A Critical Survey of Rule Learning Programs

A CRITICAL SURVEY OF RULE LEARNING PROGRAMS

Alan Bundy and Bernard Silver

Department of Artificial Intelligence
University of Edinburgh
Scotland

Abstract

We survey the rule learning programs of [Brazdil 81, Langley 81, Mitchell et al 81, Shapiro 81]. Each of these programs has two main parts: a critic for identifying faulty rules and a modifier for correcting them. To aid comparison we describe the techniques of the various authors using a uniform notation. We find several similarities in the techniques used by the various authors and uncover the relations between them. In particular, the concept learning technique of Young et al, [Young et al 77] is shown to subsume most of the rule modifying techniques. We also uncover some funnies in some of the research.

Keywords

Learning programs, concept learning, production systems, PROLOG, generalization, focussing.

Acknowledgements

We could not have conducted this survey without some clarificational conversations with the surveyees, namely: Ranan Banerji, Pavel Brazdil, Pat Langley, Gordon Plotkin, Ehud Shapiro and Richard Young. We thank them for their time and patience. We also got valuable feedback from some informal seminars given in the Edinburgh AI department.*

1. Introduction

This paper is a critical survey of the following work in the area of AI learning programs:

- The ELM program, [Brazdil 78, Brazdil 81], which transforms a specification into a program in the domains of: simple arithmetic, algebra and letter series completion.

- The AMBER program, [Langley 81], which acquires the ability to generate simple English utterances.

- The LEX program, [Mitchell et al 81], which acquires heuristics in the domain of symbolic integration.

- The Model Inference System of Shapiro, [Shapiro 81], which synthesizes programs from examples in the domains of arithmetic, list processing, etc.

- The extension by Young, Plotkin and Linz, [Young et al 77], of Winston's concept learning program, [Winston 75], which learns the definitions of simple structures, e.g. an arch, from examples and near misses.

This survey arose from the authors' attempts to understand the state of the art in learning before proceeding to build a self improving algebraic manipulation program, [Bundy and Sterling 81]. It seemed to us that the above listed researchers had provided, sometimes complementary, sometimes alternative techniques for solving isomorphic problems, but that this was obscured by their use of different formalisms and terminology. Having invested some effort in understanding the relationships between the various techniques, we thought we would share this understanding with a wider audience. Hence the present paper.

In order to clarify the similarities and differences between the techniques we have described them with a uniform formalism and terminology. To keep the comparisons simple we have suppressed some of the details of the techniques, but we hope we have retained their spirit. We have also suppressed all domain specific aspects of the techniques, except for the use of domain specific rules in our worked examples, and even here we have deliberately applied one person's technique to another's rules.

2. The Learning Task

The task tackled by all the researchers listed above, except Young et al, is to modify a set of rules of the form hypothesis implies conclusion, i.e.

H_1 & ... & H_n -> C

where each of the H_is is a condition of the hypothesis. In the case of Brazdil and Shapiro these are PROLOG clauses, which are run in backwards chaining mode by the PROLOG interpreter. In the case of Langley and Mitchell et al they are production rules, which are run in forwards chaining mode. Some example rules are given in figure 2-1.

```
A rule* , subs1, for addition in Peano
arithmetic, [Brazdil 81]
  X5=X6+X4 & X4+X2=X3 -> (X5+X6)+X2=X3

  A rule for language generation,
  [Langley 81].

  describe(X) & object(X,Y)
  & ~definite(X) & singular(X)
                  -> prefix(X,a)
which can be paraphrased as:
"If you want to describe X and X is the object
of Y and X is not definite and X is singular
then prefix X with 'a'."
```

Figure 2-1: Some Example Rules

For our purposes it is necessary that the rules have a truth value. It will be convenient to consider the rules as being formulae of Predicate Calculus, with a truth value assigned by a standard model in the usual Tarskian manner.

The rules are modified because there is something wrong with them. These faults can be of two types:

- Factual faults: A rule is false, i.e. the rules constitute a program which calculates incorrect answers.

- Control faults: The rules are true, but have undesirable control behaviour when run as a program, e.g. they do not terminate.

The faults in Langley and Shapiro's rules were factual and those in Mitchell et al's rules were control faults. Brazdil considered faults of both type.

