AILP : Abductive Inductive Logic Programming

Hilde Adé and **Marc Denecker**
Department of Computer Science, K.U.Leuven
Celestijnenlaan 200 A, B-3001 Heverlee, Belgium

Abstract

Inductive Logic Programming (ILP) is often situated as a research area emerging at the intersection of Machine Learning and Logic Programming (LP). This paper makes the link more clear between ILP and LP, in particular, between ILP and Abductive Logic Programming (ALP), i.e., LP extended with abductive reasoning. We formulate a generic framework for handling incomplete knowledge. This framework can be instantiated both to ALP and ILP approaches. By doing so more light is shed on the relationship between abduction and induction. As an example we consider the abductive procedure SLDNFA, and modify it into an inductive procedure which we call SLDNFAI.

Keywords: Inductive Logic Programming, Abductive Logic Programming, Incomplete Knowledge, Intensional Knowledge Base Updating, Theory Revision.

1 Introduction

It is often argued that the use of - a subset of - first order logic as a representation language situates Inductive Logic Programming (ILP) at the intersection of Logic Programming and Machine Learning. The research in ILP is concerned with the derivation of logic programs from (positive and negative) evidence in the presence of background knowledge. Characteristic for this approach is the use of *induction*, i.e., a form of synthetic reasoning that infers general laws from observations. [Muggleton and De Raedt, 1994] reviews theoretical results, implemented systems and practical applications in this area. Two major problems tackled in ILP are predicate learning and theory revision. The former is concerned with the induction of rules for an undefined or partially defined predicate from examples, whereas the latter concerns the updating of a theory (a logic program) when it is inconsistent with newly incoming information. Prototype systems are - amongst others - the predicate learners MIS [Shapiro, 1983], FOIL [Quinlan, 1990], GOLEM [Muggleton and Feng, 1990], CLINT [De Raedt, 1992], and the theory revision systems RX [Tankitvanitch and Shimura, 1992], KR-FOCL [Pazzani and Brunk, 1991] and RUTH [Adé *et al.*, 1994].

In the area of Logic Programming (LP) *abduction* has been recognized as an important form of non-monotonic reasoning. It is a form of synthetic reasoning, which - as opposed to induction - infers explanations for observed facts, according to known (general) laws. Abduction has been shown useful for fault diagnosis [Charniak and McDermott, 1985], planning and temporal reasoning [Denecker *et al.*, 1992] and knowledge assimilation [Kakas and Mancarella, 1990], and it has also been successfully applied in Intensional Knowledge Base Updating approaches such as [Bry, 1990], [Kakas and Mancarella, 1990] and [Guessoum and Lloyd, 1990]. A critical survey on the extension of logic programming to perform abductive reasoning can be found in [Kakas *et al.*, 1993].

In this paper we argue that both abduction and induction are different, yet related forms of reasoning on *incomplete knowledge*. They are both forms of hypothetical reasoning and attempt to "complete" the knowledge by proposing additional hypotheses. However, they differ in the sort of hypotheses.

According to the "declarative semantics", a logic program can be seen as a set of definitions for a number of concepts. This interpretation is already present in Clark's work on completion semantics [Clark, 1978]. Under Clark's interpretation, a program consists of a set of possibly empty definitions for all predicates; predicates are defined by enumerating exhaustively the cases in which they are true. Under this interpretation, a problem with the use of logic programs for knowledge representation, is that an expert needs to provide complete definitions for all predicates. In many applications, such complete information is not available. A natural solution is to extend the logic program formalism such that only a subset of the predicates is to be defined while other predicates can be left undefined. In addition, partial knowledge about these undefined predicates can be given using FOL axioms. The role of these FOL axioms is intimately tied to the representation of uncertainty: when a number of concepts cannot be defined, then other, less precise information may be available which can be represented as a set of *assertions*. E.g., if the predicates *father/1* and *mother/1* are declared as being undefined, the axiom $mother(X) \lor father(X) \leftarrow parent(X)$ expresses the weaker partial knowledge that it cannot be the case that someone is a parent without being a mother or a father.

The actual procedures for "completing" incomplete knowledge developed in the ALP and ILP area, are - as to be expected - quite different. The ALP approaches construct hypotheses in the form of ground facts for the undefined predicates. These facts are seen as the description of a scenario that explains a given observation. As opposed to this, ILP systems in general - even in case of a specific-to-general search - disallow facts. The hypotheses they generate are sets of rules that express general - scenario independent - information on the undefined predicates. As such an ALP approach is appropriate when the expert has great confidence in his general knowledge, but wants to find a concrete explanation for an observation. On the other hand, an ILP approach is more appropriate when the expert searches for general rules that compact the positive and negative evidence, and that later can be reused to classify new examples.

A first step towards integration of techniques from both domains was taken by [De Raedt and Bruynooghe, 1992]. De Raedt and Bruynooghe argue that - when reformulated within a logical framework - both intensional knowledge base updating and predicate learning are instances of the belief updating problem. As a proof for this claim they present an adapted version of Shapiro's MIS system in which techniques from both parent domains are integrated. These ideas were later further elaborated and implemented in the theory revision system RUTH [Adé *et al.*, 1994].

