Learning Relations: Basing Top-Down Methods on Inverse Resolution

F. Bergadano[1] and D. Gunetti[2]

[1]University of Catania, via A. Doria 6/A,
95100 Catania, Italy, bergadan@mathct.cineca.it

[2]University of Torino, corso Svizzera 185,
10149 Torino, Italy, gunetti@di.unito.it

Abstract

Top-down algorithms for relational learning specialize general rules until they are consistent, and are guided by heuristics of different kinds. In general, a correct solution is not guaranteed. By contrast, bottom-up methods are well formalized, usually within the framework of inverse resolution. Inverse resolution has also been used as an efficient tool for deductive reasoning, and here we prove that input refutations can be translated into inverse unit refutations. This result allows us to show that top-down learning methods can be also described by means of inverse resolution, yielding a unified theory of relational learning.

Keywords: Machine learning, Automated reasoning, Relational Learning.

1 Introduction

Recently, in the Machine Learning community there has been a growing interest on *relational learning algorithms*, which learn restricted first order formulas from positive and negative examples. Early work is well represented by Plotkin's study on least general generalizations [8] and Shapiro's Model Inference System [10]. The former is related to the problem of generalizing clauses, as a basis for the bottom-up induction of logic formulas: the least general generalization of a number of examples will serve as a compressed description. The latter is based on the top-down specialization of clauses, until a set of Horn formulas which is consistent with the available examples is produced. These works have guided much of the recent research. Plotkin's idea of least general generalizations of logic formulas has inspired later work on "inverse resolution" [7], while providing the basis for most bottom-up approaches to the induction of logic programs [6].

Shapiro's use of a refinement operator is a natural reference for recent top-down clause induction methods [9, 2].

Early and more recent methods must now be analyzed w.r.t. some criterion of inductive success. The most natural property one may require is that the learned description P behaves correctly on the given examples E:

Definition 1 *A description P is complete w.r.t. E, iff* $\forall e^{+} \in E \; P \vdash e^{+}$.

Definition 2 *A description P is consistent w.r.t. E, iff* $\forall e^{-} \in E \; P \not\vdash e^{-}$.

If we suppose that an inductive method accepts as input a set of examples and prior information (an inductive bias) in the form of a set $\mathcal{P}$ of allowed descriptions, the two properties defined below are desirable:

Definition 3 *An induction procedure M is correct iff whenever M terminates successfully and M(E,$\mathcal{P}$)=P, then P is complete and consistent w.r.t. E.*

Definition 4 *An induction procedure M is sufficient iff whenever a complete and consistent description w.r.t. E exists in $\mathcal{P}$, then M(E,$\mathcal{P}$) will output one such description.*

Completeness and consistency of the learned descriptions was considered important in previous Machine Learning research and is easily obtained for propositional and non-recursive relational rules. However, when we move to recursive clauses, or to the problem of learning multiple predicates, the above requirements are not always met. In fact, most systems generate candidate program clauses one at a time in a top-down fashion, and check them against the examples independently of one another. For instance, the clause "p(X) :- q(X,Y), p(Y)." is normally said to cover the example p(a) if there is a positive example q(a,b) of q such that p(b) is a positive example of p. Other clauses for p and q (e.g. those learned previously) are not used to try a derivation for q(a,b) and p(b). In other words, clauses are evaluated extensionally at the time of learning. However, when the final description has been learned, at the time of testing, it will be used intensionally and this may yield unexpected results, e.g. positive examples that were found to be covered may not be derived, while valid proofs for negative examples may become possible [1]. As a consequence, well known systems such as Foil [9] and Golem [6] are not correct nor sufficient [1]. Some systems do solve the problem while keeping the extensional evaluation of clauses by asking queries to the user, so that missing examples are provided and unexpected derivations cannot be found [10]. However, these systems are incremental, and added examples may require backtracking.

A consequence of the above considerations is that top-down methods, which evaluate clauses extensionally, are considered to be empirical and heuristic, without a strong formal basis. This is aggravated by the fact that most top-down systems do not explore the whole hypothesis space, but use statistical information in order to guide the search. This is usually contrasted with Bottom-up methods based on inverse resolution, which can proved to be theoretically well founded [6]. In this paper we show that it is possible to give a theoretical basis to extensional top-down learning methods by restating them in terms of a special kind of inverse resolution not employing inverse unifiers.

