Adopting an Object-Oriented Data Model in Inductive Logic Programming

M. Milano and **A. Omicini** and **F. Riguzzi**
LIA - DEIS - Università di Bologna
Viale Risorgimento, 2 40136 – Bologna, Italy
{mmilano,aomicini,friguzzi}@deis.unibo.it

Abstract

The increasing amount of information to be managed in knowledge-based systems has promoted, on one hand, the exploitation of machine learning for the automated acquisition of knowledge and, on the other hand, the adoption of object-oriented representation models for easing the maintenance. In this context, adopting techniques for structuring knowledge representation in machine learning seems particularly appealing.

Inductive Logic Programming (ILP) is a promising approach for the automated discovery of rules in knowledge based systems. We propose an object-oriented extension of ILP employing multi-theory logic programs as the representation language. We define a new learning problem and propose the corresponding learning algorithm. Our approach enables ILP to benefit of object-oriented domain modelling in the learning process, such as allowing structured domains to be directly mapped onto program constructs, or easing the management of large knowledge bases.

Introduction

As the application of knowledge-based systems in real world situations is becoming more and more common, the amount of knowledge that must be acquired and maintained is growing larger and larger. In order to manage knowledge bases of considerable size representing real world domains, object-based representations such as frames (Minsky 1975), description logic (Woods & Schmolze 1991), semantic networks (Brachman 1979), inheritance networks (Touretzky, Horty, & Thomason 1988) have been proven to be effective techniques. In these formalisms, each object in the domain is directly represented by means of appropriate syntactic structures, and objects sharing similar properties are grouped into classes. On their turn, classes are organised in a hierarchy and inheritance is exploited in order to represent only once properties that are shared by all objects of a class.

Inductive Logic Programming (ILP) (Muggleton 1991) is a promising approach for the automated discovery of rules in knowledge based systems (Morik *et al.* 1993). Therefore, it seems appealing to adopt techniques for structuring knowledge representation in ILP. ILP is concerned with the problem of learning logic programs from a background knowledge and a set of positive and negative examples. However, logic programming does not provide any support for structuring domain representation.

In this paper, we aim at investigating the impact of an object-oriented data model on ILP. By employing a *multi-theory logic language* (Bugliesi, Lamma, & Mello 1994; McCabe 1992), the knowledge is structured as a collection of separate logic theories organised in a hierarchy. Each node in the hierarchy represents a class of objects while each leaf represents a single object.

The contribution of this paper is twofold: (*i*) the definition of a learning problem in a multi-theory environment; (*ii*) the description of the corresponding learning algorithm.

The advantages of adopting a multi-theory logic language are twofold: from the knowledge representation viewpoint and from the learning process viewpoint. As regards the representation of the domain knowledge, structured domains can be directly mapped onto program constructs. The standard ILP approach does not allow the notions of *class* and *attribute* to be distinguished, since they are uniformly represented by means of predicate symbols. An object-oriented data model, instead, provides intrinsic support for the notion of class and class hierarchy, whereas object's attributes are represented as predicates. Moreover, inheritance provides an efficient and compact way of representing properties shared by a set of objects.

As regards the learning process, each class constitutes a natural boundary for the learning process, and provides it with a finer granularity level. Inheritance mechanisms provide for new generalisation and specialisation operators and can be exploited both in the learning process, and when using the learned program.

Preliminaries

In this section, we briefly recall some basic concepts of ILP and multi-theory logic languages.

Inductive Logic Programming

We first give a definition of the ILP problem, adapted from (Bergadano & Gunetti 1996):

Given:
a set E^+ of positive examples
a set E^- of negative examples
a consistent logic program B (*background knowledge*)
Find:
a logic program P (*target program*) such that
$\forall e^+ \in E^+$, $B \cup P \models e^+$ (*completeness*)
$\forall e^- \in E^-$, $B \cup P \not\models e^-$ (*consistency*)

We say that P *covers* example e if $P \cup B \models e$.

The problem of finding a solution to the ILP problem can be seen as a problem of search in the space of logic clauses. This space is (partially) ordered by means of the following generality relation.

Definition 1. (θ*-subsumption*) Clause C θ*-subsumes* D if there exist a substitution θ such that $C\theta \subseteq D$. We write $C \preceq D$. □

We exploit a bottom-up (i.e., from specific to general) operator similar to *relative least general generalisation* (*rlgg*) (Plotkin 1970) and a learning algorithm similar to the system GOLEM (Muggleton & Feng 1990). GOLEM learns theories bottom-up by using *rlgg*, i.e., an extension of the *least general generalisation* (*lgg*) taking into account the background knowledge.