The contribution of our work is that we reformulate the ALP and ILP approaches as instances of a generic version of Bry's framework [Bry, 1990] for intensional knowledge base updating. In doing so we show that there is a clear relationship between inductive and abductive reasoning. Moreover, this setting enables us to introduce inductive reasoning into an abductive schema, thus resulting in a setting which one could call Abductive Inductive Logic Programming (AILP). As an example we transform a typical abductive procedure, namely SLDNFA into a new procedure SLDNFAI (SLDNFA-with-Induction) by enhancing it with a general-to-specific procedure for inducing clauses. Furthermore, this enhancement could also be beneficial for the ILP approach, since SLDNFA is a procedure for normal clauses, whereas the ILP setting used to be restricted to definite clauses.

This paper is organized as follows. In Section 2 we introduce a generic framework that formalizes different forms of hypothetical reasoning on incomplete knowledge. In Section 3 we map both the ALP and ILP approaches to this framework, and discuss the similarities and differences. In Section 4 we elaborate an abductive procedure for ILP, by introducing inductive reasoning into the abductive procedure SLDNFA. Finally, in Section 5 we formulate our conclusions.

2 Problem formalization

Intuitively one can outline the problem of handling *incomplete knowledge* as follows. Given is a logic program P, which is declaratively read as the description of the problem domain. One part of this program is regarded as being correct, the other part is regarded as incomplete. Also given is evidence E, containing requirements for a logic program P_{new}. The aim is to find such a P_{new} that satisfies all the restrictions imposed by E.

A logic programs P consists of a finite set $\mathcal{C}(P)$ of program clauses, a set $\mathcal{U}(P)$ of predicate names, namely those predicates that are not or only partially defined by clauses in $\mathcal{C}(P)$, and a finite set of First Order Logic (FOL) axioms $\mathcal{I}(P)$. The declarative semantics of P is given by the classical logic theory $comp(P) = CET \cup \mathcal{I}(P) \cup \{p \leftarrow B \mid p \leftarrow B \in \mathcal{C}(P) \text{ and } p \in \mathcal{U}(P)\} \cup \{ compdef(p, \mathcal{C}(P)) \mid p \notin \mathcal{U}(P)\}$. Here CET denotes Clark's equality theory and $compdef(p, \mathcal{C}(P))$ denotes the completed definition of the predicate p in $\mathcal{C}(P)$ [Clark, 1978]. CET expresses that two different terms represent different objects. $compdef(p, C(P))$ expresses that if $p \leftarrow B$ is in $\mathcal{C}(P)$ and B is true, then p is true, but also vice versa, if p is true then for at least one (instance of a) rule $p \leftarrow B$ of $\mathcal{C}(P)$, B is true. Note that for undefined or partially defined predicates, the rules in $\mathcal{C}(P)$ are not completed. $\mathcal{L}(P)$ denotes the language of P, i.e., the set of all well-formed formulas using predicates and functors in P.

A logic program P is said to be consistent iff comp(P) is a consistent FOL theory. We say that a logic program P is consistent w.r.t. a set of well-formed formulas $\mathcal{F} \subseteq \mathcal{L}(P)$ iff $comp(P) \cup \mathcal{F}$ is a consistent FOL theory. Finally, if $F \in \mathcal{L}(P)$ is a well-formed formula, $P \models F$ means $comp(P) \models F$.

To formalize hypothetical reasoning, we use an adapted version of Bry's framework [Bry, 1990]. We rely on a meta-predicate *new* for expressing the evidence E, i.e., for expressing statements that describe the desired logic program P_{new}. E.g., the constraint $new(mother(x)) \vee new(father(x)) \leftarrow parent(x)$ states that in P_{new} all parents should also be mothers or fathers.

According to [Bry, 1990] P and P_{new} can be viewed as two distinct logical theories, and a declarative expression of an update of P resulting in P_{new} is a finite set of logical formulas defining P_{new} in terms of P. The set of well-formed formulas $\mathcal{L}_{new}(P)$ is defined as an extension of $\mathcal{L}(P)$. The base of this set consists of the well-formed formulas of $\mathcal{L}(P)$ and the formula $new(F)$, where $F \in \mathcal{L}(P)$, and this set is closed under negation, logical connectives and the quantifiers $\forall$ and $\exists$. Definition 1 formalizes the notion of *evidence E to transform a logic program P into a logic program P_{new}*.

Definition 1 *Evidence E to transform a logic program P into a logic program P_{new} is a set of closed formulas $F_E \subseteq \mathcal{L}_{new}(P) \setminus \mathcal{L}(P)$.* □

Requiring that P_{new} satisfies all restrictions imposed by E is formalized to the requirement that $P_{new} \models E$. Finally, in Figure 1, we formalize the problem setting.

