Specialization of Logic Programs by Pruning SLD-Trees

Henrik Boström and Peter Idestam-Almquist

Dept. of Computer and Systems Sciences
Stockholm University
Electrum 230, 164 40 Kista, Sweden
henke,pi@dsv.su.se

Abstract

The problem of finding an inductive hypothesis by specializing a logic program w.r.t. positive and negative examples can be viewed as the problem of pruning an SLD-tree such that all refutations of negative examples and no refutations of positive examples are excluded. It is shown that the actual pruning can be performed by applying unfolding and clause removal. The algorithm SPECTRE is presented, which is based on this idea. The input to the algorithm is, besides a logic program and positive and negative examples, a computation rule, which determines the shape of the SLD-tree that is to be pruned. It is shown that the generality of the resulting specialization is dependent on the computation rule, and experimental results are presented from using three different computation rules. The experiments indicate that the computation rule should be formulated so that the number of applications of unfolding is kept as low as possible. The algorithm, which uses a divide-and-conquer method, is also compared with a covering algorithm. The experiments show that a higher predictive accuracy can be achieved if the focus is on discriminating positive from negative examples rather than on achieving a high coverage of positive examples only.

1 Introduction

The search for an inductive hypothesis can be performed either top-down, i.e. from an overly general hypothesis to a more specific, or bottom-up, i.e. from an overly specific hypothesis to a more general. In the field of inductive logic programming, the top-down search for an inductive hypothesis has been performed by using various specialization techniques [14, 2, 8, 12, 10, 3, 7, 13, 16, 9].

In this work, we assume that the overly general hypothesis is given as a logic program. The problem of finding an inductive hypothesis which is consistent with given sets of positive and negative examples then becomes the problem of specializing the program such that all negative but no positive examples are excluded.

In the next section, we show that this problem can be viewed as the problem of pruning an SLD-tree such that all refutations of negative examples and no refutations of positive examples are excluded. We show that the actual pruning can be performed by applying unfolding and clause removal, and we present the algorithm SPECTRE[1], which is based on this idea. The input to the algorithm is, besides a logic program and positive and negative examples, a computation rule, which determines the shape of the SLD-tree that is to be pruned.

The major difference between this algorithm and the previous approaches that use specialization to find inductive hypotheses is the search method. According to the standard terminology [12], the previous approaches (except for [2]) use covering methods, like the AQ family [6], while SPECTRE uses a divide-and-conquer method, like ID3 [11]. The specialization technique in [2] also uses a divide-and-conquer method but is, in contrast to SPECTRE, restricted to pruning SLD-trees which shapes are determined by the standard Prolog computation rule. One of the major contributions of this work is to show that the choice of computation rule is crucial for the result when pruning SLD-trees.

In section three, experimental results are presented when using SPECTRE together with three different computation rules. We also compare the performance of SPECTRE with an algorithm that uses a covering method. In section four, we relate our approach to previous work on the specialization problem. In section five, we discuss limitations of the algorithm and present ways of overcoming these. Finally, in section six we give some concluding remarks and point out directions for future research. In the following, we assume the reader to be familiar with the standard terminology in logic programming [5].

[1] SPECialization by TRansformation and Elimination

2 Specialization by Pruning SLD-trees

We first define the specialization problem that is addressed in this work. We then show how this problem can be solved by pruning SLD-trees. Finally, we present the algorithm SPECTRE, which is based on this technique.

2.1 The Specialization Problem

The problem of specializing a logic program (definite program) w.r.t. positive and negative examples can be stated as follows:

Given: a definite program P and two disjoint sets of ground atoms E^+ and E^- (positive and negative examples).
Find[2]: a definite program P', called a *specialization* of P w.r.t. E^+ and E^-, such that $M_{P'} \subseteq M_P$, $E^+ \subseteq M_{P'}$ and $M_{P'} \cap E^- = \emptyset$.

2.2 Pruning SLD-trees by Unfolding and Clause Removal

In this work we assume that all positive and negative examples are instances of the same atom T, referred to as the *target predicate*. SLD-refutations of all the examples are then included in an SLD-tree of $P \cup \{\leftarrow T\}$. This means that for each SLD-refutation of a particular example, there is a branch in the corresponding SLD-tree leading from the root to the empty goal ($\square$).