The programs listed above all used the following main control loop.

Until the rules are satisfactory:
1. Identify a fault with a rule;
2. Modify the rule to remove the fault.

Following Mitchell et al, we will call the subprogram responsible for identifying faults the critic. We will call the subprogram responsible for modifying the rules the modifier. In the next section we consider the criticism techniques used by each of the above researchers and in the following section we consider the modification techniques.

3. Identifying Faults

All the programs we are surveying identify faults by running the existing rules on a problem and then analysing the resulting program trace. The analysis must identify where the rules behaved correctly, called positive training instances by Mitchell et al, and where they behaved incorrectly, called negative training instances. Both sorts of information can be used: the positive instances to generalise the rules and the negative instances to correct them. Negative instances can be of two types.

- Errors of commission: A rule fired incorrectly, because it was insufficiently constrained.

- Errors of omission: A rule failed to fire, either because it was incorrectly constrained, or the required rule simply does not exist.

The modifier requires three pieces of information on each instance.

- The type of instance: positive, negative-commission or negative-omission.

- The rule.

- The context, consisting of the variable bindings when the rule was fired.

Following Brazdil, we will adopt the convention that the variable bindings of positive instances are called the selection context and the variables bindings of negative instances are called the rejection context.

In the following section we describe some criticism techniques for identifying control faults and factual faults.

3.1. Using Ideal Traces to Find Faults

The only technique used for finding control faults is comparing the program trace with an ideal trace. (The ideal trace is also used to find factual faults) The first point at which the traces differ is located and this enables the faulty rules to be identified.

Running the rules causes a search tree to be grown (see figure 3-1). The program trace is a path through this tree. If this trace differs from the ideal trace it is because, at some point, the rule that fired in the program trace, R_p, differs from the rules that fired in the ideal trace, R_i. R_p exhibits an error of commission and R_i exhibits an error of omission. If both rules exist then correcting one error automatically corrects the other. Since we will not be concerned with the creation of new rules, we are free to concentrate on errors of one type. We will restrict our attention to errors of commission.

Figure 3-1: Search Tree for Programs Rules

The technique can be summarised as follows:

(a)Grow the program trace by running the rules on a problem.

(b)Compare with the ideal trace and find the first place at which they differ.

(c)The rules which fired before this point, together with their associated selection contexts, are positive training instances.

(d)The program rule which fired at this point, together with its associated rejection context, is a commission error.

For instance, suppose the rule

subs: X1=X4 & X4+X2=X3 -> X1+X2=X3

fires in the context (3/X1, 2/X2, Ans/X3) but that the rule

subs: X2=X4 & X1+X4=X3 -> X1+X2=X3

is fired in the ideal trace, then subs in the context (3/X1, 2/X2, Ans/X3), is an error of commission.

3.2. Constructing the Ideal Trace

In Brazdil 's program the user must provide his own ideal trace, but Mitchell et al's program constructs its own ideal trace by pruning the program trace. The basic idea is to find a desirable branch of the program's search tree, and prune away all other branches. In the simplest case the desirable branch will be any branch leading to a solution. Mitchell et al go further and try to find a least cost solution.

The program which builds the program trace is called the problem solver. The rules are only partially specified (see section 4.3). A numerical score is assigned to how well they apply in a situation and this score is used as the evaluation function in a heuristic search. A resource limit is given to the problem solver,which puts an upper bound on the amount of c.p.u. time and memory it may use in attempting to solve a problem. These limitations may prevent the problem solver finding the least cost solution and so lead to an erroneous ideal trace. To mitigate this a further expansion is made of negative training instances before they are finally sent to the rule modifier.

3.3. Using Contradictions to Find Factual Faults

In this section we consider Shapiro's technique for locating factual faults. This technique is called contradiction backtracing.

Suppose that the current rule set implies P, but that P is known to be false. The falsity of P may be given by the program user or calculated from the standard model. Clearly, at least one of the current rules is factually faulty, but we may not be able to tell which one from the model. If the faulty rule contains free variables and the model has an infinite domain then an infinite series of instances must be considered. Contradiction backtracing localises the search to the process of forming the program trace.

