Specialization of Recursive Predicates

Henrik Boström

Dept. of Computer and Systems Sciences
Stockholm Univer sity
Electrum 230, 164 40 Kista, Sweden
henke@dsv.su.se
fax: +46 8 703 90 25 tel: +46 8 16 16 16

Abstract. When specializing a recursive predicate in order to exclude a set of negative examples without excluding a set of positive examples, it may not be possible to specialize or remove any of the clauses in a refutation of a negative example without excluding any positive examples. A previously proposed solution to this problem is to apply program transformation in order to obtain non-recursive target predicates from recursive ones. However, the application of this method prevents recursive specializations from being found. In this work, we present the algorithm SPECTRE II which is not limited to specializing non-recursive predicates. The key idea upon which the algorithm is based is that it is not enough to specialize or remove clauses in refutations of negative examples in order to obtain correct specializations, but it is sometimes necessary to specialize clauses that appear only in refutations of positive examples. In contrast to its predecessor SPECTRE, the new algorithm is not limited to specializing clauses defining one predicate only, but may specialize clauses defining multiple predicates. Furthermore, the positive and negative examples are no longer required to be instances of the same predicate. It is proven that the algorithm produces a correct specialization when all positive examples are logical consequences of the original program, there is a finite number of derivations of positive and negative examples and when no positive and negative examples have the same sequence of input clauses in their refutations.

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, such as clause removal, addition of literals and goal reduction [13, 2, 9, 12, 11, 4, 15, 5, 10, 16, 3]. In this work, it is assumed that the overly general hypothesis is given as a logic program.

In [3], the algorithm SPECTRE is presented, which specializes logic programs by applying the transformation rule *unfolding* [14] together with clause removal. One limitation of the algorithm is that it assumes the target predicate to be non-recursive. The reason for this is that when excluding a negative example of a recursive predicate, it may not be possible to specialize or remove any of the clauses in the refutation of the example without excluding a positive example. It should be noted that this is a general problem that is not limited to a particular type of specialization operator, but applies to all specialization techniques that exclude negative examples by specializing or removing clauses in their refutations. A solution to this problem that has been proposed in [3] is to apply the transformation rules *definition*, *unfolding* and *folding* [14] in order to obtain non-recursive target predicates from recursive ones. The draw-back of applying this transformation technique is that recursive specializations can not be obtained, which means that many desired specializations can not be found.

In this work, we present the algorithm SPECTRE II which is not limited to specializing non-recursive predicates. The key idea upon which the algorithm is based is that it is not enough to specialize or remove clauses in refutations of negative examples in order to obtain correct specializations, but it is sometimes necessary to specialize clauses that appear only in refutations of positive examples. In contrast to SPECTRE, the new algorithm is not limited to specializing clauses defining one predicate, but may specialize clauses defining multiple predicates. Furthermore, the positive and negative examples are no longer required to be instances of the same predicate.

In the next section we give a formal definition of the specialization problem, and we also give definitions of the three transformation rules according to [14]. In section three, we exemplify the limitation of SPECTRE, and in section four, we present SPECTRE II, which overcomes this limitation. We also prove that the algorithm produces correct specializations (under some assumptions). In section five, we present some experimental results and in section six we discuss related work. Finally, in section seven we give concluding remarks and point out problems for future research. In the following, we assume the reader to be familiar with the standard terminology in logic programming [8].

2 Preliminaries

We first give a formal definition of the specialization problem that is studied, and then define the three transformation rules that are used in this work.

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: a definite program P', called a *correct specialization* of P w.r.t. E^+ and E^-, such that $M_{P'} \subseteq M_P$, $E^+ \subseteq M_{P'}$ and $M_{P'} \cap E^- = \emptyset$[1]

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

2.2 Transformation Rules

The following rules for transformation of a definite program (below referred to as P) are taken from [14], where formal definitions can be found as well as proofs of their meaning preserving properties.