In Figure 1, an SLD-tree is shown in which leaves that correspond to refutations of positive and negative examples are labeled '+' and '−' respectively, and leaves that do not correspond to refutations of any examples are unlabeled. The broken lines show two places where the SLD-tree can be pruned, such that all refutations of the negative examples and no refutations of the positive examples are excluded.

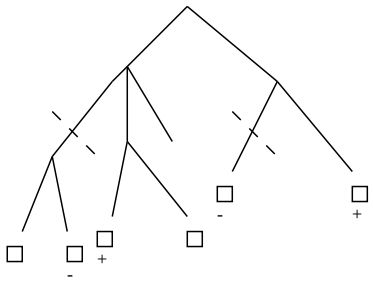

Figure 1: Two places to prune in an SLD-tree.

[2] M_P denotes the least Herbrand model of P.

Since an SLD-tree is an abstract entity, the immediate question is how the pruning of such a tree can be performed in practice, i.e. what alterations should be made to the corresponding program. The answer is that the pruning can be performed by removing clauses. However, a correct specialization can not in general be obtained by simply removing clauses in the original program, since the removal of a clause may prune several parts of the SLD-tree (possibly including refutations of positive examples). Instead, the pruning can appropriately be performed by combining clause removal with the use of the transformation rule unfolding[3] [15] in the following way.

By applying unfolding, internal nodes in the SLD-tree can be eliminated, i.e. some sub-trees can be moved closer to the root. By repeated application of unfolding, any sub-tree can be made a direct sub-tree of the root. Since each direct sub-tree corresponds to one clause defining the target predicate, any sub-tree can be pruned by first applying unfolding and then removing the clause corresponding to the (direct) sub-tree. In this way the clause removal does only affect one part of the SLD-tree. This is illustrated by Figure 2, which shows the SLD-tree after having applied unfolding once in each clause defining the target predicate. In the transformed program, there are five clauses defining the target predicate, compared with two clauses in the original program. Two of the new clauses correspond to sub-trees that include refutations of negative examples only, and if these are removed a specialization has been obtained satisfying the above definition.

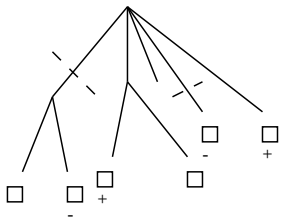

Figure 2: SLD-tree after having applied unfolding.

2.3 An Example

Assume that we are given the program in Figure 3, which can be used for determining which cards should be rewarded, together with the following sets of positive and negative examples:

[3] When unfolding upon a literal L in a clause C in a definite program P, C is replaced with the resolvents of C and each clause in P whose head unifies with L.

$E^+ = \{$ `reward(spades,7)`, `reward(clubs,3)`$\}$
$E^- = \{$ `reward(hearts,5)`, `reward(clubs,j)`$\}$

```
reward(S,R):- suit(S), rank(R).
suit(S):- black(S).
suit(S):- red(S).
rank(R):- num(R).
rank(R):- face(R).
black(spades).
black(clubs).
red(hearts).
red(diamonds).
num(1). ... num(10).
face(j). face(q). face(k).
```

Figure 3: A program for determining which cards should be rewarded.

Then all refutations of the examples are included in an SLD-tree of the target predicate `reward(S,R)`, and they use the same input clause first:

```
reward(S,R):- suit(S),rank(R).
```

Clearly, the removal of this clause would exclude all refutations of the negative examples but also all refutations of the positive examples. Instead, we can apply unfolding upon one of the literals in the body of the clause. Unfolding upon `suit(S)` replaces the clause with the following two clauses:

```
reward(S,R):- black(S), rank(R).
reward(S,R):- red(S), rank(R).
```

The first of these clauses corresponds to a sub-tree that includes refutations of the two positive examples and one negative example, while the second corresponds to a sub-tree that includes a refutation of one negative example only. We can thus remove the second clause without excluding a refutation of a positive example. Since we cannot remove the first clause, we have to unfold upon a literal in its body. Unfolding upon `rank(R)` replaces the clause with the following two clauses:

```
reward(S,R):- black(S), num(R).
reward(S,R):- black(S), face(R).
```

The first of these clauses corresponds to a sub-tree that includes refutations of two positive and no negative examples, while the second corresponds to a sub-tree that includes a refutation of one negative example only. We can thus remove the second clause without excluding a refutation of a positive example. Hence, the resulting specialization consists of the following clause:

```
reward(S,R):- black(S), num(R).
```

The corresponding SLD-tree includes all refutations of the positive examples, and no refutations of the negative examples.