The technique can be summarised as follows:

(a)Add ~P as a new rule. (The rules are now inconsistent.)

(b)Derive ->, the empty clause, from the rules by resolution. (The derivation is a program trace, but unlike previous techniques we will not need an ideal trace.)

(c)Set -> to be the current clause of the derivation and {} to be the accumulated substitution.

(d)Until the current clause is a rule, do the following:

(i)The current clause was derived by resolving clauses, C and D, with unifier ϕ. The proposition K, from C, and negated proposition L, from D, were resolved away, where $K\phi \equiv \sim L\phi$.* Apply the accumulated substitution to $K\phi$ to form Q.

(ii)If Q contains any free variables then instantiate it to a variable free proposition, Q', in any way, using the substitution θ.

---

*This work was supported by SRC grant GR/B/29252 and an SRC studentship to Bernard Silver.

*We adopt the, PROLOG convention that identifiers beginning with a capital letter denote variables, and those beginning with a lower case letter denote constants.

*$\equiv$ means 'is syntactically identical to'.

(iii)Form a new accumulated substitution by combining it with ϕ and Θ.

(iv)If Q' is true then let D be the current clause.

(v)Otherwise Q' is false. Let C be the current clause.

(e)The current clause is a faulty rule, and applying the accumulated substitution to it gives a false instance.

The decision as to whether each Q' is true or false can either be supplied by the program user or calculated from the standard model. Note that the only calls on the model are to decide the truth value of formulae without free variables (or quantifiers). Note also that the instantiation of Q to Q' will not be necessary if Q is variable free. Different choices of Θ may lead to different faulty rules, and may all be tried.

For instance, suppose the current rule set were:

describe(Y) & object(X,Y) -> describe(X)

describe(X) & object(X,Y) -> prefix(X,a)

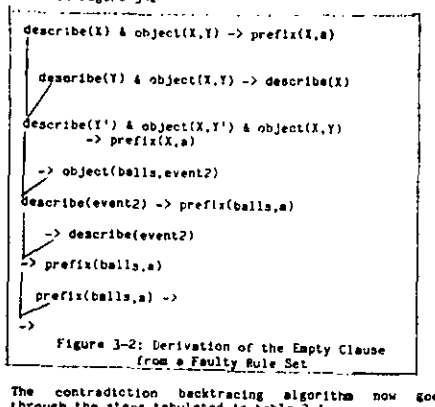
-> object(balls,event2)

-> describe(event2)

but that prefix(balls,a) were known to be false. Adding the new rule

prefix(balls,a) ->

we can derive the empty clause with the derivation given in figure 3-2



The contradiction backtracing algorithm now goes through the steps tabulated in table 3-1.

Current Clause	Q'	Truth Value
->	prefix(balls,a)	false
-> prefix(balls,a)	describe(event2)	true
describe(event2) -> prefix(balls,a)	object(balls,event2)	true
describe(Y') & object(X,Y') & object(X,Y) -> prefix(X,a)	describe(balls)	true
describe(X) & object(X,Y) -> prefix(X,a)		

Table 3-1: Backtracing Through a Contradiction

The rule

describe(X) & object(X,Y) -> prefix(X,a)

has now been identified as faulty, with substitution {balls/X, event2/Y} giving a false instance.

4. Modifying the Rules

Once a fault has been located, the faulty rule can be modified. The following modification techniques were used.

- Ordering the rules, e.g. specifying that
  H -> C should always be fired
  in preference to H' -> C'
  This technique is strictly only appropriate for control faults and was used by Brazdil. However, Langley also used it to suppress factual errors.

- Adding extra conditions to a rule's hypothesis, e.g. transforming
  H -> C to H & H' -> C
  This technique is appropriate to both factual and control faults, and was used by Brazdil, Langley and Shapiro.

- Instantiating a rule, e.g. transforming
  H(X,Y) -> C(X,Y) to
  H(X,X) -> C(X,X)
  This technique is appropriate for both factual and control faults and was used by Brazdil and Shapiro.

- Updating a rule's hypothesis, e.g. transforming
  H -> C to H' -> C
  where H' is derived from H by concept learning. This technique is appropriate to both factual and control faults, and was used by Mitchell et al.

