Toward an Improved Downward Refinement Operator for Inductive Logic Programming

S. Ferilli[1,2]

[1] Dipartimento di Informatica – Università di Bari
stefano.ferilli@uniba.it
[2] Centro Interdipartimentale per la Logica e sue Applicazioni – Università di Bari

Abstract. In real-world supervised Machine Learning tasks, the learned theory can be deemed as valid only until there is evidence to the contrary (i.e., new observations that are wrongly classified by the theory). In such a case, incremental approaches allow to revise the existing theory to account for the new evidence, instead of learning a new theory from scratch. In many cases, positive and negative examples are provided in a mixed and unpredictable order, which requires *generalization* and *specialization* refinement operators to be available for revising the hypotheses in the existing theory when it is inconsistent with the new examples. The space of Datalog Horn clauses under the OI assumption allows the existence of refinement operators that fulfill desirable properties. However, the versions of these operators currently available in the literature are not able to handle some refinement tasks. The objective of this work is paving the way for an improved version of the specialization operator, aimed at extending its applicability.

1 Introduction

Supervised Machine Learning approaches based on First-Order Logic representations are particularly indicated in real-world tasks in which the relationships among objects play a relevant role in the definition of the concepts of interest. Given an initial set of examples, a theory can be learned from them by providing a learning system with the whole 'batch' of examples. However, being inductive inference only falsity preserving, the learned theory can be deemed as valid only until there is evidence to the contrary (i.e., new observations that are wrongly classified by the theory). In such a case, either a new theory is to be learned from scratch using the new batch made up of both the old and the new examples, or the existing theory must be incrementally revised to account for the new evidence as well. To distinguish these two stages, we may call them 'training' and 'tuning', respectively. In extreme cases, the initial batch is not available at all, and learning must start and proceed incrementally from scratch. In many real cases, positive and negative examples are provided in a mixed and unpredictable order to the tuning phase, which requires two different refinement operator to be available for revising the hypotheses in the existing theory when it is inconsistent

with the new examples. A *generalization* operator is needed to refine a hypothesis that does not account for a positive example, while a *specialization* operator must be applied to refine a hypothesis that erroneously accounts for a negative example. So, the kind of modifications that are applied to the theory change its behavior non-monotonically. The research on incremental approaches is not very wide, due to the intrinsic complexity of learning in environments where the available information about the concepts to be learned is not completely known in advance, especially in a First-Order Logic (FOL) setting. Thus, the literature published some years ago still represents the state-of-the-art for several aspects of interest.

The focus of this paper is on supervised incremental inductive learning of logic theories from examples, and specifically on the extension of existing specialization operators. Indeed, while these operators have a satisfactory behavior when trying to add positive literals to a concept definition, the way they handle the addition of negative information has some shortcomings that, if solved, would allow a broader range of concepts to be learned. Here we point out these shortcomings, and propose both improvements of the existing operator definitions, and extensions to them. Theoretical results on the new version of the operator are sketched, and an algorithm for it is provided and commented. The solution has been implemented and embedded in the multistrategy incremental learning system InTheLEx [3]. The next section lays the logic framework in which we cast our proposal; then, Sections 3 and 4 introduce the learning framework in general and the state-of-the-art specialization operator for it in particular. Section 5 describes our new proposal, and finally Section 6 concludes the paper. Due to lack of space, proofs of theoretical results will not be provided.

2 Preliminaries

The logic framework in which we build our solution exploits *Datalog* [1, 4] as a representation language. Syntactically, it can be considered as a sublanguage of Prolog in which no function symbols are allowed. I.e., a Datalog term can only be a variable or a constant, which avoids potentially infinite nesting in terms and hence simplifies clause handling by the operators we will define in the following. The missing expressiveness of function symbols can be recovered by *Flattening* [9], a representational change that transforms a set of clauses containing function symbols into another, semantically equivalent to it, made up of function-free clauses[1]. In a nutshell, each n-ary function symbol is associated to a new $(n+1)$-ary predicate, where the added argument represents the function result. Functions are replaced, in the literals in which they appear, by variables or constants representing their result.

In the following, we will denote by $body(C)$ and $head(C)$ the set of literals in the *body* and the atom in the *head* of a Horn clause C, respectively. Pure

[1] Flattening potentially generates an infinite function free program. This is not our case, where we aim at learning from examples, and thus our universe is limited by what we see in the examples.

Datalog does not allow the use of negation in the body of clauses. A first way to overcome this limitation is the Closed World Assumption (CWA): if a fact does not logically follows from a set of Datalog clauses, then we assume its negation to be true. This allows the deduction of negative facts, but not their use to infer other information. Conversely, real world description often needs rules containing negative information. Datalog$^\neg$ allows to use negated literals in clauses body, at the cost of a further *safety condition*: each variable occurring in a negated literal must occur in another positive literal of the body too.

Since there can be several minimal Herbrand models instead of a single a least one, CWA cannot be used. In *Stratified* Datalog$^\neg$ this problem is solved by partitioning a program P in n sets P^i (called *layers*) s.t.:

1. all rules that define the same predicate in P are in the same layer;
2. P^1 contains only clauses without negated literals, or whose negated literals correspond to predicates defined by facts in the knowledge base;
3. each layer $P^i, i > 1$, contains only clauses whose negated literals are completely defined in *lower level* layers (i.e., layers P^j with $j < i$).

Such a partition is called *stratification*, and P is called *stratified*.
A stratified program P with stratification $P^1, \ldots, P^n$ is evaluated by growing layers, applying to each one CWA locally to the knowledge base made up by the original knowledge base and by all literals obtained by the evaluation of the previous layers.