Here is the plan of the paper: in the second section we briefly review the extensional learning approach. In the third section we describe the basics of inverse resolution in theorem proving and clarify the relationship between input and linear resolution, as first pointed out by Chang. In the fourth section we use the results of section three to show how extensional methods can be rewritten in terms of theorem proving with inverse resolution.

2 Relational Learning Algorithms based on Extensionality

Many systems, such as Foil [9] and Golem [6] learn concepts described by means of Horn clauses. Clauses are evaluated extensionally, since in this way candidate clauses can be generated directly from the examples one at a time and independently of one another. The basic learning algorithm of such systems can be described as follows:

Let P be the target concept and pos_examples(P) and neg_examples(P) the given positive and negative examples of P (in the following, α and γ represent generic conjunction of literals).

```
Extensional top-down learning method:
while pos_examples(P) $\neq \emptyset$ do
    Generate one clause "P($\vec{X}$) :- $\alpha$($\vec{X}$,$\vec{Y}$)";
    pos_examples(P) $\leftarrow$ pos_examples(P) $-$ pos($\alpha$)

Generate one clause:
$\alpha \leftarrow$ true;
while pos($\alpha$) $\neq \emptyset$ do
    if neg($\alpha$) $= \emptyset$ then return(P($\vec{X}$) :- $\alpha$)
      else choose a predicate Q and its arguments Args;
           $\alpha \leftarrow \alpha \wedge$ Q(Args)
```

where pos(α) and neg(α) are the sets of positive and negative examples of P covered by P:-α, i.e. the examples P($\vec{a}$) such that T $\cup$ E $\cup$ P($\vec{X}$):-α $\vdash$ P($\vec{a}$). Where E is the set of given examples and T is a user-given Horn Theory. The presence of E means that some predicates (in particular the one representing the target concept) are not derived from T but immediately found among the given examples.

The choice of the literal Q(Args) to be added to the partial antecedent α of the clause being generated is guided by heuristic information. It might nevertheless be a wrong choice in some cases, in the sense that it causes the procedure

"Generate one clause" to fail by exiting the while loop without returning any clause. This problem can be remedied by making the choice of Q(Args) a backtracking point.
We illustrate the method on the task of learning the *append* relation:

pos(*append*) = {*append*([],[b],[b]), *append*([a],[b],[a,b]).}
neg(*append*) = {*append*([],[b],[]), *append*([a],[b],[b]).}

we also know that *append* depends on the following set of predicates, with their usual definition supplied (except for *append*, of course):

null, *head*, *tail*, *cons*, *assign*, *append*.

This is an important information, but obviously still very far away from the actual description that we want to learn: we need to associate variables to these predicates, and divide the obtained literals among the unknown number of clauses that will be necessary.

The algorithm starts to generate the first clause - the antecedent α is initially empty. We need to choose the first literal Q(Args) to be added to α. As we have left the heuristics unspecified, we will choose it so as to make the discussion short. Variables are taken from the clause head, or from a finite set of additional typed variables.

Let α=*assign*(Y,Z). A positive example is covered, but we cannot accept the clause
append(X,Y,Z) :- *assign*(Y,Z) as it is, because its body is true for the negative example *append*([],[b],[b]), so more literals need to be added.
Let α=*assign*(Y,Z) $\wedge$ *null*(X); the first example is covered and no wrong outputs can be computed. A clause is generated and the covered example *append*([],[b],[b]) is removed from examples(*append*).

We proceed to the generation of another clause; α is empty again. Suppose we have already generated α=*head*(X,H) $\wedge$ *tail*(X,T); the remaining positive example is covered, but again we have to specialize because α is true for the second negative example.
Let α = *head*(X,H) $\wedge$ *tail*(X,T) $\wedge$ *append*(T,Y,W); this clause again extensionally covers the remaining example. In fact, we have that *head*([a],a) and *tail*([a,],[]) are true, and *append*([],[b],[b]) is a given example. However, the second negative example is still covered (moreover, the output variable Z is not instantiated), and the procedure needs to be continued.
At the next step, suppose we add the literal *cons*(H,W,Z), obtaining, e.g.,
α = *head*(X,H) $\wedge$ *tail*(X,T) $\wedge$ *append*(T,Y,W) $\wedge$ *cons*(H,W,Z)
which covers all positive examples and none of the negative ones.
The final solution turns out to be:

append(X,Y,Z) :- *assign*(Y,Z), *null*(X).
append(X,Y,Z) :- *head*(X,H), *tail*(X,T), *append*(T,Y,W), *cons*(H,W,Z).