Given :
- a logic program $P = (\ \mathcal{C}(P), \mathcal{U}(P), \mathcal{I}(P)\ )$
- evidence $E \subseteq \mathcal{L}_{new}(P) \setminus \mathcal{L}(P)$

Find

a logic program P_{new} such that $P_{new} \models E$.

Figure 1: The formalized problem.

In [Bry, 1990] a non-Horn theory N for the meta-predicate *new* is defined in terms of operations on the object program facts and clauses. In Definition 4 we give a 'generic' version of this theory, in that we replaced the actual operations on the object program by the predicates *eval_pos_atom* and *eval_neg_atom* that are to be instantiated. The rules rely on a meta-predicate *clause* that ranges over the clauses of $\mathcal{C}(P)$ defining the predicates that are not in $\mathcal{U}(P)$. More specifically, define *Clause(P)* as the meta-program consisting of facts clause(A ← B) for each rule A ← B in $\mathcal{C}(P)$ and of facts undefined(A) for all atoms A containing a predicate of $\mathcal{U}(P)$.

Definition 2 *An* update *of a logic program P is a set of expressions of the form* eval_pos_atom(A) *or* eval_neg_atom(A) *where A is an atom.* □

Definition 3 *Let P be a logic program and U an update of P. The procedure* update *is a function that maps P and U to a logic program, such that the following correctness criterion is fulfilled*[1]*:*

- $update(P,U) \models A$ *if* $eval\_pos\_atom(A) \in U$
- $update(P,U) \models \neg A$ *if* $eval\_neg\_atom(A) \in U$

We define the logic program P_U *as the result of the call* $update(P,U)$. □

Definition 4 *The theory N for 'new'*

(1) new(A) ⇔ [∃ B: clause(A ← B) ∧ *new*(B)] ∨ [undefined(A) ∧ *eval_pos_atom*(A)]
(2) new(¬ A) ⇔ [∀ B: clause(A ← B) ⇒ *new*(¬B)] ∧ [undefined(A) ⇒ *eval_neg_atom*(A)]
(3) new(F ∧ G) ⇔ new(F) ∧ new(G)
(4) new(F ∨ G) ⇔ new(F) ∨ new(G)
(5) new(∀x R ⇒ F) ⇔ ∀x new(R) ⇒ new(F)
(6) new(∀x ¬R) ⇔ ∀x new(¬R)
(7) new(∃x F) ⇔ ∃x new(F)
(8) new(¬(F ∧ G)) ⇔ new(¬F) ∨ new(¬G)
(9) new(¬(F ∨ G)) ⇔ new(¬F) ∧ new(¬G)
(10) new(¬ ∀x R ⇒ F) ⇔ ∃x new(¬ R ∧ F)
(11) new(¬∀x R) ⇔ ∃x new(¬ R)
(12) new(¬∃x F) ⇔ ∀x new(¬F)

where A denotes an atomic formula in $\mathcal{L}$(P)*, F and G denote formulas in* $\mathcal{L}$(P)*, and R is a conjunction of atoms with x a free variable.* □

The if-parts of the equivalences can be viewed as a meta-program for *new*. They recursively distribute *new* over quantifiers and logical connectives. Once *new* applies to atoms, it is evaluated on the object program facts and clauses. The equivalences can be viewed as the completion [Clark, 1978] of this meta-program for *new*. Since negation in *new* occurs only at the object level and not in the meta-language, this theory correctly defines *new*. The following proposition is analogous to the one found in [Bry, 1990].

Proposition 1 *Let P be a logic program, U an update of P and* $F \in \mathcal{L}(P)$.

$$P_U \models F \quad \Leftrightarrow \quad Clause(P) \cup U \cup N \models new(F)$$

[1]As usual $T \models A$ with A an open atom is to be interpreted as $T \models \forall(A)$.

3 Abduction versus Induction

In this section we discuss the ALP and ILP approaches in terms of the generic problem setting of Figure 1 and the rewrite rules for the meta-predicate *new*.

3.1 ALP

In general, abductive reasoning is concerned with the construction of explanations for observations using general laws. These explanations are expressed in terms of a predetermined set of predicates, called *abducible* predicates. Moreover, most ALP approaches require the explanations to be a set of ground facts. First, this restricts the number of possibilities, and second, in ALP one is interested in constructing a concrete scenario rather than general rules. As a consequence updates of a logic program P are restricted to removing and inserting ground facts for the abducible predicates.

In the formalization of the previous section this means that ALP regards the logic program P as having a fixed part, consisting of $\mathcal{C}(P)$ and $\mathcal{I}(P)$, and the set $\mathcal{U}(P)$ containing the names of the *abducible* predicates. All of $\mathcal{C}(P)$ belongs to the complete part; the incomplete part defining the predicates of $\mathcal{U}(P)$ is empty. Once evidence E is given, it is translated into operations on the abducible predicates.

The general idea of an abductive approach is captured by the rules $(1)_{ALP}$ and $(2)_{ALP}$.