Rule 1. Definition
Add to P a clause C of the form $p(x_1, \ldots, x_n) \leftarrow A_1, \ldots, A_m$ where p is a predicate symbol not appearing in P, $x_1, \ldots, x_n$ are distinct variables and $A_1, \ldots, A_m$ are literals whose predicate symbols all appear in P.

Rule 2. Unfolding
Let C be a clause in P, A a literal in its body and $C_1, \ldots, C_n$ be all clauses in P whose heads are unifiable with A. Let $C_i'(1 \leq i \leq n)$ be the result of resolving C with C_i upon A. Then replace C with $C_1', \ldots, C_n'$.

Rule 3. Folding
Let C be a clause in P of the form $A \leftarrow A_1, \ldots, A_{i+1}, \ldots, A_{i+m}, \ldots, A_n$ and C_1 be a clause that previously have been introduced by the rule of definition of the form $B \leftarrow B_1, \ldots, B_m$. If there is a substitution θ such that $A_{i+1}, \ldots, A_{i+m} = B_1, \ldots, B_m\theta$ where θ substitutes distinct variables for the internal variables of C_1 and moreover those variables do not occur in $A, A_1, \ldots, A_i$ or $A_{i+m+1}, \ldots, A_n$, then replace C by the clause $A \leftarrow A_1, \ldots, A_i, B\theta, A_{i+m+1}, \ldots, A_n$.

3 Applying SPECTRE to Recursive Predicates

The algorithm SPECTRE [3] specializes logic programs with respect to positive and negative examples by applying unfolding together with clause removal. This is done in the following way. As long as there is a clause in the program that *covers* (i.e. is used as the first input clause in the refutation of) a negative example, it is checked whether it covers any positive examples or not. If the clause covers any positive examples, then it is unfolded, otherwise it is removed. SPECTRE is not guaranteed to obtain correct specializations when specializing recursive target predicates. 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 only clause that is used in the refutation of the first negative example is the first clause in the definition. Clearly, this clause cannot be removed or specialized without excluding the positive examples, since it is used in their refutations. Thus it is not enough to consider removing or specializing clauses in refutations of negative examples in order to obtain correct specializations.

In order to avoid the problem with recursive predicates in SPECTRE, it is assumed that recursive target predicates are transformed into equivalent non-recursive definitions in the following way.

Let T be the recursive target predicate. Then introduce a new predicate T' by adding a clause $T' \leftarrow T$, where the arguments of T' are all variables in T (*definition*). Unfold upon T in the clause, and replace each instance $T\theta$ in the bodies of the clauses defining T and T' (directly or indirectly) with $T'\theta$ (*folding*). Then an equivalent definition of T has been obtained which is non-recursive.

For example, a non-recursive definition of odd(X) would then be:

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

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 we are given the above non-recursive definition of odd(X) together with the examples above. Then the specialization produced by SPECTRE 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).

4 Specializing Recursive Predicates

In this section we present the algorithm SPECTRE II, which overcomes SPECTRE's inability to produce recursive specializations. We first describe the algorithm, and then illustrate it by some examples. Finally, we prove that the algorithm produces correct specializations.

4.1 SPECTRE II

Like SPECTRE, SPECTRE II specializes logic programs by using unfolding and clause removal. The major difference between the algorithms is that while SPECTRE only applies unfolding upon clauses that appear first in refutations of both positive and negative examples, SPECTRE II may apply unfolding upon any clause that is used in a refutation of a positive example. This means that in contrast to SPECTRE, SPECTRE II is not limited to specializing clauses that are used in refutations of negative examples and to specializing clauses that define the target predicate only.

The algorithm works in two steps. First, as long as there is a refutation of a negative example, such that all input clauses are used in refutations of positive examples, a literal in an input clause is unfolded. Second, for each refutation of a negative example, one input clause that is not used in any refutation of a positive example is removed. It is not defined by the algorithm how to make the choices of which literal to unfold upon and which clause to remove. As will be shown in section 4.3, these choices can be made arbitrarily without affecting the correctness of the algorithm. However, they are crucial to the performance of the algorithm, since the generality of the resulting specialization is dependent on them. This will be further discussed in section five. A formal description of the algorithm SPECTRE II is given below.