3 Inverse Refutations and the Relationship between Input and Unit Resolution

Inverse Resolution has been used as an effective tool for learning Horn Clauses from examples [7]. The basic idea is that a clause C_2 is "learned" from an example C and a clause C_1 given in the background knowledge if C is the resolvent of C_1 and C_2. In [4] we show that inverse resolution can also be the basis for efficient forms of deductive reasoning, with a procedure which we shall call *inverse refutation.* The idea is simply to invert the refutation process based on resolution, in order to go from the empty clause to the given clause set instead of vice versa. This results in a strongly guided refutation process, because it is based on the form of the given clauses. Intuitively, a clause is generated by means of inverse resolution only if it is a subset (proper or not) of a given clause. The process of inverse refutation ends when all clauses (or at least a minimally unsatisfiable subset of them) have been reconstructed, and if read in its turn in reverse order it appears just like a usual classical refutation of the given set. in the following, we assume familiarity with the basic concepts of resolution and theorem proving as in [5]. We remember that in unit deductions at least one of the parent clauses involved in a resolution step is a unit clause, while in input deductions at least one of the parent clauses is one of those given initially.
We illustrate the method on the following set S of clauses.

c1 = ACH, c2 = AD, c3 = ¬A, c4 = B¬C, c5 = ¬B, c6 = ¬H.

Just as resolution of two complementary unit clauses is the last step in a classical deduction, it is the first step of the inverse resolution deduction. Initially we have the empty clause; we open two branches and label them with two complementary unit clauses consisting of the first two complementary literals (say A and ¬A) found in the given set (we will build the inverse refutation from bottom to top, so it reads from top to bottom as a classical resolution refutation - see Fig. 1a, first step). Now we focus attention on the two unit clauses. If a refutation for S exists in which the last step resolves these two units, then there must exist an (inverse) deduction of each of them separately. Hence we have decomposed the problem of deriving the empty clause into the two independent subproblems of deriving the chosen unit clauses. Obviously, both of these subproblems must be solved, so they share an *and* relationship. On the other hand, we could have chosen different pairs of complementary literals to start the deduction, and a solution stemming from any of those choices is sufficient, so there we see an *or* relationship. Hence, our search for an inverse deduction will take the form of a typical and-or search tree.

Consider A. Since there are two clauses in the given set containing A (namely, c1 and c2), we open from A two new pairs of branches. The first pair corresponds to clause c1 and one of its branches is labeled AC (intuitively A with C added) because C is the next unexplored literal of c1. The other branch is labeled with ¬C, the unit clause built from the complementary of the literal added (Fig. 1a, second step). The second pair of branches corresponds to c2 and is labeled with AD and ¬D via a similar analysis. In general, corresponding to every given clause that contains the literal of the clause to be derived, there is a possible derivation. These derivations are or-related subproblems. Each of these subproblems corresponds to a choice of one such given clause and is, in turn, expressed as two and-related tasks: in one, a new literal from the given clause under consideration is added to the clause to be derived (thus this new vertex is labeled with a larger subset of the given clause chosen). The other is the problem of deriving the unit clause consisting of the complement of the added literal. So in our example, the two pairs of branches opened from A are two *or* tasks, and the branches in each pair are *and* tasks. In this example, to give a derivation of A we must demonstrate that there exists a derivation of the two clauses AC and ¬C, or that there exists a derivation of AD and ¬D.

Now again we must give attention separately to the two pairs of opened branches trying to build a complete (inverse) derivation of A. Let us concentrate only on the first pair of branches. On the right branch we are rebuilding the clause c1, so from the current clause AC (a subset of c1) we open two branches. One of them is labeled with AC to which we add H, the remaining literal of c1, and the other is labeled with its complementary ¬H (Fig. 1a, third step). But now the two new generated clauses belong to the given set and so there is nothing more to do with their subtasks. Now, only an *and* branch remains to be considered, the one labeled with ¬C, and we note that there is only one given clause containing it, so at its top we open a new pair of branches labeled with B¬C and ¬B respectively (Fig.1a, fourth step). We note that these two clauses belong to S, and because also clause ¬A is a given one, the inverse refutation is completed and all the other *or* branches opened while trying to derive A can be discarded. If read from top to bottom, the inverse refutation represents a classical unit derivation of the empty clause from the given set.