2.4 The Algorithm SPECTRE

The algorithm SPECTRE takes as input a definite program, two sets of atoms (positive and negative examples) and a computation rule[4]. The output of the algorithm is a new definite program that covers[5] all positive but no negative examples.

The algorithm consists of one main loop that continues until no negative examples are covered. When a clause is found that covers a negative example, and no positive examples, it is removed. When a clause is found that covers both a negative and a positive example, it is unfolded. The choice of which literal to unfold upon is made using the computation rule.

Algorithm SPECTRE
Input: a definite program P, two disjoint sets of ground atoms E^+ and E^- and a computation rule R.
Output: a specialization P' of P w.r.t. E^+ and E^-.

LET $P' = P$.
WHILE there is a clause C in P' that covers an atom in E^- DO
    IF C does not cover any atom in E^+
    THEN
        remove C from P'
    ELSE
        unfold upon the literal in C that is selected by R.

The choice of computation rule is crucial to the performance of the algorithm. For example, given the program and examples in section 2.3 together with the standard Prolog computation rule (i.e. always select the leftmost goal), SPECTRE produces the following specialization:

```
reward(spades,R):- rank(R).
reward(clubs,R):- numbered(R).
```

This definition of the target predicate is more general than the specialization obtained in section 2.3, since the spades are not required to be numbered as in the previous case.

[4] The term computation rule is used in a broader sense than in [5]. We say that a computation rule is a function from a set of nodes (rather than a set of goals) to a set of literals.

[5] We say that a clause $A \leftarrow B_1, \ldots, B_n$ in a definite program P covers an atom $A\theta$ if and only if there is an SLD-refutation of $P \cup \{(\leftarrow B_1, \ldots, B_n)\theta\}$.

3 Empirical Evaluation

In this section we empirically evaluate the performance of the algorithm SPECTRE together with three different computation rules in four domains. We also compare SPECTRE with an algorithm that uses a covering method.

3.1 Computation Rules

The computation rules that are considered in this empirical evaluation are the following: the standard Prolog computation rule (i.e. select the leftmost literal), a computation rule that randomly selects a literal and a computation rule that minimizes the residual impurity of the resolvents when having applied unfolding.

The hypothesis is that the number of applications of unfolding should be kept as low as possible, since the number of clauses increases when unfolding is applied. The underlying motivation is that the fewer clauses defining the target predicate the more unlikely it is that each clause will only cover positive or negative examples. Therefore the fewer clauses that are needed in the definition of the target predicate to discriminate between positive and negative examples the more likely it is not by coincidence, cf. the preference of simple decision trees in the TDIDT-paradigm [11]. The measure of residual impurity that is used in this experiment is the same as the one used in ID3 [11]:

$$-c\sum_{i=1}^{n} cov(C_i, E^+)\log_2\frac{cov(C_i, E^+)}{cov(C_i, E^+ \cup E^-)} + cov(C_i, E^-)\log_2\frac{cov(C_i, E^-)}{cov(C_i, E^+ \cup E^-)}$$

where $C_1, \ldots, C_n$ are the resolvents upon one of the literals in the current clause C and $cov(C_i, E)$ denotes the number of elements in E that are covered by C_i. The constant c is $1/cov(C, E^+ \cup E^-)$.

3.2 Domains

The domains that are used in this empirical evaluation are taken from the UCI repository of machine learning databases and domain theories. These are: King+Rook versus King+Pawn on a7, Space Shuttle Autolanding, US Congressional Voting Records 1984 and Tic-Tac-Toe.