4.1. Ordering the Rules

All rule based systems need a control strategy to decide conflicts between two or more applicable rules. If a system uses a priority ordering on the rules then control faults can often be corrected by re-ordering the rules. In this section we explain the ordering technique used by Brazdil.

Brazdil's system started with an unordered set of rules, and imposed the partial order required to keep the program trace in line with the ideal trace. His critic and modifier worked as co-routines, discovering conflicts and resolving them by imposing an order. The technique can be summarised as follows:

(a)Suppose that rules, $P_1,\ldots,P_n$, are applicable and that rule P_i is fired in the ideal trace.

(b)If $P_j > P_i$ for $i \neq j \in \{1,\ldots,n\}$ then create a new rule P'_i from P_i, by techniques to be described in subsequent sections, and impose the order $P'_i > P_j$ for all j, such that $i \neq j \in \{1,\ldots,n\}$.

(c)Otherwise impose the order $P_i > P_j$ for all j such that $i \neq j \in \{1,\ldots,n\}$.

where P > Q means that the system will fire P before Q.

For instance, suppose the rules:

subs: X1=X4 & X4=X2=X3 -> X1=X2=X3
subx: X2=X4 & X1=X4=X3 -> X1=X2=X3
eq: -> X1=X1

are all applicable, but that the ideal trace records that subx should fire, then the orders

subx > subs and subx > eq

will be imposed.

If at some later stage the same rules are in conflict, but the ideal trace records that subs should fire, then we cannot impose the order subs > subx because this would contradict the existing order subx > subs. In this case a new rule, subs1, is built from subs and the orders

subs1 > subx and subs1 > eq

are imposed. Since > is transitive these new orders also imply that subs1 > subs. The techniques for making subs1 are described in the next two sections.

Langley uses rule re-ordering to deal with factual faults. Faulty rules have their priority reduced so that they are less likely to fire in future. Consequently, the same fault may be redetected several times before the rule's priority drops so low that it is never selected. The basis for this strange technique is a rather dubious psychological argument.

4.2. Adding Extra Conditions to a Rule's Hypothesis

In this section we will consider how a rule can be modified by adding an extra condition to its hypothesis.

Suppose a rule, H -> C, has given a commission error, but that this rule has been applied correctly in the past. The variable bindings of the correct application will give us a selection context and the variable bindings of the incorrect application will give us a rejection context. The idea of this technique is to find some difference between the selection and rejection contexts and use this difference as the new condition. The technique is realised in what, following Langley, we will call the discrimination algorithm.

(a)Apply the selection and rejection context substitutions to a fixed set of literals,* called the description space.

(b)Find a literal, H', which is true in the in the selection context and false in the rejection context. H' is called a discriminating literal.

(c)Form the new rule H & H' -> C.

The new rule is only applicable to the selection context.

For instance, suppose the rule

describe(X) & object(X,Y)
-> prefix(X,a)

has been correctly applied to the word 'ball' and incorrectly applied to the word 'balls'. We have:

Selection Context: {ball/X, event1/Y}

Rejection Context: {balls/X, event2/Y}

To find the difference, H', between these contexts we apply them as substitutions to the literals in the description space:

singular(X), ¬singular(X),
definite(X), ¬definite(X)

The only discriminating literal is singular(X). Adding this to the rule as a new condition yields:

describe(X) & object(X,Y)
& singular(X) -> prefix(X,a)

4.2.1. Far Misses

In this particular combination of selection context, rejection context and description space, there is only one discriminating literal. Following Winston we call such a situation a near miss. If there is more than one discriminating literal then we will call the situation a far miss. A far miss would arise if we added to the description space the literal, past(Y), meaning event Y happened in the past. If past(event1) was true but past(event2) was false then past(Y) would also be a discriminating literal for the above contexts. Clearly the description space is of pivotal importance in determining whether a discriminating literal is found and what sort of new rules are formed. In all the programs considered here the description space is user supplied, and it is difficult to see how it could be otherwise.

Langley dealt with far miss situations by creating a new rule for each discriminating literal, e.g.

describe(X) & object(X,Y)
& singular(X) -> prefix(X,a)

describe(X) & object(X,Y)
& past(Y) -> prefix(X,a)

*A literal is either a proposition, e.g. P(X), or a negated proposition, e.g. ¬P(X).