There may be different stratifications for a given program, but all are equivalent as regards the evaluation result. Moreover, not all programs are stratified. The following notion allows to know if they are.

Definition 1 (Extended dependence graph) *Let P be a Datalog$^\neg$ program. The* extended dependence graph *of P, $EDG(P)$, is a directed graph whose nodes represent predicates defined by rules in P, and there is an edge $\langle p, q\rangle$ if q occurs in the body of a rule defining p.*
An edge $\langle p, q\rangle$ is labeled with $\neg$ if there exists at least one rule having p as its head and $\neg q$ in its body.

A program P is stratified if $EDG(P)$ contains no cycles containing edges marked with $\neg$. The evaluation of a stratified program produces a minimal Herbrand model, called *perfect* model.

A specific kind of negation is expressed by the inequality built-in predicate $\neq$. Using only this negation in Datalog yields an extension denoted by *Datalog*$^\neq$.

2.1 Object Identity

We deal with Datalog under the Object identity (OI) assumption, defined as follows:

Definition 2 (Object Identity)
Within a clause, terms denoted with different symbols must be distinct.

This notion is the basis for the definition of an equational theory for Datalog clauses that adds one rewrite rule to the set of the axioms of Clark's Equality Theory (CET) [6]:

$t \neq s \in body(C)$ **for each clause** C **in** L **and**
for all pairs t, s **of distinct terms that occur in** C **(OI)**

where L denotes the language that consists of all the possible Datalog clauses built from a finite number of predicates. The (OI) rewrite rule can be viewed as an extension of both Reiter's *unique-names* assumption [8] and axioms (7), (8) and (9) of CET to the variables of the language.

Datalog^{OI} is a sublanguage of Datalog$^{\neq}$ resulting from the application of OI to Datalog. Under OI, any Datalog clause C generates a new Datalog$^{\neq}$ clause C_{OI} consisting of two components, called *core* and *constraints*:

- $core(C_{OI}) = C$ and
- $constraints(C_{OI}) = \{t \neq s \mid t, s \in terms(C) \wedge t, s \text{ distinct}\}$ are the inequalities generated by the (OI) rewrite rule.

Formally, a Datalog^{OI} program is made up of a set of Datalog$^{\neq}$ clauses of the form

$$l_0 \ :- \ l_1, \ldots, l_n, c_1, \ldots, c_m$$

where the l_i's are as in Datalog, and the c_j's are the inequalities generated by the (OI) rule and $n \geq 0$. Nevertheless, Datalog^{OI} has the same expressive power as Datalog, that is, for any Datalog program we can find a Datalog^{OI} program equivalent to it [11].

2.2 θ_{OI}-subsumption

Applying the OI assumption to the representation language causes the classical ordering relations among clauses to be modified, thus yielding a new structure of the corresponding search spaces for the refinement operators.

The ordering relation defined by the notion of θ-subsumption under OI upon Datalog clauses [2, 10] is θ_{OI}-subsumption.

Definition 3 (θ_{OI}-subsumption ordering) *Let C, D be Datalog clauses. D* θ-subsumes C under OI *(D θ_{OI}-subsumes C), written $C \leq_{OI} D$, iff $\exists\sigma$ substitution s.t. $D_{OI}.\sigma \subseteq C_{OI}$. This means that D is* more general than or equivalent to C *(in a theory revision setting, D is an* upward refinement *of C and C is a* downward refinement *of D)* under OI. *$C <_{OI} D$ stands for $C \leq_{OI} D \wedge D \not\leq_{OI} C$. C and D are* equivalent under OI *($C \sim_{OI} D$) when $C \leq_{OI} D$ and $D \leq_{OI} C$.*

A substitution, as usual, is a mapping from variables to terms [13]. Its domain can be extended to terms: applying a substitution σ to a term t means that σ is applied to all variables in t. In our case, applying a substitution to a constant leaves it unchanged.

Like θ-subsumption, θ_{OI}-subsumption induces a quasi-ordering upon the space of Datalog clauses, as stated by the following result.

Proposition 1 *Let C, D, E be Datalog clauses. Then:*

1. $C \leq_{OI} C$ *(reflexivity)*
2. $C \leq_{OI} D$ *and* $D \leq_{OI} E \Rightarrow C \leq_{OI} E$ *(transitivity)*

Other interesting properties of θ_{OI}-subsumption are the following:

Proposition 2 *Let C, D be Datalog clauses.*

- $C \leq_{OI} D \Rightarrow C \leq_{\theta} D$, *where $\leq_{\theta}$ denotes θ-subsumption (i.e., θ_{OI}-subsumption is a weaker relation than θ-subsumption).*
- $C \leq_{OI} D \Rightarrow |C| \geq |D|$.
- $C \sim_{OI} D$ *iff C and D are renamings.*

Non-injective substitutions would yield contradictions when applied to constraints (e.g., $[x \neq y].\{x/a, y/a\} = [a \neq a]$). So, under OI, substitutions are required to be injective.

Requiring that terms are distinct 'freezes' the number of literals of the clause (since they cannot unify among each other), hence θ_{OI}-subsumption maps each literal of the subsuming clause onto a single, different literal in the subsumed one. In particular, equivalent clauses under $\leq_{OI}$ must have the same number of literals, hence the only way to have equivalence is through variable renaming. Thus, a search space ordered by θ_{OI}-subsumption is made up of *non-redundant* clauses, i.e. no subset of a clause can be equivalent to the clause itself under OI. This yields smaller equivalence classes than those in a space ordered by θ-subsumption.

Proposition 3 (Decidability of θ_{OI}-subsumption) *Given two clauses C and D, $C \leq_{OI} D$ is a decidable relationship.*

In the worst case, i.e. when $|D| \leq |C|$, all literals in D match with all literals in $|C|$, and all such matchings are pairwise compatible, an upper bound to the complexity of the θ_{OI}-subsumption test is $\binom{|C|}{|D|}$.