$(1)_{ALP}$: new(A) ⇔ [∃ B: clause(A ← B) ∧ *new*(B)] ∨ [abducible(A) ∧ *abduce*(A)]
$(2)_{ALP}$: new(¬A) ⇔ [∀ B: clause(A ← B) ⇒ *new*(¬B)] ∧ [abducible(A) ⇒ ∀ B ∈ Δ: *avoid_unification*(A,B)]

The effect of the meta-predicate *abduce* operating on the program P is that the atom A is added to the program as (part of) the definition for the abducible predicate. The meta-predicate *avoid_unification* has as effect that it continuously constrains the atom A so that it cannot unify with any of the abduced atoms. Note that both predicates *abduce* and *avoid_unification* are still generic in the sense that they do not give any exact procedural information. Every abductive system implements these predicates in its own particular way.

3.2 ILP

In general ILP approaches are concerned with the derivation of logic programs from positive and negative evidence in the presence of background knowledge. In most systems evidence is a set of positive and negative examples of predicates for which no or only incomplete definitions are given. We will call these predicates *inducible*. ILP aims at constructing general rules that define the inducible predicates, since one is interested in compacting the knowledge represented by the positive and negative examples. Furthermore, one wants to be able to reuse the induced rules for the classification of unknown examples.

In the formalization of Section 2 this means that the background knowledge is a part of $\mathcal{C}(P)$. It is the fixed part of P that is correct and complete, and that can be used as a logic program. $\mathcal{U}(P)$ is the set of *inducible* predicates. $\mathcal{C}(P)$ can contain clauses for these inducible predicates. However, these clauses are only an incomplete (or incorrect) definition. The evidence E can be written as $new(p_1 \wedge ... \wedge p_{m_p} \wedge \neg n_1 \wedge \neg n_{m_n})$ with $\{p_1, ..., p_{m_p}\}$, resp. $\{n_1, ..., n_{m_n}\}$, the set of positive, resp. negative examples given for the inducible predicates. Finally, P_{new} should be such that $\forall i, 1 \leq i \leq m_p$: $P_{new} \models p_i$, and $\forall j, 1 \leq j \leq m_n$: $P_{new} \not\models n_j$.

The rules $(1)_{ILP}$ and $(2)_{ILP}$ capture the general idea of the ILP way of implementing the predicates *eval_pos_atom* and *eval_neg_atom*.

$$
\begin{aligned}
(1)_{ILP}\ \text{new(A)} \ \Leftrightarrow\ & [\ \exists\ \text{B: clause(A} \leftarrow \text{B)} \wedge \textit{new}\text{(B)}\ ] \\
& \vee [\ \text{inducible(A)} \wedge \textit{induce}\text{(A)}\ ] \\
(2)_{ILP}\ \text{new}(\neg\text{A}) \Leftrightarrow\ & [\ \forall\ \text{B: clause(A} \leftarrow \text{B)} \Rightarrow \textit{new}(\neg\text{B})] \\
& \wedge [\ \text{inducible(A)} \Rightarrow \\
& (\ \forall\ \text{B : clause(A} \leftarrow \text{B)} \wedge \textit{new}\text{(B)} \Rightarrow \textit{refine}\text{(A} \leftarrow \text{B))]}
\end{aligned}
$$

A call to the meta-predicate *induce* has the following effect. First it is checked whether there is a clause in $\mathcal{C}(P)$ that covers the atom A[2]. If this is the case, nothing needs to be done. Else, a clause is constructed that covers the atom A and that is consistent with the negative evidence seen so far. We formulate the requirements for such a clause only in a very general way, since each approach has its own particular way of implementing the induction of a clause. For example in a specific-to-general approach one would start with a most specific clause, which can then later be generalized, whereas a general-to-specific approach would rather construct a most general clause covering the example and consistent with the negative evidence.

The meta-predicate *refine* is called when a negative example is covered by one or more clauses in the partial definitions for the inducible predicates. In that case these clauses are refined, i.e., made more specific, such that they no longer cover the negative example. Making a clause more specific is usually done by adding one or more literals to its body, although this is not the only way to implement *refine*. E.g., MIS [Shapiro, 1983] just removes the whole clause, and if necessary, replaces it later by a more specific clause.

3.3 Discussion

Taking a closer look at the rules $(1)_{ALP}$, $(2)_{ALP}$, $(1)_{ILP}$ and $(2)_{ILP}$, one can see some important similarities in the reasoning mechanisms of the ALP and ILP approaches. Consider for example the rules $(1)_{ALP}$ and $(1)_{ILP}$. The first disjunct of the right hand side looks very much the same: reduction of a goal via resolution with a clause of P. The main difference of course lies in the second disjunct of the rules, namely in the meta-predicates *abduce* and *induce*. The meta-predicate *abduce* reasons on the level of ground facts, whereas the meta-predicate *induce* is concerned with definite Horn clauses. However, there still is a certain analogy in these predicates. *induce* first backtracks over the possibly incomplete definition of the inducible predicate, and only if the atom is not covered a new clause in induced. Most abductive systems realize *abduce* by just inserting the atom as a new fact, but there exist more sophisticated abductive approaches that also first backtrack over the already abduced atoms, and only abduce a new fact when unification fails.