The example sets in the UCI repository are represented by attribute vectors, and have to be transformed into ground atoms in order to be used together with SPECTRE. The number of examples is 3196 in the first domain (of which 52.2% are positive), 278 in the second domain (of which 52.1% are negative), 435 in the third domain (of which 61.4% are positive) and 958 in the fourth domain (of which 65.3% are positive).

Since SPECTRE also requires overly general programs as input, such programs are constructed for this evaluation in the following way (cf. [3]). For each domain a new target predicate is defined with as many arguments as the number of

attributes, and for each attribute a new predicate is defined to determine the possible values of the attribute. This technique is illustrated by the following overly general program for determining when to allow automatic landing in the Space Shuttle domain:

```
auto(Stability,Error,Sign,Wind,Magnitude,Visibility):-
        stability(Stability),
        error(Error),
        sign(Sign),
        wind(Wind),
        magnitude(Magnitude),
        visibility(Visibility).
stability(stab).
stability(xstab).
...
```

In order to see how additional knowledge affects the accuracy of the resulting specialization, an alternative formulation of the Tic-Tac-Toe program is tested as well, where a new intermediate predicate is introduced. The original formulation is:

```
win_for_x(S1,S2,S3,S4,S5,S6,S7,S8,S9):-
        square(S1), square(S2), square(S3),
        square(S4), square(S5), square(S6),
        square(S7), square(S8), square(S9).
square(x).
square(o).
square(b).
```

In the alternative formulation, the definition of the predicate `square(S)` is changed into the following:

```
square(x).
square(S):- o_or_b(S).
o_or_b(o).
o_or_b(b).
```

The hypothesis is that the new intermediate predicate will reduce the number of clauses in the resulting specialization, and hence increase the accuracy. The reason for this is that it does not matter in the correct definition of the target predicate whether a square has the value *o* or *b*.

3.3 Experimental Results

An experiment is performed with each domain, in which the example set is randomly split into two halves, where one half is given as input to SPECTRE and the other half is used for evaluating the output of the algorithm. The same training and test sets are used for all three computation rules. Each experiment is iterated 100 times and the mean accuracy on the test examples together with the mean number of clauses defining the target predicate in the specializations are presented. In addition, the accuracies on the positive and negative test examples are presented.

In Table 1 the results from the King-Rook versus King-Pawn domain are presented. It can be seen that when using SPECTRE together with the impurity measure, the number of clauses defining the target predicate is significantly fewer and the accuracy is significantly higher than when using SPECTRE together with any of the other computation rules.

Comp. rule	*# clauses*	*Accuracy (%)*	*Pos. acc. (%)*	*Neg. acc. (%)*
Prolog's	2240.9	76.7	83.4	69.5
Random	1411.2	81.8	86.9	76.2
Min. imp.	15.4	99.0	99.3	98.6

Table 1: Results from the KR vs. KP domain.

In Table 2 the results from the Space Shuttle Autolanding domain is shown. The same relation between the three computation rules holds as for the previous domain: SPECTRE with the impurity measure produces fewer clauses with higher accuracy than SPECTRE with leftmost or random selection of literals to unfold upon. It should be noted that SPECTRE with Prolog's computation rule performs extremely bad, with an accuracy less than 40%. This can be explained by the fact that the most informative literal is the rightmost in the clause defining the target predicate, which means that this literal is considered last by Prolog's computation rule.

Comp. rule	*# clauses*	*Accuracy (%)*	*Pos. acc. (%)*	*Neg. acc. (%)*
Prolog's	79.0	37.5	54.4	22.4
Random	33.4	79.3	84.5	74.9
Min. imp.	7.6	96.6	96.9	96.5

Table 2: Results from the Space Shuttle domain.

In Table 3 the results from the US Congressional Voting Records domain are shown. Again, SPECTRE with the impurity measure produces fewer clauses with higher accuracy than SPECTRE with leftmost or random selection of literals. But this time, SPECTRE with Prolog's computation rule is more accurate (and produces fewer clauses) than SPECTRE with random selection.