3 Incremental Inductive Synthesis

ILP aims at learning logic programs from examples. In our setting, examples are represented as clauses, whose body describes an observation, and whose head specifies a relationship to be learned, referred to terms in the body. Negative examples for a relationship have a negated head. A learned program is called a *theory*, and is made up of *hypotheses*, i.e. sets of program clauses all defining the same predicate. A hypothesis *covers* an example if the body of at least one of its clauses is satisfied by the body of the example. The *search space* is the set of all clauses that can be learned, ordered by a generalization relationship.

In ILP, a standard practice to restrict the search space is imposing biases on it [7]. In the following, we are concerned with logic theories expressed as *hierarchical* (i.e., non-recursive) programs, for which it is possible to find a *level*

mapping [6] s.t., in every program clause, the level of every predicate symbol occurring in the body is less than the level of the predicate in the head. This has strict connections with stratified programs, that are needed when the language is extended to deal with negation. Another bias on the representation language is that, whenever we write about clauses, we mean *Datalog linked* clauses. A clause is linked if, for any term appearing in its body, it also appears in the head or it is possible to find a chain of terms such that adjacent terms appear in the same literal and at least a term in the chain appears in the head.

The canonical inductive paradigm requires the learned theory to be complete and consistent. For hierarchical theories, the following definitions are given (where E^- and E^+ are the sets of all the negative and positive examples, resp.):

Definition 4 (Inconsistency)

- *A clause* C *is* inconsistent *wrt* $N \in E^-$ *iff* $\exists \sigma$ *s.t.*[2] $\quad body(C).\sigma \subseteq body(N) \wedge \neg head(C).\sigma = head(N) \wedge constraints(C_{OI}).\sigma \subseteq constraints(N_{OI})$
- *A hypothesis* H *is* inconsistent *wrt* N *iff* $\exists C \in H$*:* C *is inconsistent wrt* N*.*
- *A theory* T *is* inconsistent *iff* $\exists H \subseteq T$*,* $\exists N \in E^- : H$ *is inconsistent wrt* N*.*

Definition 5 (Incompleteness)

- *A* hypothesis H *is* incomplete *wrt* P *iff* $\forall C \in H$*: not*$(P \leq_{OI} C)$*.*
- *A* theory T *is* incomplete *iff* $\exists H \subseteq T$*,* $\exists P \in E^+$*:* H *is incomplete wrt* P*.*

When the theory is to be learned incrementally, it becomes relevant to define operators that allow a stepwise (incremental) refinement of *too weak* or *too strong* programs [5]. A *refinement operator*, applied to a clause, returns one of its upward or downward refinements. Refinement operators are the means by which wrong hypotheses in a logic theory are changed in order to account for new examples with which they are incomplete or inconsistent. In the following, we will assume that logic theories are made up of clauses that have only variables as terms, built starting from observations described as conjunctions of ground facts (i.e., variable-free atoms). This assumption causes no loss in expressive power, since a *reification* process allows to express through predicates all the information that may be carried out by constants. Put another way, we take to the extreme the flattening procedure, considering even constants as 0-ary functions that are replaced by 1-ary predicates having the same name and meaning. This restriction simplifies the refinement operators for a space ordered by θ_{OI}-subsumption defined in [2, 10], and the associated definitions and properties.

Definition 6 (Refinement operators under OI) *Let* C *be a Datalog clause.*

- $D \in \rho_{OI}(C)$ *(*downward *refinement operator) when* $body(D) = body(C) \cup \{l\}$*, where* l *is an atom s.t.* $l \notin body(C)$*.*
- $D \in \delta_{OI}(C)$ *(*upward *refinement operator) when* $body(D) = body(C) \setminus \{l\}$*, where* l *is an atom s.t.* $l \in body(C)$*.*

[2] $\neg head(C).\sigma = head(N)$ because the relationship must be the same as for N.

Particularly important are *locally finite*, *proper* and *complete* (*ideal*) refinement operators [10]. Such a kind of operators are not feasible when full Horn clause logic is chosen as representation language and either θ-subsumption or implication is adopted as generalization model, because of the existence of infinite unbound strictly ascending/descending chains. On the contrary, in the space of Datalog clauses ordered by θ_{OI}-subsumption such a kind of chains do not exist, since equivalence among clauses coincides with alphabetic variance. This makes possible the existence of ideal refinement operators under the ordering induced by θ_{OI}-subsumption [2, 10].

By the definition of δ_{OI}, any possible generalization of a clause must have as body a subset of its body, and hence there are $2^{|body(C)|}$ such generalizations.

Proposition 4 *[10] The refinement operators in Definition 6 are ideal for Datalog clauses ordered by θ_{OI}-subsumption.*

Inspired to a concept given by Shapiro [12], we have a measure for the complexity of a clause:

Definition 7 ($\mathbf{size}_{OI}$) *The* size *of a clause C under OI ($size_{OI}(C)$) is the number of literals in the body of C:*

$$size_{OI}(C) = \mid body(C) \mid$$

Under θ_{OI}-subsumption it allows to predict the exact number of steps required to perform a refinement, based only on the syntactic structure of the clauses involved (that could be known or bounded *a priori*): given two clauses C and D, if $D <_{OI} C$, then $C \in \delta^k_{OI}(D), D \in \rho^k_{OI}(C),\ k = size_{OI}(D) - size_{OI}(C) = |body(D)| - |body(C)|$

4 Downward Refinement

When a negative example is covered, a specialization of the theory must be performed. Starting from the current theory, the misclassified example and the set of processed examples, the specialization algorithm outputs a revised theory. In our framework, specializing means adding proper literals to a clause that is inconsistent with respect to a negative example, in order to avoid its covering that example. The possible options for choosing such a literal might be so large that an exhaustive search is not feasible. Thus, we want the operator to focus the search into the portion of the space of literals that contains the solution of the *diagnosed* commission error, as a result of an analysis of its algebraic structure.