Any useless creations (like the past rule) would eventually be criticised as faulty and fall low in the priority ordering.

Brazdil dealt with far miss situations by including all the discriminating literals in a disjunction, e.g.

describe(X) & object(X,Y)
& [singular(X) ∨ past(Y)]
-> prefix(X,a)

He used a modified version of the discrimination algorithm which tried to prune such disjunctions before adding new conditions. For instance, if the following contexts arose:

Selection Context: {ball/X, event1/Y}

Rejection Context: {balls/X, event3/Y}

where past(event3) were true then Brazdil's discrimination algorithm would drop past(Y) from the disjunction to form the rule:

describe(X) & object(X,Y)
& singular(X) -> prefix(X,a)

4.2.2. Instantiating a Rule

An alternative to adding an extra condition to a rule is to instantiate it. This is really a special case of adding an extra condition, but can lead to more efficient rules since the extra condition is handled by the pattern matcher. For instance, suppose we are modifying the rule:

subs: X1=X4 & X4=X2=X3
-> X1=X2=X3

in the contexts

Selection Context: {3=1/X1, 1/X2}

Rejection Context: {3/X1, 2/X2}

and the description space contains X1=X5=X6.* X1=X5=X6 is a discriminating literal, so we could add it as an extra condition. Alternatively, we could instantiate the rule with the substitution, {X5=X6 / X1} to form

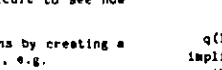
subs1: X5=X6=X4 & X4=X2=X3 -> (X5=X6)=X2=X3

Instantiation with the substitution {t/X} is always an alternative when the discriminating literal is X=t, for some variable X and term t.

4.3. Updating the Hypothesis of a Rule

In this section we consider how a rule can be modified by updating its hypothesis using concept learning techniques, like those used in [Winston 75] for learning the concept of an arch from examples and near misses. This technique can be regarded as a natural extension of the one described in the last section. This relationship is most clearly seen by considering the technique of Young et al, because it generalises the Winston and Brazdil/Langley techniques, and is similar to, but more easily explained than, the Mitchell et al technique. We, therefore, adopt the strategy of explaining the Young et al technique, pointing out the differences from the other techniques as we go. We will be defining an algorithm which we will call the focussing algorithm.

The description space, in the focussing algorithm, consists of a set of relation trees (see figure 4-1). Each node of the tree is labelled with a relation; relations in the same tree being applied to the same arguments. The label of the root node is the relation which is always true. The label of a node is logically equivalent to the exclusive disjunction of the labels of its daughters, i.e. the arrangement



implies that

p(X) <-> [q(X) ⊻ r(X) ⊻ s(X)]

These trees make explicit the relationship between a proposition and its negation by arranging them as the labels on the two daughters of the root node. A tree consisting only of a root node with two daughters will be called a minimal tree. The singular/plural and definite/indefinite trees of figure 4-1 are minimal.

*= means 'is syntactically identical to'.

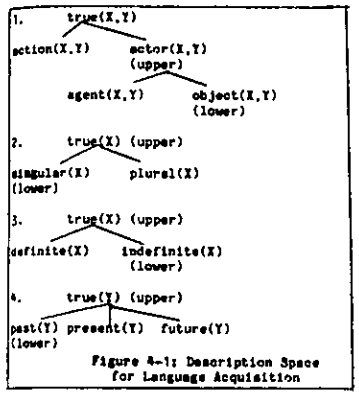


Figure 4-1: Description Space for Language Acquisition

This description space allows a partially specified rule hypothesis to be represented. During the course of rule learning this partially formed hypothesis is gradually firmed up until it is completely specified. The partial representation is achieved by placing two markers in each tree: an upper mark and a lower mark as in figure 4-1.

Any relation above the upper mark is outside the concept, e.g. action(X,Y) and true(X,Y). Any relation below the lower mark is inside the concept, e.g. object(X,Y). Any relation between the upper and lower marks is in a grey area, about which the program is not sure, e.g. agent(X,Y) and actor(X,Y). The condition is completely specified when the upper and lower marks coincide. The rule is completely specified when each of its conditions is completely specified. The focussing algorithm works by moving the upper marks down and/or the lower marks up, until they coincide.