Notice that an analogue duality can be found in the rules $(2)_{ALP}$ and $(2)_{ILP}$. The first conjunct of the right hand side expresses that once a negative atomic goal unifies with the head of a clause, the instantiated body should not be true in P_{new}. The second conjunct expresses what should be done when the negative goal contains a predicate p of $\mathcal{U}(P)$. The difference between the ALP and the ILP approach lies in the fact that in ALP one must avoid unification with the abduced atoms for the predicate p, whereas in ILP one must avoid coverage by the (induced) clauses for p.

4 SLDNFAI : an abductive procedure for ILP

In this section we consider the abductive procedure, SLDNFA [Denecker *et al.*, 1992], and show that replacing the reasoning on the level of ground facts by an inductive reasoning schema, transforms it into an inductive procedure, which we call SLDNFAI.

4.1 SLDNFA

The SLDNFA procedure of [Denecker and De Schreye, 1992] is an abductive procedure for normal abductive programs. The procedure is a natural extension of SLDNF-resolution and incorporates both abduction and negation as failure. The major strengths of the algorithm are that it is able to handle both positive and negative goals, and also its way of treating non-ground abductive goals. The procedure starts with an (incomplete) logic program and a query *?- q.* that should succeed. SLDNFA produces a set $\Delta \subset \mathcal{L}_{new}$[3] such that $comp(P + \Delta) \models q$, where $P + \Delta = \mathcal{C}(P) \cup \Delta$ and $\mathcal{U}(P + \Delta) = \emptyset$. In SLDNFA both $\mathcal{I}(P)$ and $\mathcal{I}(P + \Delta)$ are empty. In terms of the formalization of Section 2 SLDNFA handles the following problem:

Given:

• a logic program $P = (\mathcal{C}(P), \mathcal{U}(P), \mathcal{I}(P))$, with $\mathcal{U}(P)$ the abducible predicates and $\mathcal{I}(P) = \emptyset$.

• evidence $E = \{\ new(\text{q}) \leftarrow \}$[4]

Find:

a logic program $P_{new} = P + \Delta$ with Δ a set of ground facts such that $comp(P + \Delta) \models q$

Basically the SLDNFA procedure proves an initial goal by constructing a set $\mathcal{PG}$ of goals that must succeed and a set $\mathcal{NG}$ of goals that must fail. SLDNFA tries to reduce goals in $\mathcal{PG}$ to the empty goal $\Box$, and tries to build finitely failed trees for the goals in $\mathcal{NG}$. We explain how the rules $(1)_{ALP}$ and $(2)_{ALP}$ are implemented. We start with the predicate *abduce* that handles the case where A is selected in a positive goal. SLDNFA solves this - via backtracking - by resolving A against all abduced atoms and finally, by skolemizing A - replacing all of

[2] A clause C covers an atom A if $P \cup \{C\} \models A$.

[3] Note that in SLDNFA the alphabet of $\mathcal{L}_{new}$ is extended with a set of skolem constants $sk_1, sk_2, ...$

[4] Note that this is just the query *?- q.* formulated using the meta-predicate *new*.

its variables by skolem constants - and adding the result to Δ. In order to solve problems with unification with skolem constants, classical unification is extended. In a first phase, extended unification treats skolem constants as variables; in a second phase, it skolemizes the terms bound to the original skolem constants. This is illustrated in Example 1.

Example 1 *Consider the fact* $p(f(g(Z),V)) \leftarrow$ *and the query* $\leftarrow r(X), p(X)$ *where* r *is an abducible predicate. Consider the following partial derivation (selected literals are underlined):*

$\mathcal{PG} = \{\leftarrow \underline{r(X)}, p(X)\}$
$\mathcal{PG} = \{\leftarrow \underline{p(sk)}\}$, $\quad \Delta = \{r(sk)\}$

When solving the positive goal $\leftarrow p(sk)$ *with* $p(f(g(Z),V)) \leftarrow$, *SLDNFA replaces* sk *by* $f(g(sk_1), sk_2)$ *and returns* $\Delta = \{r(f(g(sk_1), sk_2))\}$. □

The predicate *avoid_unification*, which handles the case where an abductive atom A is selected in a negative goal Q, is more complex. One must compute the failure tree obtained by resolving the goal with each of the abduced atoms in Δ. A problem is that the final Δ may not be totally known when A is selected. SLDNFA solves this by interleaving the computation of the failure tree with the construction of Δ. This is implemented by storing for each negative abductive goal the triplet (Q, A_Q, D_Q) where Q is the negative abductive goal, A_Q is the abductive atom selected in Q, and D_Q is the set of abduced atoms that have been resolved with Q. When a new fact is abduced, the stored goal is retrieved and is resolved with the new fact.

Finally, SLDNFA has a special *negative unification procedure* for handling the case where skolem constants occur in negative goals. We will not go into detail on this procedure. We just mention that it produces expressions of the form $sk \neq term$ as constraints on the generated solutions.