According to the theoretical operator in Definition 6, only positive literals can be added. To this aim, we try to add to the clause one (or more) atom(s), which characterize all the past positive examples and can discriminate them from the current negative one. The search for such atoms is performed in the space of positive literals, that contains information coming from the positive examples used to learn the current theory, but not yet exploited by it. First of all, the process of abstract diagnosis detects all the clauses that caused the inconsistency. Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be the positive examples θ_{OI}-subsumed by

a clause C that is inconsistent wrt a negative example N. The set of all possible *most general downward refinements under OI* of C against N is:

$$mgdr_{OI}(C,N) = \{M \mid M \leq_{OI} C,\ M \text{ consistent wrt } N,\ \forall D \text{ s.t. } D \leq_{OI} C, D \text{ consistent wrt } N : not(M <_{OI} D)\}$$

Among these, the search process aims at finding one that is compliant with the previous positive examples in $\mathcal{P}$, i.e. one of the *most general downward refinements under OI* ($mgdr_{OI}$) of C against N given $P_1, \ldots, P_n$:

$$mgdr_{OI}(C, N \mid P_1, \ldots, P_n) = \{M \in mgdr_{OI}(C,N) \mid P_j \leq_{OI} M, j = 1, \ldots, n\}$$

Since the downward refinements we are looking for must satisfy the property of maximal generality, the operator tries to add as few atoms as possible. Thus, it may happen that, even after some refinement steps, the added atoms are still not sufficient to rule out the negative example, i.e. the specialization of C is still overly general. This suggests to further exploit the positive examples in order to specialize C. Specifically, if there exists a literal that, when added to the body of C, is able to discriminate from the negative example N that caused the inconsistency of C, then the downward refinement operator should be able to find it. The resulting specialization should restore the consistency of C, by refining it into a clause C' which still θ_{OI}-subsumes the positive examples P_i, $i = 1, 2, \ldots, n$.

The process of refining a clause by means of positive literals can be described as follows. For each P_i, $i = 1, 2, \ldots, n$, suppose that there exist n_i distinct substitutions s.t. C θ_{OI}-subsumes P_i, and consider all the possible n-tuples of substitutions obtained by picking one of such substitutions for every positive example. Each of these substitutions yields a distinct *residual*, consisting of all the literals in the example that are not involved in the θ_{OI}-subsumption test, after having properly turned their constants into variables. Formally:

Definition 8 (Residual) *Let C be a clause, E an example, and σ_j a substitution s.t. $body(C).\sigma_j \subseteq body(E)$ and $constraints(C_{OI}).\sigma_j \subseteq constraints(E_{OI})$. A* residual *of E wrt C under the mapping σ_j, denoted by $\Delta_j(E,C)$, is:*

$$\Delta_j(E,C) = body(E).\underline{\sigma_j}^{-1} - body(C)$$

where $\underline{\sigma_j}^{-1}$ is the *extended antisubstitution* of σ_j. An antisubstitution is a mapping from terms onto variables. When a clause C θ_{OI}-subsumes an example E through a substitution σ, then it is possible to define a corresponding antisubstitution, σ^{-1}, which is the inverse function of σ, mapping some constants in E to variables in C. Since not all constants in E have a corresponding variable according to σ^{-1}, we introduce the extension of σ^{-1}, denoted with $\underline{\sigma}^{-1}$, that is defined on the whole set $consts(E)$, and takes values in the set of the variables of the language[3]:

$$\underline{\sigma}^{-1}(c_n) = \begin{cases} \sigma^{-1}(c_n) & \text{if } c_n \in vars(C).\sigma \\ \_ & \text{otherwise} \end{cases}$$

[3] Variables denoted by _ are *new* variables, managed as in Prolog.

The residuals obtained from the positive examples $P_i, i = 1, \ldots, n$, can be exploited to build a *space of complete positive downward refinements*, denoted with $\mathbf{P}$, and formally defined as follows.

$$\mathbf{P} = \bigcup_{\substack{i=1,\ldots,n \\ j_i=1,\ldots,n_i}} \bigcap_{k=1,\ldots,n} \Delta_{j_k}(P_k, C)$$

where the symbol $\Delta_{j_k}(P_k, C)$ denotes one of the n_k residuals of P_k wrt C, and $\cap_{k=1,\ldots,n}\Delta_{j_k}(P_k, C)$, when $j_k \in \{1, \ldots, n_k\}$, is the set of the literals common to an n-tuple of residuals (one residual for each positive example P_k, $k = 1, \ldots, n$). Moreover, denoted with θ_j, $j = 1, \ldots, m$, all the substitutions which make C inconsistent wrt N, let us define a new space:

$$\mathbf{S} = \bigcup_{j=1,\ldots,m} \Delta_j(N, C)$$

which includes all the literals that cannot be used for refining C, because they would still be present in N.

Proposition 5 *Given a clause C that θ_{OI}-subsumes the positive examples $P_1, \ldots, P_n$ and is inconsistent wrt the negative example N, then:*
$\{C' \mid head(C') = head(C) \wedge body(C') = body(C) \cup \{l\}, l \in \mathbf{P} - \mathbf{S}\} \subseteq$
$\subseteq mgdr_{OI}(C, N \mid P_1, \ldots, P_n)$

Hence, every downward refinement built by adding a literal in $\mathbf{P} - \mathbf{S}$ to the inconsistent clause C restores the properties of consistency and completeness of the original hypothesis. Moreover, it is one of the most general downward refinements of C against N.

