A System for Abductive Learning of Logic Programs

Evelina Lamma[1], Paola Mello[2], Michela Milano[1], Fabrizio Riguzzi[1]

[1] DEIS, Università di Bologna,
Viale Risorgimento 2, I-40136 Bologna, Italy,
{elamma,mmilano,friguzzi}@deis.unibo.it
[2] Dip. di Ingegneria, Università di Ferrara,
Via Saragat 1, I-44100 Ferrara, Italy
pmello@ing.unife.it

Abstract. We present the system LAP (Learning Abductive Programs) that is able to learn abductive logic programs from examples and from a background abductive theory. A new type of induction problem has been defined as an extension of the Inductive Logic Programming framework. In the new problem definition, both the background and the target theories are abductive logic programs and abductive derivability is used as the coverage relation.
LAP is based on the basic top-down ILP algorithm that has been suitably extended. In particular, coverage of examples is tested by using the abductive proof procedure defined by Kakas and Mancarella [24]. Assumptions can be made in order to cover positive examples and to avoid the coverage of negative ones, and these assumptions can be used as new training data. LAP can be applied for learning in the presence of incomplete knowledge and for learning exceptions to classification rules.

Keywords: Abduction, Learning.

1 Introduction

Abductive Logic Programming (ALP) has been recognized as a powerful knowledge representation tool [23]. Abduction [22, 36] is generally understood as reasoning from effects to causes or explanations. Given a theory T and a formula G, the goal of abduction is to find a set of atoms Δ (explanation) that, together with T, entails G and that is consistent with a set of integrity constraints IC. The atoms in Δ are *abduced*: they are assumed true in order to prove the goal. Abduction is specially useful to reason in domains where we have to infer causes from effects, such as diagnostic problems [3]. But ALP has many other applications [23]: high level vision, natural language understanding, planning, knowledge assimilation and default reasoning. Therefore, it is desirable to be able to automatically produce a general representation of a domain starting from specific knowledge about single instances. This problem, in the case of standard Logic Programming, has been deeply studied in Inductive Logic Programming (ILP)

[7], the research area covering the intersection of Machine Learning [33] and Logic Programming. Its aim is to devise systems that are able to learn logic programs from examples and from a background knowledge. Recently, in this research area, a number of works have begun to appear on the problem of learning non-monotonic logic programs [4, 16, 8, 32].

Particular attention has been given to the problem of learning abductive logic programs [21, 26, 29, 30, 27] and, more generally, to the relation existing between abduction and induction and how they can integrate and complement each other [15, 17, 2]. Our work addresses this topic as well. The approach for learning abductive logic programs that we present in this paper is doubly useful. On one side, we can learn abductive theories for the application domains mentioned above. For example, we can learn default theories: in Section 5.1 we show an example in which we learn exceptions to classification rules. On the other side, we can learn theories in domains in which there is incomplete knowledge. This is a very frequent case in practice, because very often the data available is incomplete and/or noisy. In this case, abduction helps induction by allowing to make assumptions about unknown facts, as it is shown in the example in Section 5.2. In [29] we defined a new learning problem called Abductive Learning Problem. In this new framework we generate an abductive logic program from an abductive background knowledge and from a set of positive and negative examples of the concepts to be learned. Moreover, abductive derivability is used as the example coverage relation instead of Prolog derivability as in ILP.

We present the system LAP (Learning Abductive Programs) that solves this new learning problem. The system is based on the theoretical work developed in [21, 29] and it is an extension of a basic top-down algorithm adopted in ILP [7]. In the extended algorithm, the proof procedure defined in [24] for abductive logic programs is used for testing the coverage of examples in substitution of the deductive proof procedure of logic programming. Moreover, the abduced literals can be used as new training data for learning definitions for the abducible predicates.

The paper is organized as follows: in Section 2 we recall the main concepts of Abductive Logic Programming, Inductive Logic Programming, and the definition of the abductive learning framework. Section 3 presents the learning algorithm while its properties are reported in Section 4. In Section 5 we apply LAP to the problem of learning exceptions to rules and learning from incomplete knowledge. Related works are discussed in Section 6. Section 7 concludes and presents directions for future works.

2 Abductive and Inductive Logic Programming

2.1 Abductive Logic Programming

An *abductive logic program* is a triple $\langle P, A, IC \rangle$ where:

- P is a normal logic program;
- A is a set of *abducible predicates*;

– IC is a set of integrity constraints in the form of denials, i.e.:
$\leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_{m+n}$.

Abducible predicates (or simply abducibles) are the predicates about which assumptions (or abductions) can be made. These predicates carry all the incompleteness of the domain, they can have a partial definition or no definition at all, while all other predicates have a complete definition.

Negation as Failure is replaced, in ALP, by Negation by Default and is obtained by transforming the program into its *positive version*: each negative literal $not\ p(\boldsymbol{t})$, where $\boldsymbol{t}$ is a tuple of terms, is replaced by a literal $not\_p(\boldsymbol{t})$, where not_p is a new predicate symbol. Moreover, for each predicate symbol p in the program, a new predicate symbol not_p is added to the set A and the integrity constraint $\leftarrow p(\boldsymbol{X}), not\_p(\boldsymbol{X})$ is added to IC, where $\boldsymbol{X}$ is a tuple of variables. Atoms of the form $not\_p(\boldsymbol{t})$ are called *default atoms*. In the following, we will always consider the positive version of programs. This allows us to abduce either the truth or the falsity of atoms.

Given an abductive theory $AT = \langle P, A, IC \rangle$ and a formula G, the goal of abduction is to find a (possibly minimal) set of ground atoms Δ (*abductive explanation*) of predicates in A which, together with P, entails G, i.e., $P \cup \Delta \models G$. It is also required that the program $P \cup \Delta$ be consistent with respect to IC, i.e. $P \cup \Delta \models IC$. When there exists an abductive explanation for G in AT, we say that AT *abductively entails* G and we write $AT \models_A G$.

As model-theoretic semantics for ALP, we adopt the *abductive model* semantics defined in [9]. We do not want to enter into the details of the definition, we will just give the following proposition which will be useful throughout the paper.

We indicate with $\mathcal{L}^{\mathcal{A}}$ the set of all atoms built from the predicates of A (called *abducible atoms*), including also default atoms.

Proposition 1. *Given an abductive model M for the abductive program $AT = \langle P, A, IC \rangle$, there exists a set of atoms $H \subseteq \mathcal{L}^{\mathcal{A}}$ such that M is the least Herbrand model of $P \cup H$.*

Proof. Straightforward from the definition of abductive model (definition 5.7 in [9]).