Since the rule is only partially specified by the description space, there is some ambiguity about what rule to use when forming program traces. In particular, we can take two extreme views:

- The Most General View: that the hypothesis is specified by the conjunction of relations labelling its upper marks, which leads the rule to make errors of commission; and

- The Most Specific View: that the hypothesis is specified by the conjunction of relations labelling its lower marks, which leads the rule to make errors of omission.

For the sake of definiteness and to facilitate comparison with the last section, we will adopt the most general view. Furthermore, since root relations are always true we will omit them from the hypothesis. Thus the rule represented by the description space in figure 4-1 is

describe(X) & actor(X,Y) -> prefix(X,a)

rather than

describe(X) & object(X,Y)
& singular(X) & indefinite(X)
& past(Y)
-> prefix(X,a)

Also, for the sake of definiteness, we will assume that the rules are fired forwards. Neither of these restrictions is serious, since the algorithms for the other cases are duals of the one described below.

The partial representation of a rule provided by a description space is similar to the version space representation used by Mitchell et al. They record two sets: S, the set of most specific rules implied by the evidence so far; and G, the set of most general rules implied by the evidence so far.* For instance, the version space corresponding to the description space in figure 4-1, is:

S: {describe(X) & object(X,Y)
& singular(X) & indefinite(X) & past(Y)
-> prefix(X,a)}

G: {describe(X) & actor(X,Y) -> prefix(X,a)}

The version space representation is more compact that the description space representation, but the explanation of the focussing algorithm is more messy. Version spaces do not explicitly record a piece of information vital to the algorithm, namely the correspondence between the conditions in the different rules, e.g. between object(X,Y) in S and actor(X,Y) in G.

The Brazdil/Langley discrimination technique of the last section corresponds to moving the upper mark down from the root to a tip of a minimal tree. We will enrich the meaning of discrimination to cover all cases in which near/far misses cause the upper mark to descend.

The ascending of lower marks does not correspond to any technique used by Brazdil or Langley. It is done when the critic provides a positive training instance of a rule and it generalizes the hypothesis of that rule. In [Winston 75] it corresponds to the generalization of a concept when new examples of the concept are provided. We will call this step generalization.

The focussing algorithm does not just compare the current context with a single previous context, but with all previous contexts. This is possible because all previous contexts, both selection and rejection, are summarized by the positions of the upper and lower marks in the relation trees. We need only compare the current context with the current positions of these marks. If the critic has provided us with a positive training instance then we will have a selection context, and will apply generalization. If the critic has provided us with a commission error then we will have a rejection context, and will apply discrimination. To some extent generalization and discrimination are dual processes, but this duality is not complete and the reader should beware of assuming that it is.

We now consider generalization and discrimination in more detail.

4.3.1. Generalization

The input to generalization consists of: the selection context of a correct application of a rule; and the description space of the rule. The output consists of new lower marks for some of the trees. Each tree is considered in turn and the following steps executed.

(a)For each of the relations labelling a tip node, determine its truth value in the selection context.

(b)Exactly one of these relations will be true in the selection context, label its node, the current node.

(c)Find the least upper bound of the current node and the current lower mark and make this the new lower mark.

For instance, suppose that the rule

describe(X) -> prefix(X,a)

has been correctly applied in the selection context

{dog/X, event1/Y}

and that position of the marks in the relation trees are as in figure 4-2. The tip relations which are true in the selection context are:

agent(dog,event1),
singular(dog),
indefinite(dog),
present(event1)

*Why G is a set of rules will become evident below.

This defines the current nodes as marked in figure 4-2. Taking the least upper bound between each current node and lower mark gives the new lower marks marked in figure 4-2. Note that the lower mark for trees 2 and 3 is unchanged, but that the lower mark of tree 1 moves to 'actor(X,Y)' and the lower mark for tree 4 moves to 'true(Y)'. Despite these changes to the lower marks of the description space, the rule does not change form, because it is determined by the upper marks. However, generalization does have an effect on the rule learning process, because the lifting of the lower marks can limit the choices available to discrimination, as we will see in the next section.