In [Denecker, 1993] the SLDNFA procedure is formalized and it is proven to be sound w.r.t. completion semantics: $comp(P + \Delta) \models q$. As a completeness result it is proven that if the computation terminates, then SLDNFA generates at least all *minimal* solutions.

4.2 A simple inductive procedure

We design a simple inductive procedure which can be seen as prototypical for a general-to-specific incremental ILP approach to predicate learning. In terms of the formalization of Section 2, our procedure handles the following problem:

Given:

• a logic program $P = (\mathcal{C}(P), \mathcal{U}(P), \mathcal{I}(P))$ where $\mathcal{U}(P)$ are the names of the unknown predicates and $\mathcal{I}(P) = \emptyset$

• evidence $E = new(\bigwedge_{i=1}^{m_p} p_i \wedge \bigwedge_{j=1}^{m_n} n_j)$, where the p_i are the positive examples and n_j the negative examples

Find:

a logic program $P_{new} = P \cup \mathcal{H}$ such that $P \cup \mathcal{H} \models E$[5]

As representation language we use functor-free, linked Horn clauses. The procedure starts with an empty hypothesis $\mathcal{H}$ and handles the examples incrementally. Clauses in $\mathcal{H}$ are represented as couples of the form $(c, coverage)$, where C is a clause and *coverage* is the set of positive examples encountered so far that are covered by c. The clauses of $\mathcal{C}(P)$ defining predicates of $\mathcal{U}(P)$ are considered to belong to $\mathcal{H}$. Their *coverage* is initialized to the empty set. We use a set $\mathcal{N}$ to store the negative examples handled so far.

We now specify how the predicates *induce* and *refine* are implemented in our inductive procedure. The predicate *induce* is called when a positive example is encountered. When it is already covered by one or more of the clauses in $\mathcal{H}$, it is added to the *coverage* of each of these clauses. When the example is not covered by the current $\mathcal{H}$, a most general clause is constructed that covers the example, and does not cover any of the negative examples in $\mathcal{N}$. The actual construction of such a clause goes as follows. The procedure starts with a clause with an empty body, and with as head a variabilized version of the positive example. This very general clause is then gradually refined until it does not cover any of the negative examples in the current $\mathcal{N}$. Refining a clause is done either by unifying two distinct variables, or by adding a literal to the body. The arguments of this literal should be distinct variables, with at least one variable in common with the original clause. The predicate can be any predicate of P.

The meta-predicate *refine* is called for handling a negative example. If the example is covered by one or more clauses in $\mathcal{H}$, each of these clauses is refined such that it no longer covers the negative example. Next it is checked for each of these refined clauses which examples in their *coverage* are no longer covered. The examples that are no longer covered by any of the clauses in $\mathcal{H}$ are then again added to E.

4.3 Introducing induction in SLDNFA : SLDNFAI

The basic idea is to maintain the overall structure of the SLDNFA procedure and to replace the abduction of the set of ground facts Δ by the induction of a set of clauses $\mathcal{H}$ using the inductive procedure of Section 4.2. For doing so, we will consider positive abductive goals as positive examples, and negative abductive goals as negative examples[6]. The set $\mathcal{H}$ is initialized with the clauses of $\mathcal{C}(P)$ defining predicates of $\mathcal{U}(P)$, with an empty coverage for each of these clauses. As in SLDNFA we use sets $\mathcal{PG}$ and $\mathcal{NG}$ to store positive and negative goals. For a positive abductive goal, SLDNFAI extends $\mathcal{H}$ - if necessary - with a new clause that is consistent with the negative examples encountered so far. These negative examples are stored in the set $\mathcal{N}$. Interleaving the computation of the failure tree for a negative abductive goal

[5]Note that in a classical ILP setting $P \cup \mathcal{H}$ under FOL semantics should entail the positive examples and should be consistent with the negative examples. When $P \cup \mathcal{H}$ is a definite program, this is equivalent with $P \cup \mathcal{H}$ under least Herbrand Model semantics entails both the positive and the negated negative examples.

[6]A positive, resp. negative, abductive goal is an atom for an abducible predicate, selected in a positive, resp. negative, goal.

and the construction of the set Δ is replaced by refining $\mathcal{H}$ each time a new negative example is encountered. We clarify this idea by illustrating SLDNFAI on a small example.

Example 2 *Consider the following fault diagnosis problem for lamps.*

(1) lamp(l_1) ←
(2) lamp(l_2) ←
(3) conn_to_battery(l_1,b_1) ←
(4) conn_to_battery(l_2,b_2) ←
(5) faulty_lamp(X)←lamp(X),conn_to_empty_battery(X,Y)
(6) conn_to_empty_battery(X,Y) ← conn_to_battery(X,Y)
(7) empty(b_1) ←
(8) false ← conn_to_empty_battery(X,b_2)
Initial query : ?- faulty_lamp(l_1), ¬false.