Comp. rule	*# clauses*	*Accuracy (%)*	*Pos. acc. (%)*	*Neg. acc. (%)*
Prolog's	86.1	91.8	95.4	86.3
Random	107.2	87.1	92.4	78.8
Min. imp.	16.1	93.6	95.1	91.3

Table 3: Results from the Voting Records domain.

In Table 4 the results from the Tic-Tac-Toe domain are shown. Again, SPECTRE with the impurity measure produces fewer clauses with higher accuracy than SPECTRE with leftmost or random selection of literals, and SPECTRE with Prolog's computation rule is more accurate than SPECTRE with random selection.

Comp. rule	*# clauses*	*Accuracy (%)*	*Pos. acc. (%)*	*Neg. acc. (%)*
Prolog's	237.2	75.3	87.0	53.4
Random	278.8	71.2	84.5	46.4
Min. imp.	89.2	83.7	89.2	73.5

Table 4: Results from the Tic-Tac-Toe domain.

In Table 5 the results from the altered Tic-Tac-Toe domain are shown. The hypothesis was that the alteration would improve the accuracy, since the number of clauses would be reduced. It can be observed that this is in fact the case. However, the accuracy of SPECTRE with Prolog's computation rule is not significantly improved, although the number of clauses is reduced.

Comp. rule	*# clauses*	*Accuracy (%)*	*Pos. acc. (%)*	*Neg. acc. (%)*
Prolog's	167.1	75.7	84.6	59.0
Random	154.4	78.3	87.8	60.2
Min. imp.	43.5	90.4	93.5	84.5

Table 5: Results from the altered Tic-Tac-Toe domain.

In summary, SPECTRE together with the computation rule that minimizes the residual impurity produces more accurate programs with fewer clauses in all five experiments than the programs produced by SPECTRE together with the computation rules that select the leftmost or a random literal. In three experiments random selection outperforms leftmost selection, while the opposite holds in the other two experiments. The last experiment also illustrates that the accuracy of the resulting specialization may be improved by introducing new predicates.

In order to see how the divide-and-conquer method used by SPECTRE performs compared with a covering method, the experiments are repeated with the following covering algorithm:

REPEAT
    LET C be one of the original clauses defining the target predicate.
    WHILE C covers a negative example DO
        Replace C with one of its resolvents.
    Add C to the new definition of the target predicate.
UNTIL all positive examples are covered by the new definition.

The selection of resolvent is made so that the residual impurity is minimized using the following impurity measure:

$$-cov(R, E^{\oplus}) \log_2 \frac{cov(R, E^{\oplus})}{cov(R, E^{\oplus} \cup E^{-})}$$

where R is the resolvent and $E^{\oplus}$ are the positive examples that are not yet covered by the new definition. Note that this impurity measure only credits a high coverage of positive examples in contrast to the impurity measure used with SPECTRE, which credits a high coverage of either positive or negative examples.

In Table 6 and Table 7, it can be seen that the number of clauses in the resulting specialization is larger in all cases for the covering algorithm than for SPECTRE. This is also reflected in the accuracy in the first four cases, while the covering algorithm outperforms SPECTRE in the fifth case.

Domain	*# clauses*	*Accuracy (%)*	*Pos. acc. (%)*	*Neg. acc. (%)*
KR vs. KP	414.0	85.9	97.6	73.2
Space Shuttle	11.5	94.5	98.2	91.2
Voting Records	21.7	92.2	95.9	86.4
Tic-Tac-Toe	196.4	73.5	92.9	37.2
Tic-Tac-Toe (alt.)	58.2	95.6	100.0	87.5

Table 6: Results for the covering algorithm.

Domain	*# clauses*	*Accuracy (%)*	*Pos. acc. (%)*	*Neg. acc. (%)*
KR vs. KP	15.4	99.0	99.3	98.6
Space Shuttle	7.6	96.6	96.9	96.5
Voting Records	16.1	93.6	95.1	91.3
Tic-Tac-Toe	89.2	83.7	89.2	73.5
Tic-Tac-Toe (alt.)	43.5	90.4	93.5	84.5

Table 7: Results for SPECTRE with the impurity measure.