It may happen that no (set of) positive literal(s) is able to characterize the past positive examples and discriminate the negative example that causes inconsistency. In such a case, the above version of the operator would fail. However, we don't want to give up yet, since the addition of a negative literal to the clause body might restore consistency[4]. To take this opportunity, we extend the search space to Datalog$^\neg$. These literals are interpreted according to the CWA. Of course, suitable adaptations of the notions presented in Section 2 are used to handle these literals[5]. So, in case of failure on the search for positive literals, the algorithm autonomously performs a *representation change*, that allows it to extend the search to the space of negative literals, built by taking into account the negative example that caused the commission error. The new version of the operator tries to add the negation of a literal, that is able to discriminate the negative example from all the past positive ones. Revisions performed by this

[4] Note that a negative literal in the body corresponds to a positive literal in the clause. However, here we are expressing the fact that a condition must not hold in an observation in order to infer the relationship in the head.

[5] E.g., given two clauses under OI, $C = C^+ \cup C^-$ and $D = D^+ \cup D^-$, where C^+ and D^+ include the positive literals and the OI-constraints, and C^- and D^- are sets of negative literals, $C \leq_{OI} D$ iff $\exists\sigma$ substitution s.t. $D^+.\sigma \subseteq C^+$ and $\forall d \in D^- : \nexists c \in C^-$ s.t. $d.\sigma = c$.

operator are always minimal [14], since all clauses in the theory contain only variables as arguments. Moreover, this operator is ideal in the space of constant-free clauses. The definitions and results in the rest of this section are taken from [2].

When the space $\mathbf{P} - \mathbf{S}$ does not contain any solution to the problem of specializing an inconsistent clause, a change of representation must be performed in order to search for literals in another space, corresponding to the quotient set of the Datalog$^\neg$ linked clauses. In the following, a slightly different but equivalent specification of the operator in this space will be provided with respect to [2].

First of all, we define the new target space, called the *space of negative downward refinements*:

$$\mathbf{S}_n = \neg\mathbf{S} = \neg(\cup_{j=1,\ldots,m} \Delta_j(N, C))$$

where, given a set of literals $\varphi = \{l_1, \ldots, l_n\}$, $n \geq 1$: $\neg\varphi = \{\neg l_1, \ldots, \neg l_n\}$. Again, we are interested in a specific subset of $\mathbf{S}_n$, because of the properties satisfied by its elements. Let us introduce the following notation:

$$\overline{\mathbf{S}} = \bigcap_{j=1,\ldots,m} \Delta_j(N, C)$$

Note that $\overline{\mathbf{S}} \subseteq \mathbf{S}$. Based on $\overline{\mathbf{S}}$, the *space of consistent negative downward refinements* can be defined as:

$$\mathbf{S}_c = \neg\overline{\mathbf{S}} = \neg(\cap_{j=1,\ldots,m} \Delta_j(N, C))$$

Indeed, $\mathbf{S}_c$, compared to $\mathbf{S}_n$, fulfills the following property:

Proposition 6 *Given a clause C and an example N, then:*
$\{C' \mid head(C') = head(C) \wedge body(C') = body(C) \cup \{l\}, l \in \mathbf{S}_c\} \subseteq mgdr_{OI}(C, N)$

Overall, the search for a complete and consistent hypothesis can be viewed as a two-stage process: the former stage searches into the space $\mathbf{P} - \mathbf{S}$, the latter into $\mathbf{S}_c$. It is now possible to formally define the downward refinement operator ρ_{OI}^{cons} on the space L of constant-free Datalog^{OI} linked program clauses.

Definition 9 (ρ_{OI}^{cons}) $\rho_{OI}^{cons} : L \to 2^L \forall C \in L : \rho_{OI}^{cons}(C) =$
$\{C' \mid head(C') = head(C) \wedge body(C') = body(C) \cup \{l\},\ l \in (\mathbf{P} - \mathbf{S}) \cup \mathbf{S}_c\}$

Proposition 7 *The downward refinement operator ρ_{OI}^{cons} is ideal.*

The ideality of ρ_{OI}^{cons} is owed to the peculiar structure of the search space when ordered by the relation $\leq_{OI}$.

5 Discussion and Extension of the Specialization Operator

The existing definition of ρ_{OI}^{cons} aims at identifying a set of literals each of which, when added to a clause C, yields a new clause that is both consistent with the given negative example and complete with respect to all the previous positive examples. Now, this is true if the added literal belongs to the space of complete

positive downward refinements $\mathbf{P}-\mathbf{S}$. Conversely, the *space of consistent negative downward refinements* does not ensure completeness wrt the previous positive examples, since it is computed considering only all possible residuals of the negative example. This can be easily shown in the following example[6].