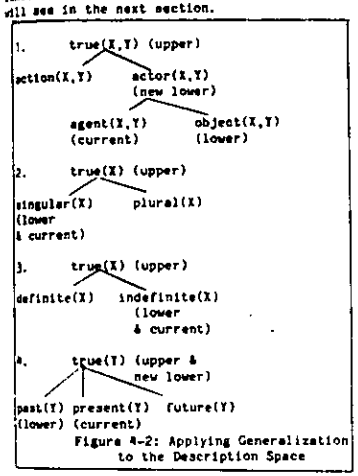


Figure 4-2: Applying Generalization to the Description Space

The version space corresponding to the new lower marks of figure 4-2 is:

S: {describe(X) & actor(X,Y) &
singular(X) & indefinite(X)
-> prefix(X,a)}

G: {describe(X) -> prefix(X,a)}

4.3.2. Discrimination

The input to discrimination consists of: the rejection context of an incorrect application of a rule; and the description space of the rule. The output consists of a new upper mark for exactly one of the trees.* Each tree is considered in turn and the following steps executed.

(a)For each of the relations labelling a tip node, determine its truth value in the rejection context.

(b)Exactly one of these relations will be true in the rejection context, label its node, the current node. Note that the current node must lie below the upper mark, otherwise the rule could not have fired.

(c)If the current node lies below the lower mark then mark the tree as a white tree.

(d)Otherwise, the current node must lie between the upper and lower marks. Mark the tree as a grey tree.

At least one of the trees must be grey, otherwise the rule application would be correct. If just one tree is grey then we are in a near miss situation. If more than one tree is grey then we are in a far miss situation. Only one of the grey trees can have its upper mark lowered. We call this grey tree the discriminant. Far misses present a trilemma:

- depth first: We can pick one of the grey trees as discriminant;

- breadth first: Or create a new rule for each grey tree;

- zero option: Or we can do nothing.

Either of the first two choices may lead to the creation of rules which are over constrained and may give rise to errors of omission. Such rules should be deleted. In the case of depth first search the program should then backup and chose another discriminant. The breadth first option corresponds to Langley's solution to far misses, as described in the last section. Brazdil's solution cannot be adopted here without violating the relation tree representation of the rule hypothesis, but it is similar to the version space solution (see below).

Once the discriminant has been picked its upper mark is lowered, just enough to exclude the current node. This is done by setting the new upper mark to be the least upper bound of the current node and the lower mark.

To illustrate discrimination suppose that the rule

describe(X) -> prefix(X,a)

has been incorrectly applied in the rejection context

{chases/X, event2/Y}

and that position of the marks in the relation trees are as in figure 4-2. The tip relations which are true in the rejection context are:

action(chases,event2),
singular(chases),
indefinite(chases),
present(event2)

This defines the current nodes marked in figure 4-3. Trees 2 and 3 are white and trees 1 and 4 are grey. If tree 1 is chosen as the discriminant then action(X,Y) can be excluded by lowering the upper mark from true(X,Y) to actor(X,Y). The new rule is:

describe(X) & actor(X,Y) -> prefix(X,a) (1)

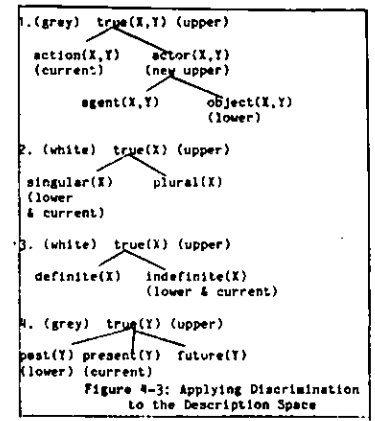


Figure 4-3: Applying Discrimination to the Description Space

If tree 4 had been picked as the discriminant then the new rule would have been:

describe(X) & past(Y) -> prefix(X,a)

Since the tense of an utterance does not affect whether the article a should prefix actors then this rule would eventually be guilty of an error of omission, e.g. in the context {dog/X, event4/Y}, where present(event4), the rule would not fire when it should. At this stage the rule should be deleted, and if the alternative rule, (1), has not already been formed, it should be formed now.

*Note lack of duality.