It should be noted that the strategy described here only builds unit refutations. See [4] for the complete strategy. In 1970 Chang proved an interesting relationship between unit and input resolution: a set of ground clauses S has a unit proof if and only if it has an input proof [5] (we recall that an input derivation is also a linear one). Here we clarify this relationship via inverse resolution, with the following theorems (Proofs can be found in [3]).

Theorem 1. For every input refutation of an unsatisfiable set S of propositional clauses there exists an inverse unit refutation for S where the same literals are introduced in the same order they are resolved in the input refutation.

The above relationship is much more easy to understand visually. Fig. 1b reports the input refutation corresponding to the inverse unit refutation of Fig. 1a. Equally numbered operations involve the same occurrences of the same literals (Fig. 1b must be read from top to bottom).

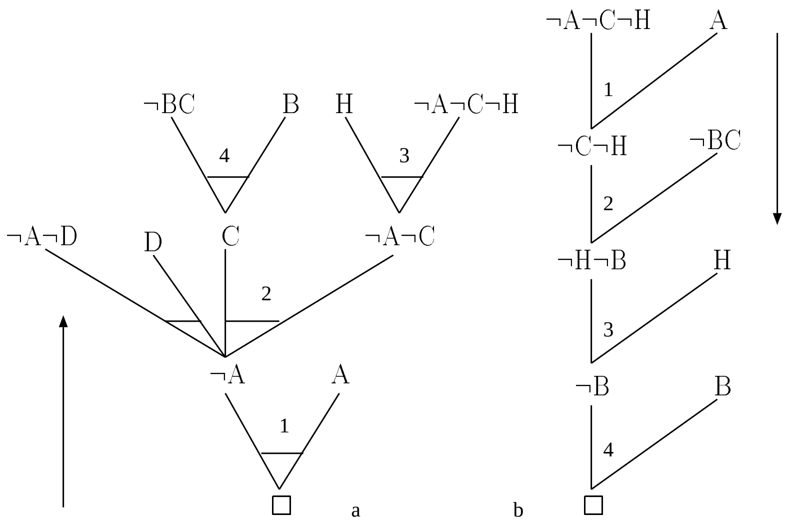


Figure 1: Fig. 1a and 1b. (Inverse) unit and input refutation for S.

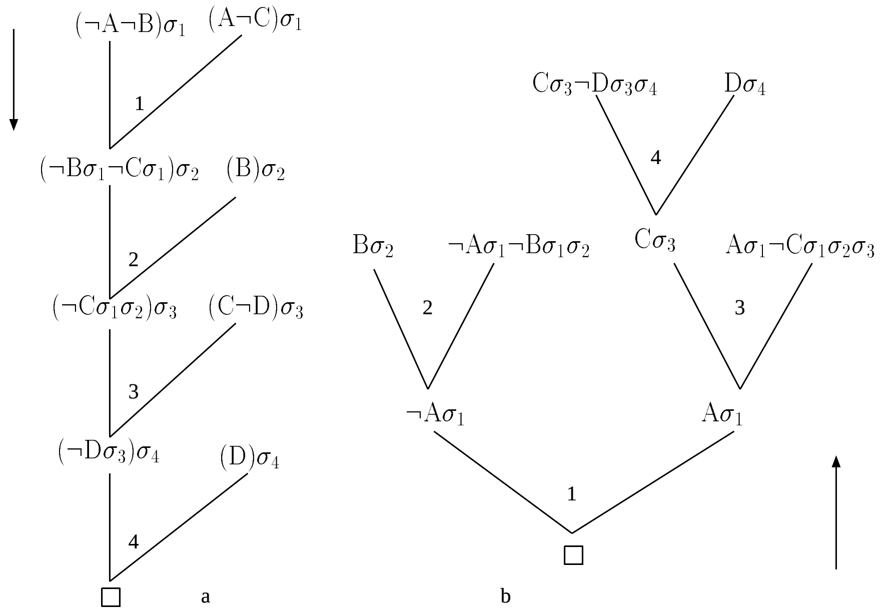
Extending inverse resolution to first order logic requires, in principle, the use of *inverse substitutions*, as in [7], whose computation can have exponential complexity. The next theorem shows that this is not the case if we limit ourselves to input deductions without factoring (in fact, an equivalent property for deductions using factoring can be easily obtained from the next theorem).