Algorithm SPECTRE II
Input: a definite program P and two sets of ground atoms E^+ and E^-.
Output: a correct specialization P' of P w.r.t. E^+ and E^-.

```
LET $P' = P$.
WHILE there is an SLD-refutation of $P' \cup \{\leftarrow e^-\}$, for some $e^- \in E^-$,
such that each input clause $C_i$ is used in an SLD-refutation of $P' \cup \{\leftarrow e_i^+\}$,
where $e_i^+ \in E^+$ DO
        Let $C$ be a non-unit input clause in an SLD-refutation of $P' \cup \{\leftarrow e^+\}$,
        where $e^+ \in E^+$.
        Unfold upon a literal in $C$.

FOR each $e^- \in E^-$ DO
        Remove an input clause in each SLD-refutation of $P' \cup \{\leftarrow e^-\}$, that
        does not appear in any SLD-refutation of $P' \cup \{\leftarrow e^+\}$, where $e^+ \in E^+$.
```

4.2 Examples

Assume that the following definition of the predicate odd(X) is given as input to SPECTRE II:

```
(c1) odd(0).
(c2) 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 there is a negative example (e.g. `odd(0)`) for which all clauses in the refutation appear in refutations of positive examples. Let c_2 be the selected clause, and unfolding upon the literal in its body results in the following program:

```
(c1) odd(0).
(c3) odd(s(0)).
(c4) odd(s(s(X))):- odd(X).
```

Then there is no negative example for which all clauses in the refutation appear in refutations of positive examples, and thus the condition for the while-loop is false. Then for each refutation of a negative example, a clause that does not appear in a refutation of a positive example is removed (in this case the clause c_1 is removed for each refutation of the negative examples). Thus the resulting specialization produced by SPECTRE II is:

```
(c3) odd(s(0)).
(c4) odd(s(s(X))):- odd(X).
```

To see how SPECTRE II works when specializing multiple predicates, consider the following program and examples:

```
p(a).
p(f(X)):- q(X).
q(b).
q(g(X)):- p(X).
```

$E^+ =$ { `p(f(b))`, `q(g(a))`}
$E^- =$ { `p(a)`, `q(b)`}

Note that although a correct specialization can be obtained for each predicate by simply removing the input clause in the refutation of the negative example, a correct specialization w.r.t. both predicates can not be obtained in this way. However, SPECTRE II produces a correct specialization in the following way.

After the first step in the algorithm, in which unfolding is applied twice, the following program is obtained:

```
p(a).
p(f(b)).
p(f(g(X))):- p(X).
q(b).
q(g(a)).
q(g(f(X))):- q(X).
```

In the final step of the algorithm, two clauses are removed resulting in the following correct specialization:

```
p(f(b)).
p(f(g(X))):- p(X).
q(g(a)).
q(g(f(X))):- q(X).
```

4.3 Correctness of SPECTRE II

In this section we prove that SPECTRE II produces a correct specialization in a finite number of steps when all positive examples are logical consequences of the original program, there is a finite number of derivations of positive and negative examples (i.e. the program terminates for all examples) and there is no positive and negative examples that have the same sequence of input clauses in their refutations.

We first prove a lemma that states that under these assumptions, each refutation of a negative example will after a finite number of applications of unfolding have an input clause that is not used in any refutation of a positive example.

<u>Lemma</u>

Let E^+ and E^- be two sets of ground atoms and P be a definite program, such that the number of SLD-derivations of $P \cup \{\leftarrow e\}$ are finite for all $e \in E^+ \cup E^-$, and there is no $e^+ \in E^+$ and $e^- \in E^-$, such that the same sequence of input clauses is used both in an SLD-refutation of $P \cup \{\leftarrow e^+\}$ and in an SLD-refutation of $P \cup \{\leftarrow e^-\}$. Let $P'' = P$. Then after a finite number of arbitrary applications of unfolding upon clauses in P'' that are input clauses in SLD-refutations of $P'' \cup \{\leftarrow e^+\}$, where $e^+ \in E^+$, each SLD-refutation of $P'' \cup \{\leftarrow e^-\}$, for all $e^- \in E^-$, has an input clause that is not used in any SLD-refutation of $P'' \cup \{\leftarrow e^+\}$, for all $e^+ \in E^+$.

Proof: Since the length of at least one SLD-refutation of $P'' \cup \{\leftarrow e^+\}$, where $e^+ \in E^+$, decreases when applying unfolding, and the number of SLD-refutations does not increase (proven in [6]), all input clauses in SLD-refutations of $P'' \cup \{\leftarrow e^+\}$, for all $e^+ \in E^+$, will be unit clauses after a finite number of applications of unfolding. All SLD-refutations of $P'' \cup \{\leftarrow e^-\}$ of length > 1, where $e^- \in E^-$, have at least one non-unit input clause and thus have at least one clause that is not used as an input clause in any SLD-refutation of $P'' \cup \{\leftarrow e^+\}$, for all $e^+ \in E^+$. Thus it suffices to show that no unit input clause in an SLD-refutation of $P'' \cup \{\leftarrow e^-\}$ of length $= 1$, where $e^- \in E^-$, is used in an SLD-refutation of $P'' \cup \{\leftarrow e^+\}$, where $e^+ \in E^+$. Let C be a unit clause in P'', such that $C \times \{\leftarrow e^-\}^2 = \square$, for some $e^- \in E^-$, and $C = C_1 \times \ldots \times C_n$, where each C_i is a variant of a clause in P. Then there is an SLD-refutation of $P \cup \{\leftarrow e^-\}$ with input clauses $C_1, \ldots, C_n$, since $C_1 \times \ldots \times C_n \times \{\leftarrow e^-\} = \square$. Assume

^{2} $C \times D$ denotes the resolvent of C and D.

that there is some $e^+ \in E^+_i$ such that $C \times \{\leftarrow e^+\} = \Box$. Then it follows that there is an SLD-refutation of $P \cup \{\leftarrow e^+\}$, with input clauses $C_1, \ldots, C_n$, since $C_1 \times \ldots \times C_n \times \{\leftarrow e^+\} = \Box$. This contradicts the assumption that no $e^+ \in E^+$ and $e^- \in E^-$ have the same sequence of input clauses in their SLD-refutations. $\Box$

Using this lemma, we can now prove that SPECTRE II produces a correct specialization under the above assumptions.

Theorem
Let the input to SPECTRE II be two sets of ground atoms E^+ and E^- and a definite program P, such that $E^+ \subseteq M_P$, the number of SLD-derivations of $P \cup \{\leftarrow e\}$ are finite for all $e \in E^+ \cup E^-$ and there is no $e^+ \in E^+$ and $e^- \in E^-_i$ such that the same sequence of input clauses is used both in an SLD-refutation of $P \cup \{\leftarrow e^+\}$ and in an SLD-refutation of $P \cup \{\leftarrow e^-\}$. Then the algorithm produces a correct specialization P' of P w.r.t. E^+ and E^-.