4 Related Work

Previous work on the specialization problem has concerned both how to minimally [4, 1] and non-minimally specialize logic programs [14, 2, 8, 10, 3, 7, 13, 16, 9, 17]. Our approach to the specialization problem falls into the second category, and in this section it will be related to the other approaches in this category. They can be divided into three (not disjoint) groups with respect to their specialization techniques: clause removal, addition of literals and goal reduction.

The only specialization operator that is used in [8, 7, 16] is clause removal. In contrast to SPECTRE, the only clauses that are considered for removal in these approaches are clauses appearing in the original program. As a consequence, these approaches will not be able to produce correct specializations when all clauses in a refutation of a negative example are used in refutations of positive examples. For example, if we are given the negative example `p(a,a)` and the positive examples `p(a,b)` and `p(b,a)`, it is not possible to obtain a correct specialization by simply removing clauses in the following program:

```
p(X,Z):- q(X,Y), r(Y,Z).
q(a,b).
q(b,b).
r(b,a).
r(b,b).
```

In [17] a technique for incremental specialization is presented, called MBR. It excludes a negative example by preventing a (minimal) set of clause applications from being used. This is achieved through clause removal and addition of literals. In contrast to SPECTRE, MBR does not consider any positive examples when specializing a program. This means that positive examples that have been used in the construction of the program may no longer be covered by the specialization.

In [14, 2, 12, 10, 13, 9], clauses are specialized by adding literals to their bodies. The literals considered for being added are in these approaches restricted to those whose predicate symbols are defined in the original program. Various restrictions are also put on the variables in the literals (e.g. at least one of the variables should appear elsewhere in the clause [12]). The branching factor in the space of possible refinements is normally much larger for these approaches than for SPECTRE. The number of alternatives for SPECTRE when refining a clause is equal to the number of literals in the body of the clause. The number of possible literals to add is m^n for each predicate symbol of arity n, where m is the number of variables that are allowed in the new literal. For example, assume that there are 10 binary and 10 ternary predicates defined, and that we are given the following clause:

```
p(X,Y):- q(X,Z), r(Z,W), s(W,V), t(V,Y).
```

Then the number of alternatives for SPECTRE is 4, while the number of possible literals to add is $10 * 5^2 + 10 * 5^3 = 1500$ (assuming that the variables are restricted to those already present in the clause as in [14]).

It should be noted that SPECTRE reduces the branching factor at the cost of expressive power, i.e. all specializations produced by SPECTRE can be obtained by the previous approaches, while the contrary does not hold. However, it should also be noted that one refinement step in SPECTRE normally comprise several steps in the previous approaches, since one goal reduction may introduce more than one literal. This means that SPECTRE may find refinements that are missed by techniques that do not search the space exhaustively (e.g. FOIL [12]). The same idea is the basis for the FOCL [10] and FOCL-FRONTIER [9] algorithms. However, FOCL and FOCL-FRONTIER use goal reduction in addition to adding literals, which means that the branching factor is increased rather than decreased in these approaches. Another algorithm that utilizes goal reduction to specialize clauses is ANA-EBL [3], which, like SPECTRE, does not consider other possible specializations. The major difference between SPECTRE and ANA-EBL, FOCL and FOCL-FRONTIER is the way in which the search for a specialization is performed. The algorithms ANA-EBL, FOCL and FOCL-FRONTIER, like MIS [14] and FOIL, use covering methods for finding correct specializations, while SPECTRE uses a divide-and-conquer method. This means that ANA-EBL, FOCL, FOCL-FRONTIER, MIS and FOIL in this aspect resemble the AQ family [6], while SPECTRE resembles ID3 [11].

The specialization technique in the ML-SMART system, which is presented in [2], is closely related to SPECTRE. Minor differences between the techniques are that ML-SMART uses best-first search and allows addition of literals while SPECTRE uses hill-climbing and is restricted to goal reduction. One major difference is however that ML-SMART only considers reducing the leftmost goal, while SPECTRE reduces the goal that is selected by the computation rule given as input to the algorithm. This means that ML-SMART is restricted to pruning SLD-trees, which shapes are determined by the standard Prolog computation rule. In the previous section, the experimental results showed that a dynamic computation rule, which considers the training examples, can give significantly more accurate specializations than this static computation rule.