The predicate conn_to_empty_battery *is inducible. Note that this implies that clause* (6) *is not necessarily complete and correct. The following enumeration describes how SLDNFAI operates in order to make the initial query succeed. In each step we indicate the operation and the updated datastructures.*

1. *initialization*
   $\mathcal{PG}$ = {faulty_lamp(l_1), ¬false }, $\mathcal{NG}=\emptyset$, $\mathcal{N}=\emptyset$,
   $\mathcal{H}$={(conn_to_empty_battery(X,Y) ←conn_to_battery(X,Y),$\emptyset$)}
2. *resolve faulty_lamp(l_1) with clause* (5)
   $\mathcal{PG}$={lamp(l_1), conn_to_empty_battery(l_1,Y), ¬false }, $\mathcal{NG}=\emptyset$, $\mathcal{N}=\emptyset$, $\mathcal{H}$={(conn_to_empty_battery(X,Y) ←conn_to_battery(X,Y),$\emptyset$)}
3. *resolve lamp(l_1) with clause* (1)
   $\mathcal{PG}$={conn_to_empty_battery(l_1,Y), ¬false }, $\mathcal{NG}=\emptyset$, $\mathcal{N}=\emptyset$, $\mathcal{H}$={(conn_to_empty_battery(X,Y) ←conn_to_battery(X,Y),$\emptyset$)}
4. *conn_to_empty_battery(l_1, Y) is a positive example. Resolution with clause* (6) *and clause* (3) *instantiates the variable Y to b_1. conn_to_empty_battery(l_1,b_1) is put in the coverage of clause* (6).
   $\mathcal{PG}$={¬false }, $\mathcal{NG}=\emptyset$, $\mathcal{N}=\emptyset$, $\mathcal{H}$={(conn_to_empty_battery(X,Y) ← conn_to_battery(X,Y), {conn_to_empty_battery(l_1,b_1)})}
5. *switch ¬false to negative goal*
   $\mathcal{PG}$={□}, $\mathcal{NG}$={false}, $\mathcal{N}=\emptyset$,
   $\mathcal{H}$={(conn_to_empty_battery(X,Y) ←conn_to_battery(X,Y),{conn_to_empty_battery(l_1,b_1)})}
6. *resolve false with clause* (8)
   $\mathcal{PG}$={□}, $\mathcal{NG}$={conn_to_empty_battery(X,b_2)}, $\mathcal{N}=\emptyset$,
   $\mathcal{H}$={(conn_to_empty_battery(X,Y) ←conn_to_battery(X,Y),{conn_to_empty_battery(l_1,b_1)})}
7. *conn_to_empty_battery(X,b_2) is a negative example. Resolution with clause* (6) *and clause* (3) *instantiates X to l_2. $\mathcal{H}$ needs to be refined such that conn_to_empty_battery(l_2,b_2) is no longer covered. This can be done by adding the literal empty(Y) to the body of the clause in $\mathcal{H}$.*
   $\mathcal{PG}$={□}, $\mathcal{NG}$={□},
   $\mathcal{N}$={conn_to_empty_battery(l_2,b_2)},
   $\mathcal{H}$={(conn_to_empty_battery(X,Y) ← conn_to_battery(X,Y),empty(Y),{conn_to_empty_battery(l_1,b_1)})}

The result is that SLDNFAI has induced the clause conn_to_empty_battery(X,Y) ← conn_to_battery(X,Y), empty(Y) *for the predicate* conn_to_empty_battery. □

Let us for the sake of comparison discuss how SLDNFA would have operated on the same example. First, SLDNFA would not take into account clause (6), since it assumes that initially there is no definition for the abducible predicates. The first three steps in example 2 stay the same for SLDNFA. In step 4 however, SLDNFA would abduce the atom *conn_to_empty_battery(l_1,sk_1)*, with sk_1 a skolem constant, and add it to Δ. Step 5 and 6 stay the same as for SLDNFAI, but step 7 is again different. Since *conn_to_empty_battery(X,b_2)* is a negative abductive goal, unification with *conn_to_empty_battery(l_1,sk_1)* should be avoided. Therefore SLDNFA adds the constraint $sk_1 \neq b_2$. The result is that SLDNFA has abduced that lamp l_1 has an empty battery sk_1 which is different from b_2.

Which of both solutions - i.e., the one produced by SLDNFA or the one produced by SLDNFAI - is preferable of course depends on the actual problem situation, i.e., whether one is interested in general rules, or in a concrete scenario.

Finally, we want to make two important remarks concerning SLDNFAI. First, we claim that in a "degenerate" case, SLDNFAI reduces to the inductive procedure of Section 4.2. Indeed, when SLDNFAI is given as initial query the conjunction of all positive and negative examples, it will behave as the inductive procedure we described, and produce a set of clauses that covers all the positive examples, and none of the negative examples.