Example 1. Consider the following situation.
Two positive examples: $P_1 = h :- p, q, t, u.$ and $P_2 = h :- p, q, v.$
produce as a least general generalization the clause $C_1 = h :- p, q.$
Then, the negative example $N_1 = h :- p, q, t, u, v.$ arrives.
The residuals of P_1 and P_2 wrt C_1 are $\{t, u\}$ and $\{v\}$, respectively.
The residual of N_1 is $\{t, u, v\}$. So, $\mathbf{P} - \mathbf{S} = (\{t, u\} \cap \{v\}) - \{t, u, v\} = \emptyset - \{t, u, v\} = \emptyset$, hence no specialization by means of positive literals can be obtained (as expected, since C was a least general generalization). Switching to the space of negative literals, we have that $\mathbf{S}_c = \neg(\{t, u, v\}) = \{\neg t, \neg u, \neg v\}$. However, none of these literals generates a clause that is complete with all previous positive examples:
$C_2' = h :- p, q, \neg t.$ where $P_1 \not\leq_{OI} C_2'$
$C_2'' = h :- p, q, \neg u.$ where $P_1 \not\leq_{OI} C_2''$
$C_2''' = h :- p, q, \neg v.$ where $P_2 \not\leq_{OI} C_2'''$

So, what we need to consider is not $\mathbf{S}_c$. Intuitively, we want to select a literal that is present in all residuals of the negative example and that is not present in any residual of any positive example. Let us define:

$$\overline{\mathbf{P}} = \bigcup_{\substack{i=1,\ldots,n \\ j_i=1,\ldots,n_i}} \Delta_{j_i}(P_i, C)$$

Now, what we need to consider is $\mathbf{S}_c' = \neg(\overline{\mathbf{S}} - \overline{\mathbf{P}})$.

Example 2. In the previous example, we would have $\mathbf{S}_c' = (\{t, u, v\}) - (\{t, u\} \cup \{v\}) = \{t, u, v\} - \{t, u, v\} = \emptyset$ which shows, as expected, that no complete refinement can be obtained for the given case.

Consider now another set of positive examples: $P_1 = h :- p, q, t, u.$, $P_2 = h :- p, q, r.$ and $P_3 = h :- p, q, s, t.$
whose least general generalization is, again, the clause $C_1 = h :- p, q.$
Then, the negative example $N_1 = h :- p, q, t, u, v, w.$ arrives.
The residuals of the positive examples are: $\Delta(C_1, P_1) = \{t, u\}$, $\Delta(C_1, P_2) = \{r\}$ and $\Delta(C_1, P_3) = \{s, t\}$.
The residual of N_1 is $\{t, u, v, w\} = \mathbf{S}$. So, $\mathbf{P} - \mathbf{S} = (\{t, u\} \cap \{r\} \cap \{s, t\}) - \{t, u, v, w\} = \emptyset - \{t, u, v, w\} = \emptyset$, hence again no specialization by means of positive literals can be obtained. Switching to the space of negative literals, we have that $\mathbf{S}_c' = \neg(\overline{\mathbf{S}} - \overline{\mathbf{P}}) = \neg(\{t, u, v, w\} - (\{t, u\} \cup \{r\} \cup \{s, t\})) = \neg(\{t, u, v, w\} -$

[6] For the sake of readability, in the following we will often switch to a propositional representation. This means that the residual is unique for each example, so the subscript in $\Delta_i(\cdot, \cdot)$ is no more necessary.

$\{r, s, t, u\}) = \neg(\{v, w\}) = \{\neg v, \neg w\}$ and indeed, adding any of these literals to C generates a clause that is complete with all previous positive examples:
$C'_2 = h :- p, q, \neg v.$ where $P_1 \leq_{OI} C'_2$, $P_2 \leq_{OI} C'_2$ and $P_3 \leq_{OI} C'_2$;
$C''_2 = h :- p, q, \neg w.$ where $P_1 \leq_{OI} C''_2$, $P_2 \leq_{OI} C''_2$, $P_3 \leq_{OI} C''_2$.

This is captured by the following result:

Proposition 8 *Given a clause $C = h :- body(C)$ that θ_{OI}-subsumes the positive examples $P_1, \ldots, P_n$ and is inconsistent wrt the negative example N, then:*
$\{C' \mid head(C') = head(C) \wedge body(C') = body(C) \cup \{l\}, l \in \mathbf{S}'_c\} \subseteq$
$\subseteq mgdr_{OI}(C, N \mid P_1, \ldots, P_n)$

Now, an additional problem arises. Indeed, in some cases a single negative literal is not enough to ensure that the correctness of the theory is restored. The situation may be clarified by the following example.

Example 3. Consider again the situation described in Example 1.

While no single (positive or negative) literal can restore completeness and consistency of the theory, either $C'_2 = h :- p, q, \neg(t, v).$ or $C''_2 = h :- p, q, \neg(u, v).$ would be correct refinements of C_1 wrt $\{P_1, P_2, N_1\}$. These solutions are not permitted in the representation language, since only literals may appear in the body of clauses. However, any of the two above clauses corresponds to the conjunction of two clauses, e.g. C'_2 is equivalent to $\{h :- p, q, \neg t., h :- p, q, \neg v.\}$. This solution would introduce some redundancy in the theory, since the body of the original clause C_1 would appear in both specialized clauses. This might be undesirable, in which case we may leverage Datalog implication and solve the problem by inventing a new predicate s as follows: $\{h :- p, q, \neg(s)., s :- t, v.\}$. The intuition behind this choice is that the need to place together the literals in the negation might be a hint of a more general relationship among them. This relationship might be captured by a so far unknown concept, that is explicitly added. A useful side effect of this setting is that when the same combination will occur in future observations, it will be recognized and explicitly added by saturation, this way obtaining higher level descriptions.