In [24] a proof procedure for abductive logic programs has been defined. This procedure starts from a goal and a set of initial assumptions Δ_i and results in a set of consistent hypotheses (abduced literals) Δ_o such that $\Delta_o \supseteq \Delta_i$ and Δ_o together with the program P allow deriving the goal. The proof procedure uses the notion of *abductive* and *consistency derivations*. Intuitively, an abductive derivation is the standard Logic Programming derivation suitably extended in order to consider abducibles. As soon as an abducible atom δ is encountered, it is added to the current set of hypotheses, and it must be proved that any integrity constraint containing δ is satisfied. For this purpose, a consistency derivation for δ is started. Since the constraints are denials only (i.e., goals), this corresponds to proving that every such goal fails. Therefore, δ is removed from

all the constraints containing it, and we prove that all the resulting goals fail. In this consistency derivation, when an abducible is encountered, an abductive derivation for its complement is started in order to prove the abducible's failure, so that the initial constraint is satisfied. When the procedure succeeds for the goal G and the initial set of assumptions Δ_i, producing as output the set of assumptions Δ_o, we say that T *abductively derives* G or that G is *abductively derivable* from T and we write $T \vdash_{\Delta_i}^{\Delta_o} G$.

In [9] it has been proved that the proof procedure is *sound* and *weakly complete* with respect to the abductive model semantics defined in [9] under a number of restrictions. We will present these results in detail in Section 4.

2.2 Inductive Logic Programming

The ILP problem can be defined as [6]:
Given:

- a set $\mathcal{P}$ of possible programs
- a set E^+ of positive examples
- a set E^- of negative examples
- a logic program B (*background knowledge*)

Find:

- a logic program $P \in \mathcal{P}$ such that
  - $\forall e^+ \in E^+$, $B \cup P \vdash e^+$ (*completeness*)
  - $\forall e^- \in E^-$, $B \cup P \nvdash e^-$ (*consistency*).

Let us introduce some terminology. The program P that we want to learn is the *target program* and the predicates which are defined in it are *target predicates*. The sets E^+ and E^- are called *training sets* and contain ground atoms for the target predicates. The program B is called *background knowledge* and contains the definitions of the predicates that are already known. We say that the program P *covers* an example e if $P \cup B \vdash e$[1], i.e. if the theory "explains" the example. Therefore the conditions that the program P must satisfy in order to be a solution to the ILP problem can be expressed as "P must cover all positive examples and must not cover any negative example". A theory that covers all positive examples is said to be *complete* while a theory that does not cover any negative example is said to be *consistent*. The set $\mathcal{P}$ is called the *hypothesis space*. The importance of this set lies in the fact that it defines the search space of the ILP system. In order to be able to effectively learn a program, this space must be restricted as much as possible. If the space is too big, the search could result infeasible.

The *language bias* (or simply *bias* in this paper) is a description of the hypothesis space. Many formalisms have been introduced in order to describe this space [7], we will consider only a very simple bias in the form of a set of literals which are allowed in the body of clauses for target predicates.

[1] In the ILP literature, the derivability relation is often used instead of entailment because real systems adopt the Prolog interpreter for testing the coverage of examples, that is not sound nor complete.

```
Initialize H := ∅
repeat (Covering loop)
    Generate one clause c
    Remove from E+ the e+ covered by c
    Add c to H
until E+ = ∅ (Sufficiency stopping criterion)

Generate one clause c:
Select a predicate p that must be learned
Initialize c to be p(X̄) ← .
repeat (Specialization loop)
    Select a literal L from the language bias
    Add L to the body of c
    if c does not cover any positive example
        then backtrack to different choices for L
until c does not cover any negative example (Necessity stopping criterion)
return c
(or fail if backtracking exhausts all choices for L)
```

Fig. 1. Basic top-down ILP algorithm

There are two broad categories of ILP learning methods: *bottom-up* methods and *top-down* methods. In bottom-up methods clauses in P are generated by starting with a clause that covers one or more positive examples and no negative example, and by generalizing it as much as possible without covering any negative example. In top-down methods clauses in P are constructed starting with a general clause that covers all positive and negative examples and by specializing it until it does no longer cover any negative example while still covering at least one positive. In this paper, we concentrate on top-down methods. A basic top-down inductive algorithm [7, 31] learns programs by generating clauses one after the other. A clause is generated by starting with an empty body and iteratively adding literals to the body. The basic inductive algorithm, adapted from [7] and [31], is sketched in Figure 1.

2.3 The New Learning Framework

We consider a new definition of the learning problem where both the background and target theory are abductive theories and the notion of deductive coverage above is replaced by abductive coverage.

Let us first define the *correctness* of an abductive logic program T with respect to the training set E^+, E^-. This notion replaces those of completeness and consistency for logic programs.

Definition 1 (Correctness). *An abductive logic program T is* correct, *with respect to E^+ and E^-, iff there exists $\Delta \subseteq \mathcal{L}^{\mathcal{A}}$ such that*

$$T \vdash^{\Delta}_{\emptyset} E^+, not\_E^-$$

where $not\_E^- = \{not\_e^- | e^- \in E^-\}$ *and* E^+, not_E^- *stands for the conjunction of each atom in* E^+ *and* not_E^-

Definition 2 (Abductive Learning Problem).
Given:

- *a set* $\mathcal{T}$ *of possible abductive logic programs*
- *a set of positive examples* E^+
- *a set of negative examples* E^-
- *an abductive program* $T = \langle P, A, IC \rangle$ *as background theory*

Find:

A new abductive program $T' = \langle P \cup P', A, IC \rangle$ *such that* $T' \in \mathcal{T}$ *and* T' *is correct wrt* E^+ *and* E^-.

We say that a positive example e^+ is *covered* if $T \vdash^{\Delta}_{\emptyset} e^+$. We say that a negative example e^- is *not covered* (or *ruled out*) if $T \vdash^{\Delta}_{\emptyset} not\_e^-$. By employing the abductive proof procedure for the coverage of examples, we allow the system to make assumptions in order to cover positive examples and to avoid the coverage of negative examples. In this way, the system is able to complete a possibly incomplete background knowledge. Integrity constraints give some confidence in the correctness of the assumptions made.

Differently from the ILP problem, we require the conjunction of examples, instead of each example singularly, to be derivable. In this way we ensure that the abductive explanations for different examples are consistent with each other.