And second, we claim that by using the SLDNFA approach, SLDNFAI provides an elegant way of extending ILP approaches to handle normal clauses. The use of completion - as in SLDNFA - allows to extend the *covers* relation in a straightforward way to cope with negated atoms in the body of clauses. This approach was also taken by [Taylor, 1993] in their extension of a particular generalization operator, nl. absorption, towards normal clauses in an inductive learning context. Moreover, since SLDNFA can cope both with positive and negative goals, it functions in a very natural way as an example generator for both positive and negative examples for the inducible predicates, as can be seen in Example 2.

5 Conclusions

In this paper we have made the link more clear between ALP and ILP. We have argued that both induction and abduction are different, yet related forms of hypothetical reasoning on incomplete knowledge. On the one hand, abductive procedures complete this incomplete knowledge by hypotheses containing ground facts. Such a solution is appropriate when one has confidence in the available general laws, and one is interested in finding a concrete scenario that explains certain observations. On the other hand, inductive procedures produce general rules to complete the incomplete knowledge. This is interesting in case one is interested in compacting the positive and negative evidence, and in later reusing the rules for classifying new examples.

Borrowing ideas of Bry's setting [Bry, 1990] for intensional knowledge base updating, we have developed a generic framework that formalizes the problem of completing incomplete knowledge. In this framework a declarative expression of an update of a logic program P resulting in a logic program P_{new} is a set of logical formulas using a meta-predicate *new* that applies on atoms and negated atoms. A set of rewrite rules translates these formulas into operations on the object level program.

Next, we mapped our framework both to the ALP and ILP approaches. In doing so, more light is shed on the relationship between abduction and induction. As an illustration we considered the abductive procedure SLDNFA of [Denecker and De Schreye, 1992]. We reformulated this procedure as an instantiation of our generic approach and showed that this enabled us to replace in a straightforward way the abduction of ground facts by a general-to-specific procedure for inducing clauses. We call the resulting procedure SLDNFAI, i.e., SLDNFA extended with induction. Finally, we claim that this enhancement of SLDNFA can also be beneficial for the ILP approach, since SLDNFA works with normal program clauses, whereas ILP is restricted to definite clauses.

As a general conclusion we can say that our paper has established a clear relationship between ALP and ILP. We pointed out that it is worthwhile transferring results and techniques from one domain to the other. E.g., ILP could benefit from formal and theoretical results obtained in ALP, whereas ALP could benefit from inductive techniques to alleviate the restriction to abducing only ground facts. We are convinced that the first promising results achieved in this paper suggest that further research on this issue can lead to interesting results.

Acknowledgements

We wish to thank Maurice Bruynooghe, Gunther Sablon and Luc De Raedt for their comments on earlier versions of this paper, and for the inspiring discussions. Research for this paper was partially supported by the Esprit Bra nr.6020 (ILP). We are also grateful to the anonymous reviewers for their encouraging comments.

References

[Adé *et al.*, 1994] H. Adé, B. Malfait, and L. De Raedt. RUTH : an ILP Theory Revision System. In *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems (ISMIS94)*, 1994.

[Bry, 1990] Francois Bry. Intensional updates: abduction via deduction. In D. Warren and P. Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 561–578. The MIT Press, 1990.

[Charniak and McDermott, 1985] E. Charniak and D. McDermott. *Introduction to Artifical Intelligence.* Addison-Wesley, 1985.

[Clark, 1978] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[De Raedt and Bruynooghe, 1992] L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence*, 53:291–307, 1992.

[De Raedt, 1992] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press, 1992.

[Denecker and De Schreye, 1992] M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for normal abductive programs. In K.R. Apt, editor, *Proc. of the International Joint Conference and Symposium on Logic Programming*, pages 686–700, 1992.

[Denecker *et al.*, 1992] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 384–388. John Wiley & Sons, 1992.

[Denecker, 1993] M. Denecker. *Knowledge Representation and Reasoning in Incomplete Logic Programming.* PhD thesis, Department of Computer Science, K.U.Leuven, 1993.

[Guessoum and Lloyd, 1990] A. Guessoum and J.W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8:71–88, 1990.

[Kakas and Mancarella, 1990] A.C. Kakas and P. Mancarella. Database updates through abduction. In *Proc. of the 16th Very large Database Conference*, pages 650–661, 1990.

[Kakas *et al.*, 1993] A. C. Kakas, R.A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[Muggleton and De Raedt, 1994] S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[Muggleton and Feng, 1990] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st conference on algorithmic learning theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[Pazzani and Brunk, 1991] M. Pazzani and C. Brunk. Detecting and correcting errors in rule-based expert systems: an integration of empirical and explanation-based learning. *Knowledge Acquisition*, 3:157–173, 1991.

[Quinlan, 1990] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Shapiro, 1983] E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1983.

[Tankitvanitch and Shimura, 1992] S. Tankitvanitch and M. Shimura. Refining a relational theory with multiple faults in the concept and subconcepts. In *Proceedings of the 9th International Workshop on Machine Learning*, pages 436–444. Morgan Kaufmann, 1992.

[Taylor, 1993] K. Taylor. Inverse Resolution of Normal Clauses. In *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 165–177, 1993.