Definition 5 In our framework, we define *inverse resolution refutations* in first order logic as follows (for simplicity we always assume literals l and ¬l' to have disjoint variables. Let |l| denote the atom of literal l).
At the first step, if there exist in the given set of clauses two literals l and ¬l' and a substitution σ_1 such that $|l\sigma_1| = |\neg l'\sigma_1|$, then we can open from the empty clause two branches labeled respectively $|l\sigma_1|$ and $|\neg l'\sigma_1|$.
Suppose now that the first literal of a clause A has been introduced at the j-th step in an inverse refutation using substitution σ_j. Then, at the k-th step it is allowed to add a literal l to the subset of A built up to that point via inverse resolution using a complementary literal ¬l' if and only if $\exists\ \sigma_k$ such that $|l\sigma_j\sigma_{j+1}...\sigma_k| = |\neg l'\sigma_k|$, where σ_j,...,σ_{k-1} are the unifiers used between steps j and k-1.
The inverse refutation is completed when all branches are labeled with clauses from the given set, ignoring the introduced unifiers.

The above definition of inverse refutation in first order logic is justified by the following theorem:

Theorem 2. For every input refutation of an unsatisfiable set S of clauses there exists a inverse unit refutation of S where complementary literals are introduced in the same order they are resolved within the input refutation and where the same unifiers are involved.

Observe that, while in propositional calculus inverse deductions read from top to bottom appear to be ordinary deductions, in first order logic this is no longer true. In Fig. 2a an input refutation for an unsatisfiable set of clauses is shown, while Fig. 2b reports the corresponding inverse unit refutation. Note how the inverse refutation, if read from top to bottom, does *not* turn out to be an ordinary refutation (for simplicity, first order literals are represented with only capital letters, without reporting their terms. But two literals A and ¬A are considered to be complementary only if there exists a substitution σ such that $|A\sigma| = |\neg A\sigma|$).



$(\neg A \neg B)\sigma_1$
$(A \neg C)\sigma_1$
1
$(\neg B\sigma_1 \neg C\sigma_1)\sigma_2$
$(B)\sigma_2$
2
$(\neg C\sigma_1\sigma_2)\sigma_3$
$(C \neg D)\sigma_3$
3
$(\neg D\sigma_3)\sigma_4$
$(D)\sigma_4$
4
$\square$
a
$C\sigma_3 \neg D\sigma_3\sigma_4$
$D\sigma_4$
4
$B\sigma_2$
$\neg A\sigma_1 \neg B\sigma_1\sigma_2$
$C\sigma_3$
$A\sigma_1 \neg C\sigma_1\sigma_2\sigma_3$
2
3
$\neg A\sigma_1$
$A\sigma_1$
1
$\square$
b

Figure 2: Fig. 2a and 2b. An input refutation an the corresponding inverse unit refutation

4 Learning Horn Theories

An important subset of first order logic which admits input (and hence unit) refutation is the set of Horn clauses, and because inverse resolution builds clauses, theorem 2 seems to suggest a way to build (i.e. to learn) Horn theories from ground examples. If P(a,b) is a positive example of a concept P, then there exists an input refutation of P $\cup$ $\neg$P(a,b) with $\neg$P(a,b) as top clause. But, by theorem two, there exists also a corresponding inverse unit refutation of P $\cup$ $\neg$P(a,b) which, in fact, rebuilds the clauses of P (or, at least, those effectively involved in the refutation).

Now, suppose we do not have a Horn description of a concept P. We only know that P may depend on a given set of predicates, where every predicate can be defined by means of logical rules or with a set of positive and negative ground instances. Obviously, at least P is only defined by a set of positive and negative instances. Then, applying inverse resolution as in the previous section, we can build an inverse unit refutation starting with a positive example P(a,b) of P and where the leaves of the proof tree represent a possible (partial) description of P. We can stop the inverse refutation (in this case, the learning process) for that example when that partial description does not entail any of the given negative examples of P.