The abductive program that is learned can contain new rules (possibly containing abducibles in the body), but not new abducible predicates and new integrity constraints.

3 An algorithm for Learning Abductive Logic Programs

In this section, we present the system LAP that is able to learn abductive logic programs according to definition 2. The algorithm is obtained from the basic top-down ILP algorithm (Figure 1), by adopting the abductive proof procedure, instead of the Prolog proof procedure, for testing the coverage of examples.

As the basic inductive algorithm, LAP is constituted by two nested loops: the covering loop (Figure 2) and the specialization loop (Figure 3). At each iteration of the covering loop, a new clause is generated such that it covers at least one positive example and no negative one. The positive examples covered by the rule are removed from the training set and a new iteration of the covering loop is started. The algorithm ends when the positive training set becomes empty. The new clause is generated in the specialization loop: we start with a clause with an empty body, and we add literals to the body until the clause does not cover any negative example while still covering at least one positive. The basic top-down algorithm is extended in the following respects.

procedure LAP(
inputs : E^+, E^- : training sets,
$AT = \langle T, A, IC \rangle$: background abductive theory,
outputs : H : learned theory, Δ : abduced literals)

$H := \emptyset$
$\Delta := \emptyset$
repeat
GenerateRule(in: AT, E^+, E^-, H, Δ; out: $Rule, E^+_{Rule}, \Delta_{Rule}$)
Add to E^+ all the positive literals of target predicates in Δ_{Rule}
Add to E^- all the atoms corresponding to
negative literals of target predicates in Δ_{Rule}
$E^+ := E^+ - E^+_{Rule}$
$H := H \cup \{Rule\}$
$\Delta := \Delta \cup \Delta_{Rule}$
until $E^+ = \emptyset$ (Sufficiency stopping criterion)
output H

Fig. 2. The covering loop

procedure GenerateRule(
inputs : AT, E^+, E^-, H, Δ
outputs : $Rule$: rule,
E^+_{Rule} : positive examples covered by $Rule$,
Δ_{Rule} : abduced literals

Select a predicate to be learned p
Let $Rule = p(X) \leftarrow true$.
repeat (specialization loop)
select a literal L from the language bias
add L to the body of $Rule$
TestCoverage(in: $Rule, AT, H, E^+, E^-, \Delta$,
out: $E^+_{Rule}, E^-_{Rule}, \Delta_{Rule}$)
if $E^+_{Rule} = \emptyset$
backtrack to a different choice for L
until $E^-_{Rule} = \emptyset$ (Necessity stopping criterion)
output $Rule, E^+_{Rule}, \Delta_{Rule}$

Fig. 3. The specialization loop

procedure TestCoverage(
inputs : $Rule, AT, H, E^+, E^-, \Delta$
outputs: E^+_{Rule}, E^-_{Rule}: examples covered by $Rule$
Δ_{Rule} : new set of abduced literals

$E^+_{Rule} = E^-_{Rule} = \emptyset$
$\Delta_{in} = \Delta$
for each $e^+ \in E^+$ **do**
if AbductiveDerivation$(\leftarrow e^+, \langle T \cup H \cup \{Rule\}, A, IC\rangle, \Delta_{in}, \Delta_{out})$
succeeds **then** Add e^+ to E^+_{Rule}; $\Delta_{in} = \Delta_{out}$
endfor
for each $e^- \in E^-$ **do**
if AbductiveDerivation$(\leftarrow not_e^-, \langle T \cup H \cup \{Rule\}, A, IC\rangle, \Delta_{in}, \Delta_{out})$
succeeds **then** $\Delta_{in} = \Delta_{out}$
else Add e^- to E^-_{Rule}
endfor
$\Delta_{Rule} = \Delta_{out} - \Delta$
output $E^+_{Rule}, E^-_{Rule}, \Delta_{Rule}$

Fig. 4. Coverage testing

First, in order to determine the positive examples E^+_{Rule} covered by the generated rule $Rule$ (procedure TestCoverage in Figure 4), an abductive derivation is started for each positive example. This derivation results in a (possibly empty) set of abduced literals. We give as input to the abductive procedure also the set of literals abduced in the derivations of previous examples. In this way, we ensure that the assumptions made during the derivation of the current example are consistent with the assumptions for other examples.

Second, in order to check that no negative example is covered ($E^-_{Rule} = \emptyset$ in Figure 3) by the generated rule $Rule$, an abductive derivation is started for the default negation of each negative example ($\leftarrow not\_e^-$). Also in this case, each derivation starts from the set of abducibles previously assumed. The set of abducibles is initialized to the empty set at the beginning of the computation, and is gradually extended as it is passed on from derivation to derivation. This is done as well across different clauses.

Third, after the generation of each clause, the literals of target predicates that have been abduced are added to the training set, so that they become new training examples. For each positive abduced literal of the form $abd(\boldsymbol{c}^+)$ where $\boldsymbol{c}^+$ is a tuple of constants, the new positive example $abd(\boldsymbol{c}^+)$ is added to E^+ set, while for each negative literal of the form $not\_abd(\boldsymbol{c}^-)$ the negative example $abd(\boldsymbol{c}^-)$ is added to E^-.

In order to be able to learn exceptions to rules, we include a number of predicates of the form not_abnorm_i/n in the bias of each target predicate of the form p/n. Moreover, $abnorm_i/n$ and not_abnorm_i/n are added to the set of abducible predicates and the constraint

$\leftarrow abnorm_i(\boldsymbol{X}), not\_abnorm_i(\boldsymbol{X})$.

is added to the background knowledge. In this way, when the current partial rule in the specialization loop still covers some negative examples and no other literal can be added that would make it consistent, the rule is specialized by adding the literal $not\_abnorm_i(\boldsymbol{X})$ to its body. Negative examples previously covered are ruled out by abducing for them facts of the form $abnorm_i(\boldsymbol{c}^-)$, while positive examples will be covered by abducing the facts $not\_abnorm_i(\boldsymbol{c}^+)$ and these facts are added to the training set.

We are now able to learn rules for $abnorm_i/n$, thus resulting in a definition for the exceptions to the current rule. For this purpose, predicates $abnorm_i/n$ are considered as target predicates, and we define a bias for them. Since we may have exceptions to exceptions, we may also include a number of literals of the form $not\_abnorm_j(\boldsymbol{X})$ in the bias for $abnorm_i/n$.

The system has been implemented in Prolog using Sicstus Prolog 3#5.