5 Limitations of the Algorithm SPECTRE

In this section, we present two cases when special care must be taken when applying the algorithm SPECTRE: when the target predicate is recursive and when the same sequence of input clauses is used both in a refutation of a positive and a negative example.

5.1 Recursive Target Predicates

The removal of a clause that defines a (directly or indirectly) recursive target predicate not only prunes a direct sub-tree of the root of an SLD-tree, but also prunes all branches in the entire tree which include the removed clause. This means that even if the direct sub-tree corresponding to the clause does not include any refutation of a positive example, it might not be possible to remove the clause without excluding a refutation of a positive example in some other part of the SLD-tree. For this reason, the definition of a recursive target predicate is assumed to be transformed into an equivalent non-recursive definition in the following way.

Let A be the recursive target predicate. Then introduce a new predicate A' by adding a clause $A' \leftarrow A$, where the arguments of A' are all variables in A. Unfold upon A in the clause, and replace each instance $A\theta$ in the bodies of the clauses defining A and A' (directly or indirectly) with $A'\theta$ (i.e. folding [15]). Then an equivalent definition of A has been obtained which is non-recursive. For example, let the recursive target predicate be `member(X,Y)`, which is defined by the two clauses:

```
member(X,[X|L]).
member(X,[Y|L]):- member(X,L).
```

The non-recursive definition of `member(X,Y)` would then be:

```
member(X,[X|L]).
member(X,[Y|L]):- rec_member(X,L).
rec_member(X,[X|L]).
rec_member(X,[Y|L]):- rec_member(X,L).
```

It should be noted that although it is always possible to avoid the problem with recursive target predicates, there are cases when SPECTRE is unable to find the desired specialization. For example, assume that the following definition of the predicate `odd(X)` is given:

```
odd(0).
odd(s(X)):- odd(X).
```

together with the following positive and negative examples:

$E^+ = \{$ `odd(s(0))`, `odd(s(s(s(0))))`, `odd(s(s(s(s(s(0))))))`$\}$
$E^- = \{$ `odd(0)`, `odd(s(s(0)))`, `odd(s(s(s(s(0)))))` $\}$

Then the specialization produced by SPECTRE, after having applied the above transformation, will exclude the negative examples only (i.e. a maximally general specialization is obtained):

```
odd(s(0)).
odd(s(s(s(0)))).
odd(s(s(s(s(s(X)))))):- rec_odd(X).
rec_odd(0).
rec_odd(s(X)):- rec_odd(X).
```

Removing the third element from E^+ results in a specialization which includes the positive examples only (i.e. a maximally specific specialization is obtained).

5.2 Ambivalent Leaves

When some leaf in an SLD-tree is labeled with both '+' and '−' (which is referred to as being ambivalent), it means that the same sequence of input clauses is used both in a refutation of a positive example and in a refutation of a negative example. In that case there is no application of unfolding that will result in a clause that can be removed without excluding a proof of a positive example. However, this problem can be solved in the following way.

With no loss of generality, we can assume that all terms in the Herbrand universe is defined by a predicate `term(X)`[6]. Then we can rewrite the original program by adding to each clause defining the target predicate, a literal `term(X)` for each variable `X` in the head of the clause. Then each example will have a unique branch in the SLD-tree. For example, assume that the original program consists of one clause: `p(X)`, and that the Herbrand universe is $\{0, s(0), s(s(0)), ...\}$. Then the program can be written as:

```
p(X):- term(X).
term(0).
term(s(X)):- term(X).
```

Whereas in an SLD-tree of the original program the refutations of the two examples `p(0)` and `p(s(0))` follow identical branches, they follow different branches in an SLD-tree of the transformed program.

[6]One clause is needed for each constant and function symbol.

6 Concluding Remarks

The problem of specializing a logic program w.r.t. to positive and negative examples can be viewed as the problem of pruning an SLD-tree such that all refutations of negative examples and no refutations of positive examples are excluded. The actual pruning can be performed by applying unfolding and clause removal. We have presented the algorithm SPECTRE, which is based on this idea. The input to the algorithm is, besides a logic program and positive and negative examples, a computation rule, which determines the shape of the SLD-tree that is to be pruned and the generality of the resulting specialization.