In this section we show that the learning process of section two can be restated in the above terms of clause construction via inverse resolution (in the following, the empty clause will be indicated with ":-", T will be the set of intensional definitions and E the given positive examples. C will be the clause we are currently learning).

As we have seen in section two, given a positive example P(a,b), where P is an inductive predicate (i.e. the concept to be learned) we start *guessing* the unit clause C = P(X,Y). In terms of inverse refutation, this means to start from the empty clause ":-", opening one branch labeled $\neg$P(a,b) and the other labeled Cσ_1 (where $\sigma_1 = \{$X/a, Y/b$\}$). Because obviously C derives negative examples of P, (in this case, ($\forall$ e^-) C $\cup$ T $\cup$ E $\vdash$ e^-) we must specialize (i.e. add literals to) its body.

At the next step, we choose a literal Q_1(Args) such that the body of C = P(X,Y) :- Q_1(Args) is extensionally evaluated to true on example P(a,b). Within inverse resolution, this means that there exists a substitution σ_2 such that Q_1(Args)$\sigma_1\sigma_2$ is a given positive example of Q_1 (if it is defined extensionally) or it is derivable from its definition (if it is defined by means of logical rules - in fact, in this case we have not to start an inverse derivation for Q_1(Args), because we already have a definition for Q_1).

In general, at the k+1-th step, we can add literal Q_k(Args) to the body of C if there exists a substitution σ_{k+1} such that Q_k(Args)$\sigma_1, ..., \sigma_{k+1}$ is a positive example of Q_k or it is derivable from its definition.

In extensional top-down learning methods we stop when C does not cover any of the negative examples, and still P(a,b) is covered. If no such C can be found, then backtracking occurs. Within inverse resolution this means that it must not be the case that C $\cup$ T $\cup$ E $\vdash e^-$ for any negative example e^-. Otherwise, alternative paths from the empty clause for example P(a,b) must be tried.

We clarify the above relationship by restating the learning task of *append* in terms of inverse resolution. Suppose we are given the following ground unit clauses, which are the positive examples of *append*:

e_1^+ = append([],[b],[b]), e_2^+ = append([a],[b],[a,b]),

and the two negated clauses

e_1^- = ¬append([],[b],[]), e_2^- = ¬append([a],[b],[b]),

which are negative examples of *append*. We are also given the following set L of literals, that make it possible to build a description of *append* (we assign variables to literals in order to make the discussion short):

{null(X), head(X,H), tail(X,T), cons(H,W,Z), assign(Y,Z), append(T,Y,W)}

where predicates, except for *append*, are defined as follows (call T this set of definitions):

```
null([],[]).
head([A|_],A).
tail([_|B],B).
cons(C,D,[C|D]).
assign(E,E).
```

Now we start the learning task by looking for a clause C such that C $\cup$ T $\cup$ $\{e_1^+\} \cup \{\neg e_2^+\}$ is unsatisfiable (again for brevity, we do not consider alternative paths).

We start from the empty clause and generate two branches, one labeled ¬append([a],[b],[a,b])σ_1 and the other one labeled C = append(X,Y,Z)σ_1, with σ_1 = {X/[a], Y/[b], Z/[a,b]}. Because ($\forall\ e^-$) C $\cup$ T $\cup$ $\{e_1^+\} \vdash e^-$, we must continue the inverse derivation.

At the next step, suppose we select from L the literal head(X,H). From C we open two branches, one labeled head([A|_],A)σ_2 and the other one labeled

C = append(X,Y,Z)σ_1 :- head(X,H)$\sigma_1\sigma_2$
with σ_2 = {A/a, H/a}.

At the third step, tail(X,T) is selected, and from C we open one branch labeled tail([_|B],B)σ_3 and the other labeled:
C = append(X,Y,Z)σ_1 :- head(X,H)$\sigma_1\sigma_2$, tail(X,T)$\sigma_1\sigma_2\sigma_3$

with σ_3 = {B/[], T/[]}.

At the fourth step append(T,Y,W) is selected, and from C we open a branch labeled append([],[b],[b])σ_4 and the other one labeled

C = append(X,Y,Z)σ_1 :- head(X,H)$\sigma_1\sigma_2$, tail(X,T)$\sigma_1\sigma_2\sigma_3$,
append(T,Y,W) $\sigma_1\sigma_2\sigma_3\sigma_4$
with σ_4 = {T/[], W/[b]}.