4 Properties of the algorithm

LAP is sound, under some restrictions, but not complete. In this section we give a proof of its soundness, and we point out the reasons of incompleteness.

Let us first adapt the definitions of soundness and completeness for an inductive inference machine, as given by [7], to the new problem definition. We will call Abductive Inductive Inference Machine (AIIM) an algorithm that solves the Abductive Learning Problem. If M is an AIIM, we write $M(\mathcal{T}, E^+, E^-, B) = T$ to indicate that, given the hypothesis space $\mathcal{T}$, positive and negative examples E^+ and E^-, and a background knowledge B, the machine outputs a program T. We write $M(\mathcal{T}, E^+, E^-, B) = \bot$ when M does not produce any output.

With respect to the abductive learning problem (definition 2), the definitions of soundness and completeness are:

Definition 3 (Soundness). *An AIIM M is sound iff if $M(\mathcal{T}, E^+, E^-, B) = T$, then $T \in \mathcal{T}$ and T is correct with respect to E^+ and E^-.*

Definition 4 (Completeness). *An AIIM M is complete iff if $M(\mathcal{T}, E^+, E^-, B) = \bot$, then there is no $T \in \mathcal{T}$ that is correct with respect to E^+ and E^-.*

The proof of LAP soundness is based on the theorems of soundness and weak completeness of the abductive proof procedure given in [9]. We will first present the results of soundness and completeness for the proof procedure and then we will prove the soundness of our algorithm.

The theorems of soundness and weak completeness (theorems 7.3 and 7.4 in [9]) have been extended by considering the goal to be proved as a conjunction of abducible and non-abducible atoms (instead of a single non-abducible atom) and by considering an initial set of assumptions Δ_i. The proofs are straightforward, given the original theorems.

Theorem 1 (Soundness). *Let us consider an abductive logic program T. Let L be a conjunction of atoms. If $T \vdash^{\Delta_o}_{\Delta_i} L$, then there exists an abductive model M of T such that $M \models L$ and $\Delta_o \subseteq M \cap \mathcal{L}^{\mathcal{A}}$.*

Theorem 2 (Weak completeness). *Let us consider an abductive logic program T. Let L be a conjunction of atoms. Suppose that every selection of rules in the proof procedure for L terminates with either success or failure. If there exists an abductive model M of T such that $M \models L$, then there exists a selection of rules such that the derivation procedure for L succeeds in T returning Δ, where $\Delta \subseteq M \cap \mathcal{L}^{\mathcal{A}}$.*

We need as well the following lemma.

Lemma 1. *Let us consider an abductive logic program $T = \langle P, A, I \rangle$. Let L be a conjunction of atoms. If $T \vdash_{\emptyset}^{\Delta} L$ then $lhm(P \cup \Delta) \models L$, where $lhm(P \cup \Delta)$ is the least Herbrand model of $P \cup \Delta$.*

Proof. Follows directly from theorem 5 in [18].

The theorems of soundness and weak completeness for the abductive proof procedure are true under a number of assumptions:

- the abductive logic program must be ground
- the abducibles must not have a definition in the program
- the integrity constraints are denials with at least one abducible in each constraint.

Moreover, the weak completeness theorem is limited by the assumption that the proof procedure for L always terminates.

The soundness of LAP is limited as well by these assumptions. However, they do not severely restrict the generality of the system. In fact, the requirement that the program is ground can be met for programs with no function symbols. In this case the Herbrand universe is finite and we obtain a finite ground program from a non-ground one by grounding in all possible ways the rules and constraints in the program. This restriction is also assumed in many ILP systems (such as FOIL [37], RUTH [1], [11]).

The restriction on the absence of a (partial) definition for the abducible does not reduce the generality of the results since, when abducible predicates have definitions in T, we can apply a transformation to T so that the resulting program T' has no definition for abducible predicates. This is done by introducing an auxiliary predicate δ_a/n for each abducible predicate a/n and by adding the clause:

$$a(\boldsymbol{x}) \leftarrow \delta_a(\boldsymbol{x}).$$

The predicate a/n is no longer abducible, whereas δ_a/n is now abducible. In this way, we are able to deal as well with partial definitions for abducible predicates, and this is particularly important when learning from incomplete data, because the typical situation is exactly to have a partial definition for some predicates, as will be shown in Section 5.2.

The requirement that each integrity constraint contains an abducible literal is not restrictive because we use constraints only for limiting assumptions and therefore a constraint without an abducible literal would be useless.

The most restrictive requirement is the one on the termination of the proof procedure. However, it can be proved that the procedure always terminates for *call-consistent* programs, i.e. if no predicate depends on itself through an odd number of negative recursive calls (e.g., $p \leftarrow not\_p$).

We need as well the following theorem. It expresses a restricted form of monotonicity that holds for abductive logic programs.

Theorem 3. *Let $T = \langle P, A, I\rangle$ and $T' = \langle P \cup P', A, I\rangle$ be abductive logic programs. If $T \vdash_{\emptyset}^{\Delta_1} L_1$ and $T' \vdash_{\Delta_1}^{\Delta_2} L_2$, where L_1 and L_2 are two conjunctions of atoms, then $T \vdash_{\emptyset}^{\Delta_2} L_1 \wedge L_2$.*

Proof. From $T \vdash_{\emptyset}^{\Delta_1} L_1$ and lemma 1 we have that

$$lhm(P \cup \Delta_1) \models L_1$$

From the definition of abductive proof procedure we have that $\Delta_1 \subseteq \Delta_2$. Since we consider the positive version of programs, $P \cup \Delta_1$ and $P \cup P' \cup \Delta_2$ are definite logic programs. From the monotonicity of definite logic programs $lhm(P \cup \Delta_1) \subseteq lhm(P \cup P' \cup \Delta_2)$ therefore

$$lhm(P \cup P' \cup \Delta_2) \models L_1$$

From $T' \vdash_{\Delta_1}^{\Delta_2} L_2$, by the soundness of the abductive proof procedure, we have that there exists an abductive model M_2 such that $M_2 \models L_2$ and $\Delta_2 \subseteq M_2 \cap \mathcal{L}^{\mathcal{A}}$. From proposition 1, there exists a set $H_2 \subseteq \mathcal{L}^{\mathcal{A}}$ such that $M_2 = lhm(P \cup P' \cup H_2)$. Since abducible and default predicates have no definition in $P \cup P'$, we have that $M_2 \cap \mathcal{L}^{\mathcal{A}} = H_2$ and $\Delta_2 \subseteq H_2$. Therefore $M_2 \supseteq lhm(P \cap P' \cap \Delta_2)$ and

$$M_2 \models L_1$$

From $M_2 \models L_2$ and from the weak completeness of the abductive proof procedure, we have that

$$T' \vdash_{\Delta_1}^{\Delta_2} L_1 \wedge L_2$$

We can now give the soundness theorem for our algorithm.