So, the extension comes into play when no single literal is sufficient to restore correctness of the theory. Indeed, when a single literal is to be negated there is no need for inventing any predicate. More formally, we are not looking anymore for a single $l \in \mathbf{S}'_c$ to be added to C, but we need a $S \subseteq \mathbf{S}'_c$ s.t. $\forall i : \exists l \in S$ s.t. $l \notin P_i$. In particular, we would like to find a minimal such set. Minimality may be in terms of set inclusion or of number of elements: $\overline{S} = \arg\min_S(|S|)$. To formally express our operator, let us define:

- $\forall r \in \Delta(N, C)$:
  - $\mathcal{P}_r = \{P_i \in \mathcal{P} | r \in \Delta(P_i, C)\}$
  - $\overline{\mathcal{P}_r} = \{P_i \in \mathcal{P} | r \notin \Delta(P_i, C)\}$
- $\forall S \subseteq \Delta(N, C)$:
  - $\mathcal{P}_S = \cap_{r \in S} \mathcal{P}_r$
  - $\overline{\mathcal{P}_S} = \cup_{r \in S} \overline{\mathcal{P}_r}$

$\mathcal{P}_r$ is the set of positive examples that are no more covered when adding $\neg r$ to C; $\overline{\mathcal{P}_r}$ is the set of positive examples that are still covered when adding $\neg r$ to C. $\mathcal{P}_S$ is the set of positive examples that are no more covered when adding $neg(S)$ to C; $\overline{\mathcal{P}_S}$ is the set of positive examples that are still covered when adding $neg(S)$ to C. This helps us to define what we are looking for. Specifically, we need a $S \subseteq \Delta(N, C)$ s.t. $\mathcal{P}_S = \emptyset \wedge \overline{\mathcal{P}_S} = \mathcal{P}$, as shown by the following example:

Example 4. Consider the set of positive examples $\mathcal{P} = \{P_1, P_2, P_3\}$ where:
$P_1 = h : -p, q, s, t.$ $P_2 = h : -p, q, t, u.$ $P_3 = h : -p, q, u, r.$
Given their least general generalization $C = h : -p, q.$, the corresponding residuals are:
$\Delta(P_1, C) = \{s, t\}$ $\Delta(P_2, C) = \{t, u\}$ $\Delta(P_3, C) = \{u, r\}$
Now, given the negative example $N = h : -p, q, s, t, u, r$ covered by C, with residual $\Delta(N, C) = \{s, t, u, r\}$, we have:
$\mathcal{P}_{\{s,u\}} = \mathcal{P}_s \cap \mathcal{P}_u = \{P_1\} \cap \{P_2, P_3\} = \emptyset$; $\overline{\mathcal{P}_{\{s,u\}}} = \overline{\mathcal{P}_s} \cap \overline{\mathcal{P}_u} = \{P_2, P_3\} \cup \{P_1\} = \mathcal{P}$: SOLUTION!
$\mathcal{P}_{\{t,r\}} = \mathcal{P}_t \cap \mathcal{P}_r = \{P_1, P_2\} \cap \{P_3\} = \emptyset$; $\overline{\mathcal{P}_{\{t,r\}}} = \overline{\mathcal{P}_t} \cap \overline{\mathcal{P}_r} = \{P_3\} \cup \{P_1, P_2\} = \mathcal{P}$: SOLUTION!
$\mathcal{P}_{\{t,u\}} = \mathcal{P}_t \cap \mathcal{P}_u = \{P_1, P_2\} \cap \{P_2, P_3\} = \{P_2\} \neq \emptyset$; $\overline{\mathcal{P}_{\{t,u\}}} = \overline{\mathcal{P}_t} \cap \overline{\mathcal{P}_u} = \{P_3\} \cup \{P_1\} = \{P_3, P_1\} \neq \mathcal{P}$: NOT A SOLUTION!
. . . and so on.

Of course, a trial-and-error approach would solve the problem, but there is an exponential number of subsets to be tried. In order to devise a more efficient algorithm, let us analyze the sets $\mathcal{P}_r$, $\overline{\mathcal{P}_r}$, $\mathcal{P}_S$ and $\overline{\mathcal{P}_S}$ to better understand them and their behavior. First of all, the $\mathcal{P}_x$'s and $\overline{\mathcal{P}_x}$'s are complementary:

Proposition 9 *Given a clause C and a negative example N covered by C:*

1. $\forall r \in \Delta(N, C) : \{\mathcal{P}_r, \overline{\mathcal{P}_r}\}$ *is a partition of* $\mathcal{P}$;
2. $\forall S \subseteq \Delta(N, C) : \{\mathcal{P}_S, \overline{\mathcal{P}_S}\}$ *is a partition of* $\mathcal{P}$.

This ensures, in particular, that $\mathcal{P}_S = \emptyset \wedge \overline{\mathcal{P}_S} = \mathcal{P} \Leftrightarrow \mathcal{P}_S = \emptyset \Leftrightarrow \overline{\mathcal{P}_S} = \mathcal{P}$. We also note that positive example (un-)coverage is monotonic:

Proposition 10 $\forall S' \subset S'' \subseteq \Delta(N, C) : \mathcal{P}_{S''} \subseteq \mathcal{P}_{S'} \wedge \overline{\mathcal{P}_{S'}} \subseteq \overline{\mathcal{P}_{S''}}$

Finally, let us note that any element of the residual of the negative example, added to C, causes some positive example to become uncovered (which will be used in the first iteration of our algorithm):

Proposition 11 *If ρ_{OI}^{cons} fails, then* $\forall r \in \Delta(N, C) : \mathcal{P}_r \neq \emptyset$.