Finally, cons(H,W,Z) is selected, and from C we open one branch labeled: cons(C,D,[C|D])σ_5 and the other one labeled:

C = append(X,Y,Z)σ_1 :- head(X,H)$\sigma_1\sigma_2$, tail(X,T)$\sigma_1\sigma_2\sigma_3$,
append(T,Y,W)$\sigma_1\sigma_2\sigma_3\sigma_4$, cons(H,W,Z) $\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$.
with σ_5 = {C/a, D/[b]}.

At this point it is the case that $\forall\ e^-$ C $\cup$ T $\cup$ $\{e_1^+\} \not\vdash e^-$. It can be easily verified that there exists an input refutation of C $\cup$ T $\cup$ $\{e_1^+\} \cup \{\neg e_2^+\}$ where at the first step C is resolved against $\neg e_2^+$ and where the substitutions σ_1,..., σ_5 are employed, in that order.
By removing substitutions introduced along the inverse refutation,

C= *append*(X,Y,Z) :- *head*(X,H), *tail*(X,T), *append*(T,Y,W), *cons*(H,W,Z)

represents the first learned clause for *append*. It should be noted that the used substitutions correspond to the assignment of values to variables performed in the extensional evaluation of the body of C, in section two.
A similar procedure could then be followed to learn the non-recursive clause of *append*.

5 Conclusion

We have argued that theorem proving with inverse resolution represents a theoretical basis for top-down extensional learning methods. Our result can have many interesting consequences.

First, the method suggests how to query the user for missing examples, by possibly asking for the truth values of the unit clauses used in every inverse resolution step. Obviously, the number of queries depends on the size of the hypothesis space and on the chosen variabilization for the various predicates. However, it has been shown [10, 2] that queries provide the basis for inductive methods which are efficient if appropriate syntactic restrictions are adopted.

Second, such methods can also be proved to be correct and sufficient as defined in the introduction [2], and the present paper can also be seen as an alternative argument for proving the same results. In fact, if a correct Horn theory exists in the hypothesis space, an input refutation from all the given and queried examples is possible. But then these clauses may be learned by means of some inverse unit refutation. The argument of section 4 will then imply that top-down learning of a complete and consistent program is also possible.

Third, some computational problems of inverse resolution, as developed in [7], are avoided because inverse substitutions need not be computed.

Finally, our results suggest that one initial given positive example is sufficient to learn all the clauses necessary to derive it (if they exist in the hypothesis space). This means that, most of the time, one well chosen example is sufficient to learn a complete description of a concept. This should be contrasted with classical extensional methods, where a lot of examples are required.

Acknowledgments: This work was in part supported by BRA ESPRIT III Project 6020 on Inductive Logic Programming.

References

[1] F. Bergadano. Inductive Data Base Relations. *To appear in IEEE Trans. on Data and Knowledge Engineering*, 1993.

[2] F. Bergadano and D. Gunetti. An Interactive System to Learn Functional Logic Programs. In *Proc. 13th Int. Joint Conf. on Artificial Intelligence*, Chambery, France, 1993. Morgan Kaufmann.

[3] F. Bergadano and D. Gunetti. Unifying top-down and inverse resolution approaches to inductive logic programming. Technical report, 1993. (Tech. Rep. 93.3.28, CS Dept., Univ. of Torino.

[4] D. Gunetti. Efficient Proofs in Propositional Calculus with Inverse Resolution. In P. Dewilde and J. Vanderwalle, editors, *Proc. of the CompEuro*, The Hague, 1992. IEEE Comp. Soc. Press.

[5] D. W. Loveland. *Automated Theorem Proving: A Logical Basis.* North Holland, Amsterdam, 1978.

[6] S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.

[7] S. Muggleton and W. Buntine. Machine Invention of First Order Predicates by Inverting Resolution. In *Proc. of the Fifth Int. Conf. on Machine Learning*, pages 339–352, Ann Arbor, MI, 1988. Morgan Kaufmann.

[8] G. Plotkin. A Note on Inductive Generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh Univ. Press, 1970.

[9] J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266, 1990.

[10] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.