Theorem 4 (Soundness). *The AIIM LAP is sound.*

Proof. Let us consider first the case in which the target predicates are not abducible and therefore no assumption is added to the training set during the computation. In order to prove that the algorithm is sound, we have to prove that, for any given sets E^+ and E^-, the program T' that is output by the algorithm is such that

$$T' \vdash_{\emptyset}^{\Delta} E^+, not\_E^-$$

LAP learns the program T' by iteratively adding a new clause to the current hypothesis, initially empty. Each clause is tested by trying an abductive derivation for each positive and for the complement of each negative example. Let $E_c^+ = \{e_1^+ \ldots e_{n_c}^+\}$ be the set of positive examples whose conjunction is covered

by clause c and let $E^- = \{e_1^- \ldots e_m^-\}$. Clause c is added to the current hypothesis H when:

$$\exists E_c^+ \subseteq E^+ : \quad E_c^+ \neq \emptyset, \ \forall i \in \{1 \ldots n_c\} : \quad P \cup H \cup \{c\} \vdash_{\Delta_{i-1}^+}^{\Delta_i^+} e_i^+$$

$$\forall j \in \{1 \ldots m\} : \quad P \cup H \cup \{c\} \vdash_{\Delta_{j-1}^-}^{\Delta_j^-} not\_e_j^-$$

where $\Delta_0^+ = \Delta_H$, $\Delta_{i-1}^+ \subseteq \Delta_i^+$ and $\Delta_0^- = \Delta_{n_c}^+$. By induction on the examples and by theorem 3 with $P' = \emptyset$, we can prove that

$$\langle P \cup H \cup \{c\}, A, IC \rangle \vdash_{\Delta_H}^{\Delta_{H \cup \{c\}}} E_c^+, not\_E^-$$

where $\Delta_{H \cup \{c\}} = \Delta_m^-$. At this point, it is possible to prove that

$$T' \vdash_{\emptyset}^{\Delta} E_{c_1}^+ \cup \ldots \cup E_{c_k}^+, not\_E^-$$

by induction on the clauses and by theorem 3. From this and from the sufficiency stopping criterion (see Figure 2) we have that $E_{c_1}^+ \cup \ldots \cup E_{c_k}^+ = E^+$.

We now have to prove soundness when the target predicates are abducible as well and the training set is enlarged during the computation. In this case, if the final training sets are E_F^+ and E_F^-, we have to prove that

$$T' \vdash_{\emptyset}^{\Delta} E_F^+, not\_E_F^-$$

If a positive assumption is added to E^+, then the resulting program will contain a clause that will cover it because of the sufficiency stopping criterion. If a negative assumption not_e^- is added to E^- obtaining E'^-, clauses that are added afterwards will derive $not\_E'^-$. We have to prove also that clauses generated before allow $not\_E'^-$ to be derived. Consider a situation where not_e^- has been assumed during the testing of the last clause added to H. We have to prove that

$$\langle P \cup H, A, IC \rangle \vdash_{\emptyset}^{\Delta} E_H^+, not\_E^- \Rightarrow \langle P \cup H, A, IC \rangle \vdash_{\emptyset}^{\Delta} E_H^+, not\_E'^-$$

where $not\_e^- \in \Delta$ and $e^- \in E'^-$. From the left part of the implication and for the soundness of the abductive proof procedure, we have that there exists an abductive model M such that $\Delta \subseteq M \cap \mathcal{L}^{\mathcal{A}}$. From $not\_e^- \in \Delta$, we have that $not\_e^- \in M$ and therefore by weak completeness

$$\langle P \cup H, A, IC \rangle \vdash_{\emptyset}^{\Delta} not\_e^-$$

By induction and by theorem 3, we have the right part of the implication.

We turn now to the incompleteness of the algorithm. LAP is incomplete because a number of choice points have been overlooked in order to reduce the computational complexity. The first source of incompleteness comes from the fact that, after a clause is added to the theory, it is never retracted. Thus, it can be the case that a clause not in the solution is learned and the restrictions imposed on

the rest of the learning process by the clause (through the examples covered and their respective assumptions) prevent the system from finding a solution even if there is one. In fact, the algorithm performs only a greedy search in the space of possible programs, exploring completely only the smaller space of possible clauses. However, this source of incompleteness is not specific to LAP because most ILP systems perform such a greedy search in the programs space.

The following source of incompleteness, instead, is specific to LAP. For each example, there may be more than one explanation and, depending on the one we choose, the coverage of other examples can be influenced. An explanation Δ_1 for the example e_1 may prevent the coverage of example e_2, because there may not be an explanation for e_2 that is consistent with Δ_1, while a different choice for Δ_1 would have allowed such a coverage. Thus, in case of a failure in finding a solution, we should backtrack on example explanations.

We decided to overlook these choice points in order to obtain an algorithm that is more effective in the average case, but we might not have done so. In fact, these choice points have a high computational cost, and they must be considered only when a high number of different explanations is available for each example. However, this happens only for the cases in which examples are highly interrelated, i.e., there are relations between them or between objects (constants) related to them. This case is not very common in concept learning, where examples represent instances of a concept and the background represents information about each instance and its possible parts. In most cases, instances are separate entities that have few relations with other entities.

5 Examples

5.1 Learning exceptions

In this section, we show how LAP learns exceptions to classification rules. The example is taken from [16].