The experimental results indicate that the computation rule should be formulated so that the number of applications of unfolding is kept as low as possible. Since it is computationally expensive to find the optimal choice of which literal to unfold upon, it can be approximated by choosing the literal which minimizes the residual impurity of the clauses produced by unfolding. The experiments showed that SPECTRE together with this computation rule produces more accurate hypotheses than together with leftmost or random selection of literals. SPECTRE was also compared with a covering algorithm, and in four of five cases, SPECTRE outperformed the other algorithm. This result shows that a higher accuracy can be achieved if the focus is on discriminating positive from negative examples rather than on achieving a high coverage of positive examples only.

Two problems that should be further investigated were discussed in section five: the handling of recursive target predicates and ambivalent leaves. Questions for future research are how to change the algorithm so that recursive specializations can be obtained, and how to handle ambivalent leaves automatically. Another question for future research has to do with redundant refutations of positive examples. As the algorithm SPECTRE is formulated now, all refutations of negative examples, and no refutations of positive examples, are excluded. This is in fact too restrictive, since what actually is desired is that all refutations of negative examples are excluded, and that not all refutations of any positive example are excluded. This means that it might be the case that a specialization can be found by excluding a (redundant) refutation of a positive example, which is not found by the current formulation of the algorithm. Another problem for future research is to include some loop prevention mechanism. In the current formulation of the algorithm, it is assumed that the cover check always terminates and that the negative examples have a finite number of refutations.

Acknowledgements

This work has been supported by the European Community ESPRIT BRA 6020 ILP (Inductive Logic Programming) and TFR (the Swedish Research Council for Engineering Sciences).

References

[1] Bain M. and Muggleton S., "Non-Monotonic Learning", in Muggleton S. (ed.), *Inductive Logic Programming*, Academic Press, London (1992) 145–161

[2] Bergadano F. and Giordana A., "A Knowledge Intensive Approach to Concept Induction", *Proceedings of the Fifth International Conference on Machine Learning*, Morgan Kaufmann, CA (1988) 305–317

[3] Cohen W. W., "The Generality of Overgenerality", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 490–494

[4] Ling C. X., "Non-Monotonic Specialization", *Proceedings of International Workshop on Inductive Logic Programming*, Portugal (1991) 59–68

[5] Lloyd J. W., *Foundations of Logic Programming*, (2nd edition), Springer-Verlag (1987)

[6] Michalski R. S., "Pattern Recognition as Rule-Guided Inductive Inference", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **2** (1980) 349–361

[7] Mooney R. J. and Ourston D., "Constructive Induction in Theory Refinement", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 178–182

[8] Ourston D. and Mooney R. J., "Changing the Rules: A Comprehensive Approach to Theory Refinement", *Proceedings of the Eighth National Conference on Artificial Intelligence*, MIT Press (1990) 815–820

[9] Pazzani M. and Brunk C., "Finding Accurate Frontiers: A Knowledge-Intensive Approach to Relational Learning", *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Morgan Kaufmann (1993) 328–334

[10] Pazzani M., Brunk C. and Silverstein G., "A Knowledge-Intensive Approach to Learning Relational Concepts", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 432–436

[11] Quinlan J. R., "Induction of Decision Trees", *Machine Learning* **1** (1986) 81–106

[12] Quinlan J. R., "Learning Logical Definitions from Relations", *Machine Learning* **5** (1990) 239–266

[13] Richards B. L. and Mooney R. J., "First-Order Theory Revision", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 447–451

[14] Shapiro E. Y., *Algorithmic Program Debugging*, MIT Press (1983)

[15] Tamaki H. and Sato T., "Unfold/Fold Transformations of Logic Programs", *Proceedings of the Second International Logic Programming Conference*, Uppsala University, Uppsala, Sweden (1984) 127–138

[16] Wogulis J., "Revising Relational Domain Theories", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 462–466

[17] Wrobel S., "On the Proper Definition of Minimality in Specialization and Theory Revision", *Proceedings of the European Conference on Machine Learning*, Springer-Verlag (1993) 65–82