Proof: According to the lemma, the while-loop in the algorithm terminates after a finite number of steps resulting in a program P'', such that each SLD-refutation of $P'' \cup \{\leftarrow e^-\}$, for all $e^- \in E^-_i$ has an input clause that is not used in any SLD-refutation of $P'' \cup \{\leftarrow e^+\}$, for all $e^+ \in E^+$. Then $P' = P'' \setminus R$, where R is the set of clauses removed in the last step of the algorithm. Since one input clause is removed for each SLD-refutation of $P'' \cup \{\leftarrow e^-\}$, for all $e^- \in E^-_i$ it follows that there is no SLD-refutation of $P' \cup \{\leftarrow e^-\}$, for any $e^- \in E^-$. Thus $M_{P'} \cap E^- = \emptyset$. Since $M_P = M_{P''}$ and thus $E^+ \subseteq M_{P''}$ and since none of the clauses in R is used as an input clause in an SLD-refutation of $P'' \cup \{\leftarrow e^+\}$, for any $e^+ \in E^+_i$ it follows that $E^+ \subseteq M_{P'}$. Since $M_P = M_{P''} \supseteq M_{P'}$, it follows that P' is a correct specialization of P w.r.t. E^+ and E^-. $\Box$

The condition that no refutations of positive and negative examples should have the same sequence of input clauses can be relaxed by slightly altering the algorithm in the following way. Instead of considering all refutations of the positive examples, only a subset needs to be considered such that each positive example has at least one refutation in the subset. Then by guaranteeing that no refutation in the subset has a sequence of input clauses in common with any refutation of a negative example, the condition can be relaxed to require only that for each positive example there is at least one refutation that does not have the same sequence of input clauses as any refutation of a negative example. This altered version of the algorithm allows that many correct specializations can be found, which can not be found by the algorithm in its original formulation.

5 Experimental Results

The choices of which literal to unfold upon and which clauses to remove when applying SPECTRE II determine the generality of the resulting specialization. Moreover, when applying the altered version of the algorithm, the selection of which refutations to consider for the positive examples are of crucial importance for the result. In the current implementation of the algorithm, these choices are made in order to minimize the number of applications of unfolding. However, since it is computationally expensive to find the optimal choices, the following approximations are used.

When selecting one of the refutations of a positive example, a refutation that already has been selected is preferred. If no such refutation exists, a refutation that does not subsume a refutation of a negative example is preferred. The motivation for this heuristic is that it is presumably easier to exclude the refutation of a negative example if it is not included in a refutation of a positive example.

When selecting a (non-unit) clause to apply unfolding upon, the only refutations of negative examples that are considered are those for which all clauses are used in refutations of positive examples. A clause is preferred that appears first in one of these refutations, if no other clause in the same refutation is first in one of the other refutations. If no such clause exists, a non-unit clause that appears first in a refutation of a negative example is preferred. Finally, if there is no such clause, a non-unit clause that appears first in a refutation of a positive example is preferred. The motivation for this heuristic is that it is presumably easier to find a correct specialization by first excluding the refutations which only involves one clause that is first in a refutation of a negative example, than starting with refutations that contain several such clauses.

The leftmost (unfoldable) literal is always selected, and all clauses defining the target predicate that are not used as input clauses in any refutations of positive examples are removed.

In order to test the performance of the algorithm together with the above heuristics, an experiment is performed using the following program as input, where the predicate `target(X,Y,Z)` can be specialized into various list handling predicates:

```
target(X,Y,Z):- comp(X,Y,Z).
target(X1,Y1,Z1):-
        new_lists(X1,Y1,Z1,X2,Y2,Z2), target(X2,Y2,Z2).

comp(X,Y,Z):-
        eq_or_neq(X,Y), eq_or_neq(X,Z), eq_or_neq(Y,Z),
        list(X), list(Y), list(Z).

eq_or_neq(E,E).
eq_or_neq(E1,E2):- E1\==E2.
```

```
list([]).
list([_|_]).

new_lists([E|X],Y,[E|Z],X,Y,Z):- check(E,Y).
new_lists([E|X],Y,Z,X,Y,Z):- check(E,Y).

check(E,_).
check(E,X):- member(E,X).
check(E,X):- not_member(E,X).


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

not_member(X,[]).
not_member(X,[Y|L]):- X\==Y,not_member(X,L).
```

The examples used in this experiment are all instances of `target(X,Y,Z)`, where the variables are replaced by lists of length ≤ 2, containing the constants a, b and c (i.e. 2197 instances). When the examples are classified using the predicate `append(X,Y,Z)` (resulting in that 34 of the instances are positive), and given as input to SPECTRE II, the following program is produced in 54.6+6.8 seconds:[3]

```
target([],[],[]):-
        list([]),list([]).
target([A|B],[],[A|B]):-
        [A|B]\==[], []\==[A|B], list([A|B]).
target([],A,A):-
        []\==A, []\==A, list(A), list(A).
target([B|C],D,[B|A]):-
        check(B,D), target(C,D,A).
```

It can be seen that the resulting program is in fact equivalent to the definition of `append(X,Y,Z)`, although it contains a redundant clause and some redundant literals.

When the examples are classified using the predicate `intersection(X,Y,Z)` (resulting in that 169 of the instances are positive), and given as input to SPECTRE II, the following (correct) program is produced in 49.6+5.8 seconds:

[3] The algorithm was implemented in SICStus Prolog 2.1 on a Sun SPARCstation 5. The first number shows the time taken to find all refutations, and the second number shows the time taken to find a correct specialization, given the refutations.

```
target(A,A,A):-
        eq_or_neq(A,A), list(A), list(A), list(A).
target([],A,[]):-
        []\==A, A\==[], list(A), list([]).
target([A|B],[],[]) :-
        [A|B]\==[], [A|B]\==[], list([]).
target([B|C],D,[B|A]) :-
        member(B,D), target(C,D,A).
target([A|B],C,D):-
        not_member(A,C), target(B,C,D).
```

When the examples are classified using the predicate `difference(X,Y,Z)` (resulting in that 169 of the instances are positive), and given as input to SPECTRE II, the following (correct) program is produced in 50.8+8.3 seconds:

```
target([],[],[]):-
        list([]), list([]).
target([A|B],[A|B],[]):-
        [A|B]\==[], [A|B]\==[].
target([],A,[]):-
        []\==A, A\==[], list(A), list([]).
target([A|B],[],[A|B]):-
        [A|B]\==[], []\==[A|B], list([A|B]).
target([B|C],D,[B|A]):-
        not_member(B,D), target(C, D, A).
target([A|B],C,D):-
        member(A,C), target(B, C, D).
```

The following experiment illustrates how the computation time grows with the number of examples given as input to the algorithm. The entire example set is randomly split into two halves, where one half is used for training and the other for testing. The number of examples in the training sets that are given as input to the algorithm is varied, representing 1%, 5%, 10%, 20%, 30%, 40% and 50% of the entire example set, where the last subset corresponds to the entire set of training examples and a greater subset always includes a smaller. Each experiment is iterated 50 times and the mean time to find a correct specialization for the given examples is presented in Fig 1. It can be seen that the computation time grows linearly with the number of examples. In Fig 2, the accuracy of the produced hypothesis on the test examples is presented. The accuracy is above 99% for all three example sets, when using 20% of the examples as input to SPECTRE II.

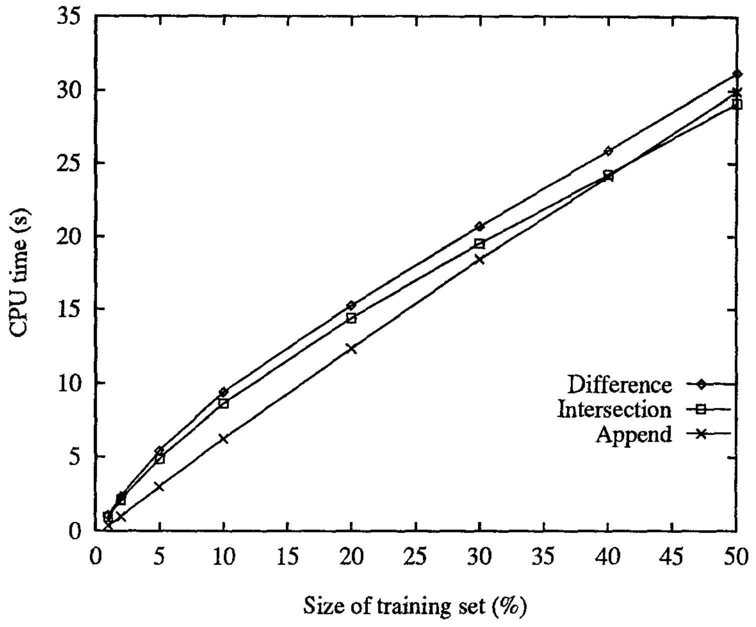

Fig. 1. Time taken for SPECTRE II to find correct specializations.

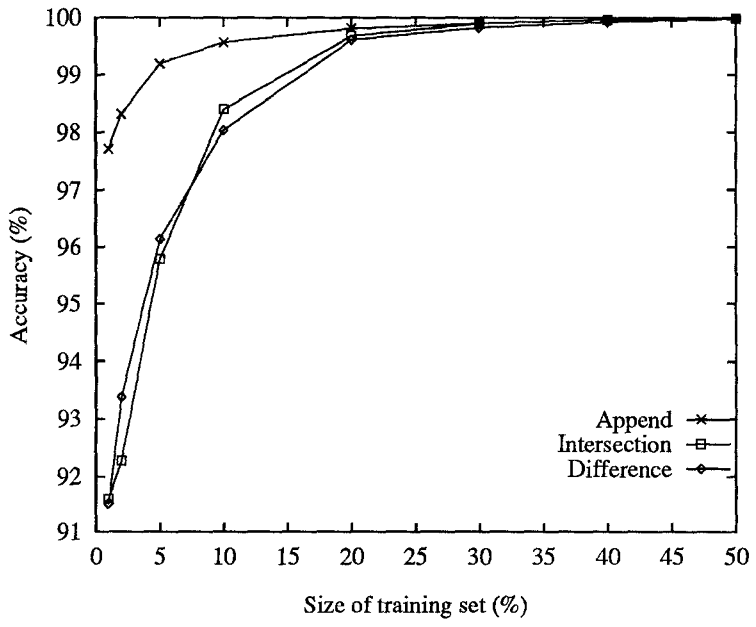

Fig. 2. Accuracy of the resulting specializations on the test examples.

6 Related Work

In this section, we discuss other specialization techniques, and in particular how they work when specializing recursive predicates.

In contrast to SPECTRE II, the techniques in [7, 1] attempt to minimally specialize logic programs. However, as pointed out in [16], the specializations produced by these techniques are not always minimal. That is the case when the specialized predicate appears in bodies of clauses, and this is true in particular when specializing a recursive predicate. For example, given the following program and the negative example `p(a,a)`:

```
p(X,X).
p(f(X),g(Y)):- p(X,Y).
```

Then the specialization produced by the technique in [1] will be:

```
p(X,X):- not(p1(X)).
p(f(X),g(Y)):- p(X,Y).
p1(a).
```

Note that not only is the negative example excluded, but also `p(f(a),g(a))`, `p(f(f(a)),g(g(a)))`, ..., and thus the resulting specialization is certainly not minimal. However, this problem can be solved by applying the same transformation technique as was presented in section three, namely define a new predicate to be equivalent to the target predicate and then replace each occurence of the original predicate in bodies of clauses with the appropriate instances of the new predicate before applying the specialization technique.

In [16] a technique for incremental specialization is presented, called MBR (Minimal Base Revision). It uses clause removal and addition of literals to prevent negative examples from being derived. The aim of this technique is not to specialize a program with respect to positive and negative examples, but to make a minimal revision of the program in the sense that a minimal set of clause applications is prevented from being used. This should be contrasted to SPECTRE II, which may produce specializations that are not minimal in this sense. Another difference is that the only clauses that are considered for being specialized by MBR are those appearing in refutations of negative examples, since this is sufficient when specializing a program with respect to negative examples only.

The only specialization operator that is used in [9, 15] is clause removal. These approaches are limited to removing clauses that appear in the original program that is to be specialized. As a consequence, these approaches will not be able to produce correct specializations when all input clauses in a refutation of a negative example are used in refutations of positive examples. However, the condition for when correct specializations can be obtained by clause removal can be relaxed significantly, if combined with program transformation, as shown by SPECTRE II.

In [13, 2, 12, 11, 10], 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]). It should be noted that literal addition is in general not sufficient to obtain correct specializations of logic programs. Consider again the program defining the predicate `odd(X)`:

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

The only way to exclude the negative example `odd(0)` by literal addition is to add a false literal to the body of the first clause. But the resulting specialization will not be correct if there are any positive examples, since these are excluded as well.

A number of previous specialization techniques use goal reduction to specialize clauses (ML-SMART [2], ANA-EBL [4], FOCL [11], GRENDEL [5], FOCL-FRONTIER [10]). One major difference between SPECTRE II and these techniques is the way in which the search for a specialization is performed. The algorithms ANA-EBL, FOCL, GRENDEL and FOCL-FRONTIER, like MIS [13] and FOIL [12], use covering methods for finding correct specializations, i.e. the resulting definition is found by repeatedly specializing an overly general definition and for each repetition adding a clause to the resulting definition. This contrasts to SPECTRE II and ML-SMART, which do not specialize the same definition more than once. The most important difference is however, that the previous techniques specialize only one predicate at a time in contrast to SPECTRE II, which may specialize multiple predicates. In section 4.2 it was shown that this capability can be crucial in order to obtain correct specialization w.r.t. multiple predicates.

7 Concluding Remarks

We have presented the algorithm SPECTRE II, which is based on the observation that in order to obtain correct specializations of recursive predicates it is not in general sufficient to specialize clauses that are used in refutations of negative examples, but it is sometimes necessary to specialize clauses that are used in refutations of positive examples only. The main properties of the algorithm are the ability to produce recursive specializations and to revise multiple predicates. In contrast to its predecessor SPECTRE, the algorithm does not require the examples to be instances of the same predicate.

The main problem for future research is to develop more sophisticated heuristics than what is used in the current implementation of the algorithm, i.e. heuristics for how to select which refutations of positive examples to consider and what literals to apply unfolding upon. Other directions for future research are to investigate how to handle cases when all refutations of a positive example have the same sequences of input clauses as refutations of negative examples, and when some positive examples are not included in the meaning of the original program.

Acknowledgements

This work has been supported by the European Community ESPRIT BRA 6020 ILP (Inductive Logic Programming). Thanks to the anonymous reviewers and Robert Engels for comments and suggestions for improvement on the article.

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. Boström H. and Idestam-Almquist P., "Specialization of Logic Programs by Pruning SLD-Trees", *Proceedings of the 4th International Workshop on Inductive Logic Programming,* volume 237 of *GMD-Studien, Gesellschaft für Mathematik und Datenverarbeitung MBH* (1994) 31–48
4. Cohen W. W., "The Generality of Overgenerality", *Machine Learning: Proceedings of the Eighth International Workshop,* Morgan Kaufmann (1991) 490–494
5. Cohen W. W., "Compiling Prior Knowledge Into an Explicit Bias", *Machine Learning: Proceedings of the Ninth International Workshop,* Morgan Kaufmann (1992) 102–110
6. Kanamori T. and Kawamura T., "Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation (II)", ICOT Technical Report TR-403, Japan (1988)
7. Ling C. X., "Non-Monotonic Specialization", *Proceedings of International Workshop on Inductive Logic Programming,* Portugal (1991) 59–68
8. Lloyd J. W., *Foundations of Logic Programming,* (2nd edition), Springer-Verlag (1987)
9. 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
10. 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
11. 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
12. Quinlan J. R., "Learning Logical Definitions from Relations", *Machine Learning* 5 (1990) 239–266
13. Shapiro E. Y., *Algorithmic Program Debugging,* MIT Press (1983)
14. 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
15. Wogulis J., "Revising Relational Domain Theories", *Machine Learning: Proceedings of the Eighth International Workshop,* Morgan Kaufmann (1991) 462–466
16. 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