Let us consider the following abductive background theory $B = \langle P, A, IC \rangle$ and training sets E^+ and E^-:

$$
\begin{aligned}
P = \{&bird(X) \leftarrow penguin(X).\\
&penguin(X) \leftarrow superpenguin(X).\\
&bird(a). \quad bird(b). \quad penguin(c). \quad penguin(d).\\
&superpenguin(e). \quad superpenguin(f).\}\\
A = \{&abnorm_1/1, abnorm_2/1, not\_abnorm_1/1, not\_abnorm_2/1\}\\
IC = \{&\leftarrow abnorm_1(X), not\_abnorm_1(X).\\
&\leftarrow abnorm_2(X), not\_abnorm_2(X).\}\\
&\leftarrow flies(X), not\_flies(X).\}
\end{aligned}
$$

$$
\begin{aligned}
E^+ &= \{flies(a), flies(b), flies(e), flies(f)\}\\
E^- &= \{flies(c), flies(d)\}
\end{aligned}
$$

Moreover, let the bias be:

$flies(X) \leftarrow \alpha$ where $\alpha \subset \{superpenguin(X), penguin(X), bird(X),$
$not\_abnorm_1(X), not\_abnorm_2(X)\}$
$abnorm_1(X) \leftarrow \beta$ and $abnorm_2(X) \leftarrow \beta$ where
$\beta \subset \{superpenguin(X), penguin(X), bird(X)\}$

The algorithm first generates the following rule (R_1):

$flies(X) \leftarrow superpenguin(X).$

which covers $flies(e)$ and $flies(f)$ that are removed from E^+. Then, in the specialization loop, the rule $R_2 = flies(X) \leftarrow bird(X).$ is generated which covers all the remaining positive examples $flies(a)$ and $flies(b)$, but also the negative ones. In fact, the abductive derivations for $not\_flies(c)$ and $not\_flies(d)$ fail. Therefore, the rule must be further specialized by adding a new literal. The abducible literal not_abnorm_1 is added to the body of R_2 obtaining R_3:

$flies(X) \leftarrow bird(X), not\_abnorm_1(X).$

Now, the abductive derivations for the negative examples $flies(a)$ and $flies(b)$ succeed abducing $\{not\_abnorm_1(a), not\_abnorm_1(b)\}$ and the derivations $not\_flies(c)$ and $not\_flies(d)$ succeed abducing $\{abnorm_1(c), abnorm_1(d)\}$.

At this point the system adds the literals abduced to the training set and tries to generalize them, by generating a rule for $abnorm_1/1$. Positive abduced literals for $abnorm_1/1$ form the set E^+, while negative abduced literals form the set E^-. The resulting induced rule is (R_4):

$abnorm_1(X) \leftarrow penguin(X).$

No positive example is now left in the training set therefore the algorithm ends by producing the following abductive rules:

$flies(X) \leftarrow superpenguin(X).$
$flies(X) \leftarrow bird(X), not\_abnorm_1(X).$
$abnorm_1(X) \leftarrow penguin(X).$

A result similar to ours is obtained in [16], but exploiting "classical" negation and priority relations between rules rather than abduction. By integrating induction and abduction, we obtain a system that is more general than [16].

5.2 Learning from incomplete knowledge

Abduction is particularly suitable for modelling domains in which there is incomplete knowledge. In this example, we want to learn a definition for the concept $father$ from a background knowledge containing facts about the concepts *parent* and *male*. Knowledge about *male* is incomplete and we can make assumptions about it by considering it as an abducible. We have the abductive background theory $B = \langle P, A, IC \rangle$ and training set:

$P = \{$ $parent(john, mary).$ $\quad male(john).$
$parent(david, steve).$ $\quad parent(kathy, ellen).$
$female(kathy).\}$
$A = \{male/1, female/1\}$
$IC = \{\leftarrow male(X), female(X).\}$
$E^+ = \{father(john, mary), father(david, steve)\}$
$E^- = \{father(john, steve), father(kathy, ellen)\}$

Moreover, let the bias be

$father(X,Y) \leftarrow \alpha$ where $\alpha \subset \{parent(X,Y), parent(Y,X),$
$male(X), male(Y), female(X), female(Y)\}$

At the first iteration of the specialization loop, the algorithm generates the rule
$father(X,Y) \leftarrow$.
which covers all the positive examples but also all the negative ones. Therefore another iteration is started and the literal $parent(X,Y)$ is added to the rule
$father(X,Y) \leftarrow parent(X,Y)$.
This clause also covers all the positive examples but also the negative example
$father(kathy, ellen)$.
Note that up to this point no abducible literal has been added to the rule, therefore no abduction has been made and the set Δ is still empty. Now, an abducible literal is added to the rule, $male(X)$, obtaining
$father(X,Y) \leftarrow parent(X,Y), male(X)$.
At this point the coverage of examples is tested. $father(john, mary)$ is covered abducing nothing because we have the fact $male(john)$ in the background. The other positive example, $father(david, steve)$, is covered with the abduction of $\{male(david), not\_female(david)\}$.

Then the coverage of negative examples is tested by starting the abductive derivations
$\leftarrow not\_father(john, steve)$.
$\leftarrow not\_father(kathy, ellen)$.
The first derivation succeeds with an empty explanation while the second succeeds abducing $not\_male(kathy)$ which is consistent with the fact $female(kathy)$ and the constraint $\leftarrow male(X), female(X)$. Now, no negative example is covered, therefore the specialization loop ends. No atom from Δ is added to the training set because the predicates of abduced literals are not target. The positive examples covered by the rules are removed from the training set which becomes empty. Therefore also the covering loop terminates and the algorithm ends, returning the rule
$father(X,Y) \leftarrow parent(X,Y), male(X)$.
and the assumptions
$\Delta = \{male(david), not\_female(david), not\_male(kathy)\}$.

6 Related Work

We will first mention our previous work in the field, and then related work by other authors.

In [29] we have presented the definition of the extended learning problem and a preliminary version of the algorithm for learning abductive rules.

In [30] we have proposed an algorithm for learning abductive rules obtained modifying the extensional ILP system FOIL [37]. Extensional systems differ from intensional ones (as the one presented in this paper) because they employ a different notion of coverage, namely *extensional coverage*. We say that the

program P *extensionally covers* example e if there exists a clause of P, $l \leftarrow l_1, \ldots, l_n$ such that $l = e$ and for all i, $l_i \in E^+ \cup lhm(B)$. Thus examples can be used also for the coverage of other examples. This has the advantage of allowing the system to learn clauses independently from each other, avoiding the need for considering different orders in learning the clauses and the need for backtracking on clause addition. However, it has also a number of disadvantages (see [13] for a discussion about them). In [30] we have shown how the integration of abduction and induction can solve some of the problems of extensional systems when dealing with recursive predicates and programs with negation.

In [17] the authors discuss various approaches for the integration of abduction and induction. They examine how abduction can be related to induction specifically in the case of Explanation Based Learning, Inductive Learning and Theory Revision. The authors introduce the definition of a learning problem integrating abduction (called Abductive Concept Learning) that has much inspired our work. Rather than considering it as the definition of a problem to be solved and presenting an algorithm for it, they employ the definition as a general framework where to describe specific cases of integration.

