, curr-map)
23) If (cover-r – *targettuples*) ≠ nil then
24)       Return (nil)
25) Else
26)       Return (rule-ptr)
```

Function: ALL-INSTANTIATIONS (target-head)

Input: target-head is the head of an instantiated rule. It contains at least one variable. This happens when evaluating rules that are not full.

Output: A list of all tuples (positive and negative) that will instantiate to target-head.

Note: This function is used in evaluating the cover of a rule in SET-COVER-RETURN-IF-VALID.

Method:

```
1)  vars-list ← REMOVE-DUPLICATES (tail (target-head)) minus *universe*
2)  If length (vars-list) = 1 then
3)    pairlist-list ← nil
4)    For each const in *universe* do
5)      pairlist-list ← pairlist-list ⊕ list (list (head (vars-list), const)) else
6)    const-substitutions ← ALL-K-TUPLES (*universe*, length (vars-list))
7)    pairlist-list ←
8)      For each const-tuple in const-substitutions
9)        For each var in varlist and each const in const-tuple do
10)         (list var const)
11) all-tuples ← nil
12) For each pairlist in pairlist-list do
13)   all-tuples ← all-tuples ⊕ SUBINLIST (pairlist, target-head)
14) Return (all-tuples)
```

Function: RULE1-SUBSUMES-RULE2 (rule1, rule2)

Input: rule1 and rule2 contain no negation. rule1 is a general rule (it contains no constants).

Output: A list of variable-value pairs which map rule1 to rule2. If such a map does not exist, then nil is returned.

Note: This function is used in BUILD-S to determine if a new rule subsumes any existing rules in *S*.
A ⊕ B is interpreted as (append A (list B)).

Method:

```
1)  comp1 ← FIND-VARS (rule1)          {all duplicates removed}
2)  comp2 ← FIND-VARS (rule2)          {all duplicates removed}
3)  common ← comp1 ∩ comp2
4)  If common ≠ nil then               {generate new symbols that will}
5)    pairlist ← nil                   {replace common ones}
6)    For each var in common do
7)      pairlist ← pairlist ⊕ list (var, GENERATE-SYMBOL)
8)    rule1 ← subpairs (pairlist, rule1)
9)  target1 ← head (rule1)
10) target2 ← head (rule2)
11) body1 ← tail (rule1)
12) body2 ← tail (rule2)
13) headmap ← INSTANTIATE-LITERALS (target1, target2, comp2)   {does not return true}
14) If headmap = nil then Return (nil)
15) bodymap ← BODY1-SUBSUMES-BODY2 (SUBPAIRS (headmap, body1), body2, comp2)
16) If bodymap = nil then Return (nil)
17) bodymap ← headmap ∪ remove ('true, bodymap)
18) Return (bodymap)                   {return mapping}
```

Function: BODY1-SUBSUMES-BODY2 (body1, body2, comp2)

Input: body1 and body2 are rule bodies which contain no negation. body1 contains variables or components(previous substitutions) from body2. body2's components may be either variables or constants.

comp2 is a list of components found in the rule from which body2 was derived.

Output: A list of variable-value pairs which map body1 to body2, if such a map exists; otherwise nil is returned.

Note: This function is called by RULE1-SUBSUMES-RULE2. It is used to check the subsumption of one rule body against another.

Method:

```
1)  If body1 = nil Return ('true)          {all of body1 has been matched}
2)  If body2 = nil Return ('nil)
3)  map ← INSTANTIATE-BODY-TO-TUPLES (body1, body2, comp2)
4)  If map = nil then
5)     Return (nil)
6)  else
7)     Return (head (map))
```

Function: INSTANTIATE-BODY-TO-TUPLES (body, tuples-list, tuple-comp)

Input: body is the body of a rule that is to be instantiated to tuples in tuples-list.
tuples-list is a list of all base tuples and any additional target tuples that have already been covered by previously found rules.
tuple-comp is the components from tuples-list. In most cases this is the universe of constants.

Output: A list of variable/value pairs that was used in mapping tuples to rule.

Note: A ⊕ B is interpreted as (append A (list B)).
This function is used in evaluating the coverage of a rule. The function first finds 'firstmap' which is the first mapping not equal to '(t) which maps a body literal to a non-empty set of tuples in tuplelist. If all first mappings return '(t), then '(t) is returned. If all first mappings return nil, then nil is returned; otherwise mappings is set to firstmap. Mappings is then substituted into the next body literal and it is instantiated to the set of tuples in tupleslist to produce mid-map. If mid-map is nil then nil is returned. If mid-map is '(t) then mappings is preserved; otherwise mappings is replaced by mid-map appended to mappings. The substitutions, instantiations, and updating of mapings is repeated until the end of the body is reached, then mappings is returned.

Method:

```
1)  first-body-literal ← head (body)
2)  maps ← nil
3)  tuples ← tuplelist
4)  Do while tuples ≠ nil
5)     map ← INSTANTIATE-LITERALS (first-body-literal, head (tuples), tuple-comp)
6)     If map ≠ nil then
7)        maps ← maps ⊕ map
8)     tuples ← tail (tuples)
9)  End Do
10) firstmap ← maps
11) if firstmap = '((t)) then
12)    if tail (body) = nil then
13)       Return (firstmap)
14)    else
15)       body ← tail (body)
16)       goto 1)
17) else
18)    firstmap ← remove ('(t), firstmap)
19) If firstmap = nil then Return (nil)
20) If (length (body) = 1) then Return (firstmap)
21) mappings ← firstmap
22) restbody ← tail (body)
23) Do while restbody ≠ nil
24)    if restbody = nil then
```

```
25)        Return (mappings)
26)      maps ← mappings
27)      mid-map ← nil
28)      mappings ←
29)        Do while maps ≠ nil
30)          body-lit ← SUBINLIST (head (maps), head (restbody))
31)          If maps = nil then
32)            Return (mid-map)
33)          tuples ← tuplelist
34)          newmap ← nil
35)          mid-map ←
36)            Do while tuple ≠ nil
37)              If tuples = nil then
38)                Return (newmap)
39)              map ← INSTANTIATE-LITERALS (body-lit, head(tuples), tuple-comp)
40)              newmap ← append (newmap,
41)                Case
42)                  null map:  nil
43)                  map = '(t):  list (head (maps))
44)                  ELSE:  append (head (maps), map))
45)            End Do
46)        End Do
47)    End Do
```

Function: INSTANTIATE-LITERALS (literal1, literal2, comp2)

Input: literal1 and literal2 are positive literals which may contain either variables or constants. comp2 is a list of components of the rule from which literal2 was derived. If literal2 is a ground tuple then comp2 may be the universe of constants.

Output: A list of variable-value pairs used in the instantiation of literal1 to literal2, if such an instantiation exists; otherwise nil is returned.

Method:

```
1)  If length (literal1) ≠ length (literal2) then Return (nil)
2)  If head (literal1) ≠ head (literal2) then Return (nil)
3)  pairlist ← nil                {make a list of mappings }
4)  For each var in tail (literal1) and val in tail (literal2) do
5)    If var ≠ val then    {only include unlike components in map}
6)      pairlist ← pairlist ⊕ list (var, val)
7)  pairlist ← REMOVE-DUPLICATES (pairlist)
8)  If pairlist = nil then Return ('true)    {all components map}
9)  If length (pairlist) = 1 then
10)   If member (head (head (pairlist)),comp2) then  {literal1 contains previously mapped}
11)     Return (nil)                          {component that's ≠ component of literal2.}
12)   else
13)     Return (pairlist)
14) For each pair in pairlist do
15)   var ← head (head (pair))
16)   If (member (var, comp2)) then
17)     Return (nil)
18)   If (sublistmember (var, tail (member (pair, pairlist)))) then
19)     Return (nil)
20) Return (pairlist)
```

Function: SUBPAIRS (pairlist, rule)

Input: pairlist is a list of variable(old)-value(new) pairs. The new values are to be substituted for the old variables in rule.
rule contains no negation.

Output: rule with all substitions made.

Note: A ⊕ B is interpreted as (append A (list B)).
This function is used to do substitutions in rules. To do substitutions in single literals SUBIN-LIST should be used.

Method:

```
1)  pairs ← pairlist
2)  Do while pairs ≠ nil
3)    If pairs = nil then Return (rule)
4)    old-var ← head (head (pairs))
5)    new-val ← head (tail (head (pairs)))
6)    literals ← rule
7)    newrule ← nil
8)    newrule ← Do while literals ≠ nil
9)      If literals = nil Return (newrule)
10)     relation-head ← head (head (literals))
11)     arguments← tail (head (literals))
12)     newrule ← newrule ⊕ cons (relation-head,
13)       For each argument in arguments do
14)         If argument = old-var then
15)           Return (new-val)
16)         else
17)           Return (argument)
18)     literals ← tail (literals)
19)   End Do
20)   pairs ← tail (pairs)
21)   rule ← newrule
22) End Do
```

Vita

Mary Pamela Langley was born in Kinder, Louisiana on February 20, 1957. She graduated from Fenton High School in 1975. Ms. Langley received a B.S. degree from McNeese State University in 1979 and an M.S. degree in Mathematics in 1982 from the same university. From 1981 until 1988 she was employed by McNeese State University first as a research technician and then as an instructor. She began work on the Ph.D. degree in Spring of 1988 in the area of Computer Science at Louisiana State University. The Ph.D. degree was completed in December 1992 and Ms. Langley returned to McNeese State University as an Assistant Professor in the Fall of 1992.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Mary Pamela Langley

Major Field: Computer Science

Title of Dissertation: Learning Horn-Clauses as Classification Rules for Relations

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

10/23/92