We propose a sequential covering-like strategy to find such an S, according to Algorithm 1. Note that, at the beginning of the algorithm, $\mathcal{S} = \emptyset \Rightarrow \mathcal{P}_{\mathcal{S}} = \emptyset \Rightarrow |\mathcal{P}_{\mathcal{S}}| = 0$ and $\mathcal{S} = \emptyset \Rightarrow \overline{\mathcal{P}_{\mathcal{S}}} = \mathcal{P} \Rightarrow |\overline{\mathcal{P}_{\mathcal{S}}}| = n$. However, as soon as the loop is entered, the selection and addition of the first r makes $\mathcal{P} \neq \emptyset$ by Proposition 11; so, the condition of the IF statement is true, hence $\mathcal{S}$ is updated and a second round of the loop is guaranteed to take place. At each round, a new

Algorithm 1 Backtracking specialization strategy

$\mathcal{S} = \emptyset$; $\mathcal{R} = \Delta(N, C)$
repeat
  $r \leftarrow$ select from $\mathcal{R}$ OR *backtrack*
  $\mathcal{R} \leftarrow \mathcal{R} \setminus \{r\}$; $\mathcal{S}' \leftarrow \mathcal{S} \cup \{r\}$
  if $\mathcal{P}_{\mathcal{S}'} \neq \mathcal{P}_{\mathcal{S}}$ ($\Leftrightarrow \overline{\mathcal{P}_{\mathcal{S}'}} \neq \overline{\mathcal{P}_{\mathcal{S}}}$) ($\Leftrightarrow \mathcal{P}_{\mathcal{S}'} \subset \mathcal{P}_{\mathcal{S}} \Leftrightarrow \overline{\mathcal{P}_{\mathcal{S}'}} \supset \overline{\mathcal{P}_{\mathcal{S}}}$) **then**
    $\mathcal{S} \leftarrow \mathcal{S}'$
  end if
until $\mathcal{P}_{\mathcal{S}} = \emptyset$ ($\Leftrightarrow \overline{\mathcal{P}_{\mathcal{S}}} = \mathcal{P}$) OR no more backtracking available
if $\mathcal{P}_{\mathcal{S}} = \emptyset$ ($\Leftrightarrow \overline{\mathcal{P}_{\mathcal{S}}} = \mathcal{P}$) **then**
  return $\mathcal{S}$
else
  return *failure*
end if

r is removed from $\mathcal{R}$ and added to $\mathcal{S}$ only if the coverage improves, otherwise it is discarded. If the last r does not satisfy the loop condition, the overall solution is not complete and backtracking is applied. Note that, in the worst case, adding the whole residual to C would be a solution, which ensures termination of the algorithm (unless there is a positive example that includes the whole residual, which can be checked before starting the algorithm). If different solutions are requested, backtracking can be applied to non-discarded items.

6 Conclusions and Future Work

Incremental supervised Machine Learning approaches using First-Order Logic representations are mandatory when tackling complex real-world tasks, in which relationships among objects play a fundamental role. A noteworthy framework for these approaches is based on the space of Datalog Horn clauses under the Object Identity assumption, which ensures the existence of (upward and downward) refinement operators fulfilling desirable requirements. The refinement operators for this framework proposed in the current literature have some limitations that this paper aims at overcoming. So, after recalling the most important elements of the framework and of the current operators, this paper points out these deficiencies and proposes solutions that result in improved operators. Specifically, the downward refinement operator is considered. A preliminary prototype of the operator has been implemented, and is currently being integrated in the InTheLEx learning system.

Future work includes a study of the possible connections of the extended operator with related fields of the logic-based learning, such as deduction, abstraction and predicate invention. Experiments aimed at assessing the efficiency and effectiveness of the operator in real-world domains are also planned.

Acknowledgments

This work was partially funded by the Italian PON 2007-2013 project PON02_00563_3489339 'Puglia@Service'.

References

[1] S. Ceri, G. Gottlöb, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, Heidelberg, Germany, 1990.

[2] F. Esposito, A. Laterza, D. Malerba, and G. Semeraro. Locally finite, proper and complete operators for refining datalog programs. In Z. W. Raś and M. Michalewicz, editors, *Foundations of Intelligent Systems*, number 1079 in Lecture Notes in Artificial Intelligence, pages 468–478. Springer, 1996.

[3] F. Esposito, G. Semeraro, N. Fanizzi, and S. Ferilli. Multistrategy Theory Revision: Induction and abduction in INTHELEX. *Machine Learning Journal*, 38(1/2):133–156, 2000.

[4] P. C. Kanellakis. Elements of relational database theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B — Formal Models and Semantics, pages 1073–1156. Elsevier Science Publishers, 1990.

[5] H. J. Komorowski and S. Trcek. Towards refinement of definite logic programs. In Z. W. Raś and M. Zemankova, editors, *Methodologies for Intelligent Systems*, number 869 in Lecture Notes in Artificial Intelligence, pages 315–325, Berlin, 1994. Springer-Verlag.

[6] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, second edition, 1987.

[7] C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. de Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, Amsterdam, NL, 1996.

[8] R. Reiter. Equality and domain closure in first order databases. *Journal of the ACM*, 27:235–249, 1980.

[9] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. In *Inductive Logic Programming*, pages 64–90. Academic Press, 1992.

[10] G. Semeraro, F. Esposito, and D. Malerba. Ideal refinement of datalog programs. In M. Proietti, editor, *Logic Program Synthesis and Transformation*, number 1048 in Lecture Notes in Computer Science, pages 120–136. Springer-Verlag, 1996.

[11] G. Semeraro, F. Esposito, D. Malerba, N. Fanizzi, and S. Ferilli. A logic framework for the incremental inductive synthesis of datalog theories. In N. E. Fuchs, editor, *Logic Program Synthesis and Transformation*, number 1463 in Lecture Notes in Computer Science, pages 300–321. Springer-Verlag, 1998.

[12] E.Y. Shapiro. Inductive inference of theories from facts. Technical Report Research Report 192, Yale University, 1981.

[13] J. H. Siekmann. An introduction to unification theory. In R. B. Banerji, editor, *Formal Techniques in Artificial Intelligence - A Sourcebook*, pages 460–464. Elsevier Science Publisher, 1990.

[14] S. Wrobel. *Concept Formation and Knowledge Revision*. Kluwer Academic Publishers, Dordrecht Boston London, 1994.