Our definition differs from Abductive Concept Learning on the condition that is imposed on negative examples: in [17] the authors require that negative examples not be abductively entailed by the theory. Our condition is weaker because it requires that there be an explanation for not_e^-, which is easier to be met than requiring that there is no explanation for e^-. In fact, if there is an explanation for not_e^-, this does not exclude that there is an explanation also for e^-, while if there is no explanation for e^- then there is certainly an explanation for not_e^-. We consider a weaker condition on negative examples because the strong condition is difficult to be satisfied without learning integrity constraints. For example, in section 5.2, the learned program also satisfies the stronger condition of [17], because for the negative example $father(kathy, ellen)$ the only abductive explanation $\{male(kathy)\}$ is inconsistent with the integrity constraint $\leftarrow male(X), female(X)$. However, if that constraint was not available in the background, the stronger condition would not be satisfiable.

Moreover, in [17] the authors suggest another approach for the integration of abduction in learning that consists in explaining the training data of a learning problem in order to generate suitable or relevant background data on which to base the inductive generalization. Differently from us, the authors allow the use of integrity constraints for rule specialization, while we rely only on the addition of a literal to the body of the clause. Adding integrity constraints for specializing rules means that each atom derived by using the rules must be checked against the constraints, which can be computationally expensive. Moreover, the results of soundness and weak completeness can not be used anymore for the extended proof procedure.

In [2] an integrated abductive and inductive framework is proposed in which abductive explanations that may include general rules can be generated by incorporating an inductive learning method into abduction. The authors transform a proof procedure for abduction, namely SLDNFA, into a proof procedure for

induction, called SLDNFAI. Informally, SLDNFA is modified so that abduction is replaced by induction: when a goal can not be proven, instead of adding it to the theory as a fact, an inductive procedure is called that generates a rule covering the goal. However, the resulting learning is not able to a learn a rule and, at the same time, make specific assumptions about missing data in order to cover examples.

The integration of induction and abduction for knowledge base updating has been studied in [11] and [1]. Both systems proposed in these papers perform incremental theory revision: they automatically modify a knowledge base when it violates a newly supplied integrity constraint. When a constraint is violated, they first extract an uncovered positive example or a covered negative example from the constraint and then they revise the theory in order to make it consistent with the example, using techniques from incremental concept learning. The system in [11] differs from the system in [1] (called RUTH) because it relies on an oracle for the extraction of examples from constraints, while RUTH works non interactively. Once the example has been extracted from the constraint, both systems call similar inductive operators in order to update the knowledge base. In [11] the authors use the inductive operators of Shapiro's MIS system [38].

In [28], we have shown that LAP can be used to perform the knowledge base updating tasks addressed by the systems in [11] and [1], by exploiting the abductive proof procedure in order to extract new examples from a constraint on target predicates. While systems in [11, 1] can generate examples that violate other integrity constraints and new inconsistencies have to be recovered at the next iteration of the learning loop, in [28] we are able to select the examples that allow the minimal revision of the theory. Another relevant difference is that our system is a batch learner while the systems in [11, 1] are incremental learners: since we have all the examples available at the beginning of the learning process, we generate only clauses that do not cover negative examples and therefore we do not have to revise the theory to handle covered negative examples, i.e., to retract clauses. As regards the operators that are used in order to handle uncovered positive examples, we are able to generate a clause that covers a positive example by also making some assumptions, while in [11] they can cover an example either by generating a clause or by assuming a fact for covering it, but not the two things at the same time. RUTH, instead, is able to do this, and therefore would be able to solve the problem presented in Section 5.2. Moreover, RUTH considers abduced literals as new examples, therefore it would be able to solve as well the problems in Section 5.1.

As concerns the treatment of exceptions to induced rules, it is worth mentioning that our treatment of exceptions by means of the addition of a non-abnormality literal to each rule is similar to the one in [35]. The difference is that the system in [35] performs declarative debugging, not learning, therefore no rule is generated. In order to debug a logic program, in [35] the authors first transform it by adding a different default literal to each rule in order to cope with inconsistency, and add a rule (with an abducible in the body) for each predicate in order to cope with predicate incompleteness. These literals are then used as

assumptions of the correctness of the rule, to be possibly revised in the face of a wrong solution. The debugging algorithm determines, by means of abduction, the assumptions that led to the wrong solution, thus identifying the incorrect rules.

In [5] the authors have shown that is not possible, in general, to preserve correct information when incrementally specializing within a classical logic framework, and when learning exceptions in particular. They avoid this drawback by using learning algorithms which employ a nonmonotonic knowledge representation. Several other authors have also addressed this problem, in the context of Logic Programming, by allowing for exceptions to (possibly induced) rules [16, 10]. In these frameworks, nonmonotonicity and exceptions are dealt with by learning logic programs with negation. Our approach in the treatment of exceptions is very related to [16]. They rely on a language which uses a limited form of "classical" (or, better, syntactic) negation together with a priority relation among the sentences of the program [25]. However, in [20] it has been shown that negation by default can be seen as a special case of abduction. Thus, in our framework, by relying on ALP, we can achieve greater generality than [16]: besides learning exceptions, LAP is able to learn from incomplete knowledge and to learn theories for abductive reasoning.

In what concerns learning from incomplete information, many ILP systems include facilities in order to handle this problem, for example FOIL [37], Progol [34], mFOIL [19]. The approach that is followed by all these systems is fundamentally different with respect to ours: they are all based on the use of heuristic necessity and sufficiency stopping criteria and of special heuristic functions for guiding the search. The heuristic stopping criteria relaxes the requirements of consistency and completeness of the learned theory: the theory must cover (not cover) "most" positive (negative) examples, where the exact amount of "most" is determined heuristically. These techniques allow the systems to deal with imperfect data in general, including noisy data (data with random errors in training examples and in the background knowledge) and incomplete data. In this sense, their approach is more general than ours, because we are not able to deal with noisy data. Their approach is equivalent to discarding some examples, considering them as noisy or insufficiently specified, while in our approach no example is discarded, the theory must be complete and consistent (in the abductive sense) with each example.

7 Conclusions and Future Work

We have presented the system LAP for learning abductive logic programs. We consider an extended ILP problem in which both the background and target theory are abductive theories and coverage by deduction is replaced with coverage by abduction.

In the system, abduction is used for making assumptions about incomplete predicates of the background knowledge in order to cover the examples. In this way, general rules are generated together with specific assumptions relative to

single examples. If these assumptions regard an abnormality literal, they can be used as examples for learning a definition for the class of exceptions.

LAP is obtained from the basic top-down ILP algorithm by substituting, for the coverage testing, the Prolog proof procedure with an abductive proof procedure. LAP has been implemented in Sicstus Prolog 3#5: the code of the system and of the examples shown in the paper are available at

`http://www-lia.deis.unibo.it/Staff/FabrizioRiguzzi/LAP.html`

In the future, we will test the algorithm on real domains where there is incompleteness of the data. As regards the theoretical aspects, we will investigate the problem of extending the proposed algorithm in order to learn full abductive theories, including integrity constraints as well. The integration of the algorithm with other systems for learning constraints, such as Claudien [12] and ICL [14], as proposed in [27], seems very promising in this respect.

Our approach seems also promising for learning logic programs with two kinds of negation (e.g., default negation and explicit negation), provided that positive and negative examples are exchanged when learning a definition for the (explicit) negation of a concept, and suitable integrity constraints are added to the learned theory so as to ensure non-contradictoriness. This is also subject for future work.

Acknowledgment

We would like to thank the anonymous referees and participants of the post-ILPS97 Workshop on Logic Programming and Knowledge Representation for useful comments and insights on this work. Fabrizio Riguzzi would like to thank Antonis Kakas for many interesting discussions on the topics of this paper they had while he was visiting the University of Cyprus.

References

1. H. Adé and M. Denecker. RUTH: An ILP theory revision system. In *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems*, 1994.
2. H. Adé and M. Denecker. AILP: Abductive inductive logic programming. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1995.
3. J. J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*, volume 1111 of *LNAI*. SV, Heidelberg, 1996.
4. M. Bain and S. Muggleton. Non-monotonic learning. In S. Muggleton, editor, *Inductive Logic Programming*, chapter 7, pages 145–161. Academic Press, 1992.
5. M. Bain and S. Muggleton. Non-monotonic learning. In S. Muggleton, editor, *Inductive Logic Programming*, pages 145–161. Academic Press, 1992.
6. F. Bergadano and D. Gunetti. Learning Clauses by Tracing Derivations. In *Proceedings 4th Int. Workshop on Inductive Logic Programming*, 1994.
7. F. Bergadano and D. Gunetti. *Inductive Logic Programming*. MIT press, 1995.
8. F. Bergadano, D. Gunetti, M. Nicosia, and G. Ruffo. Learning logic programs with negation as failure. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 107–123. IOS Press, 1996.

9. A. Brogi, E. Lamma, P. Mancarella, and P. Mello. A unifying view for logic programming with non-monotonic reasoning. *Theoretical Computer Science*, 184:1–59, 1997.
10. L. De Raedt and M. Bruynooghe. On negation and three-valued logic in interactive concept learning. In *Proceedings of the 9th European Conference on Artificial Intelligence*, 1990.
11. L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence*, 53:291–307, 1992.
12. L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993.
13. L. De Raedt, N. Lavrač, and S. Džeroski. Multiple predicate learning. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 221–240. J. Stefan Institute, 1993.
14. L. De Raedt and W. Van Lear. Inductive constraint logic. In *Proceedings of the 5th International Workshop on Algorithmic Learning Theory*, 1995.
15. M. Denecker, L. De Raedt, P. Flach, and A. Kakas, editors. *Proceedings of ECAI96 Workshop on Abductive and Inductive Reasoning.* Catholic University of Leuven, 1996.
16. Y. Dimopoulos and A. Kakas. Learning Non-monotonic Logic Programs: Learning Exceptions. In *Proceedings of the 8th European Conference on Machine Learning*, 1995.
17. Y. Dimopoulos and A. Kakas. Abduction and inductive learning. In *Advances in Inductive Logic Programming.* IOS Press, 1996.
18. P.M. Dung. Negation as hypothesis: An abductive foundation for logic programming. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 3–17. MIT Press, 1991.
19. S. Džeroski. Handling noise in inductive logic programming. Master's thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, 1991.
20. K. Eshghi and R.A. Kowalski. Abduction compared with Negation by Failure. In *Proceedings of the 6th International Conference on Logic Programming*, 1989.
21. F. Esposito, E. Lamma, D. Malerba, P. Mello, M. Milano, F. Riguzzi, and G. Semeraro. Learning abductive logic programs. In Denecker et al. [15].
22. C. Hartshorne and P. Weiss, editors. *Collected Papers of Charles Sanders Peirce, 1931–1958*, volume 2. Harvards University Press, 1965.
23. A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2:719–770, 1993.
24. A.C. Kakas and P. Mancarella. On the relation between truth maintenance and abduction. In *Proceedings of the 2nd Pacific Rim International Conference on Artificial Intelligence*, 1990.
25. A.C. Kakas, P. Mancarella, and P.M. Dung. The acceptability semantics for logic programs. In *Proceedings of the 11th International Conference on Logic Programming*, 1994.
26. A.C. Kakas and F. Riguzzi. Learning with abduction. Technical Report TR-96-15, University of Cyprus, Computer Science Department, 1996.
27. A.C. Kakas and F. Riguzzi. Learning with abduction. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, 1997.
28. E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Integrating induction and abduction in logic programming. To appear on Information Sciences.

29. E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Integrating Induction and Abduction in Logic Programming. In P. P. Wang, editor, *Proceedings of the Third Joint Conference on Information Sciences*, volume 2, pages 203–206, 1997.
30. E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Introducing Abduction into (Extensional) Inductive Logic Programming Systems. In M. Lenzerini, editor, *AI*IA97, Advances in Artificial Intelligence, Proceedings of the 5th Congress of the Italian Association for Artificial Intelligence*, number 1321 in LNAI. Springer-Verlag, 1997.
31. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
32. L. Martin and C. Vrain. A three-valued framework for the induction of general logic programs. In *Advances in Inductive Logic Programming*. IOS Press, 1996.
33. R. Michalski, J.G. Carbonell, and T.M. Mitchell (eds). *Machine Learning - An Artificial Intelligence Approach*. Springer-Verlag, 1984.
34. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
35. L. M. Pereira, C. V. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In L. M. Pereira and A. Nerode, editors, *Proceedings of the 2nd International Workshop on Logic Programming and Non-monotonic Reasoning*, pages 316–330. MIT Press, 1993.
36. D.L. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 32, 1988.
37. J. R. Quinlan and R.M. Cameron-Jones. Induction of Logic Programs: FOIL and Related Systems. *New Generation Computing*, 13:287–312, 1995.
38. E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.