Definition 2. (*least general generalisation*) We say that clause C is the least general generalisation (*lgg*) of clauses C_1 and C_2 if $C \preceq C_1$, $C \preceq C_2$ and, for every clause E such that $E \preceq C_1$ and $E \preceq C_2$, it holds that $E \preceq C$. □

GOLEM generates a single clause by randomly picking couples of examples, by computing their *rlgg*, and by choosing the one with the greatest coverage of other positive examples. This clause is then generalised by randomly choosing new uncovered positive examples and by computing the *rlgg* of the clause and each of the examples. Among the resulting clauses, the one that covers more examples is chosen and generalised again until either the coverage of the clause can not be further extended, or any further generalisation would cover some negative examples. Then, a post-processing phase follows, where irrelevant literals are discarded. In case there is not a single clause that covers all the positive examples, the procedure is iterated until no uncovered positive example remains.

Multi-theory logic languages

Multi-theory logic languages (Bugliesi, Lamma, & Mello 1994) extend the standard logic programming paradigm by partitioning a logic program into a multiplicity of logic theories. In this paper, we exploit a simple multi-theory first-order logic language (Omicini 1996), called $\mathcal{L}_{ind}$, for domain representation. A program of $\mathcal{L}_{ind}$ is a collection of logic theories connected by *parent* relations denoted as $Child \Vdash\!\!< Parent$, where theory $Parent$ is the (only) *parent theory* of theory $Child$. Each *theory* of $\mathcal{L}_{ind}$, denoted by a ground term, is a collection of labelled clauses. A *labelled clause* is a relation of the form

$$Theory\!:\!Head :- Body$$

representing a clause $Head :- Body$ of the theory denoted by the ground term $Theory$, where $Head$ is an atomic formula and $Body$ is a goal formula. An *atomic formula* of $\mathcal{L}_{ind}$ has the form $p(\tilde{t})$, where p is an n-ary predicate symbol, and $\tilde{t}$ an n-tuple of terms. A *goal formula* is either an atomic formula, a provability formula, a hierarchical formula, or a conjunction of goal formulae. A *provability formula* has the form $o \not\vdash g$, where o is a term denoting a theory, and g is a goal formula. Roughly speaking, $o \not\vdash g$ is true when g can be derived from theory o. A *hierarchical formula* has the form $o \mid\!< o'$, where o and o' are terms denoting theories. $o \mid\!< o'$ is true when theory o' is an ancestor of theory of o ($\mid\!<$ is equivalent to the transitive closure of $\Vdash\!\!<$).

A $\mathcal{L}_{ind}$ program $\mathcal{P}$ can be seen as a pair $\langle T_{\mathcal{P}}, IsA_{\mathcal{P}}\rangle$, where $T_{\mathcal{P}}$ is a collection of labelled clauses, and $IsA_{\mathcal{P}}$ is a set of *parent* relations. $IsA_{\mathcal{P}}$ defines a tree whose root is theory $\emptyset_{th}$, denoted by the constant *root* in any $\mathcal{L}_{ind}$ program, which is the only one with no parent theory. Leaves of the tree are called *instances*, i.e., theories with no descendant theories. $IsA_{\mathcal{P}}$ associates a *context* to every theory of $\mathcal{P}$. If $t_1, \ldots, t_n \in \mathcal{H}_{\mathcal{P}}$ (where $\mathcal{H}_{\mathcal{P}}$ denotes the Herbrand Universe of $\mathcal{P}$) such that $t_{i+1} \Vdash\!\!< t_i \in IsA_{\mathcal{P}}$, $1 \leq i < n$ and $t_1 \Vdash\!\!< \emptyset_{th} \in IsA_{\mathcal{P}}$, then the whole information about the object denoted by t_n is given by its context $\langle t_n, \ldots, t_1\rangle$, denoted by ctx_{t_n}, or by t_n itself, whenever no misunderstanding can arise. A context is a logic theory obtained as the union of the clauses of its component theories.

The entailment relation, $\models_{ctx}$, determines the truth value of a formula w.r.t. a context. Since a context is actually an ordinary first-order logic theory, context entailment exactly matches classical LP entailment. Obviously, we have to add the following definitions for provability and hierarchical formulae, where o and o' are theory identifiers:

$$o \models_{ctx} o' \not\vdash g \quad \text{iff} \quad o' \models_{ctx} g$$

$$o \models_{ctx} t \mid\!< t' \quad \text{iff} \quad IsA_{\mathcal{P}} \models t \overset{\star}{\Vdash\!\!<} t'$$

The definition of the entailment relation for $\mathcal{L}_{ind}$ (denoted with $\models_M$) is obviously based on $\models_{ctx}$. Given a program $\mathcal{P}$ and a formula g of $\mathcal{L}_{ind}$,

$$\mathcal{P} \models_M g \quad \text{iff} \quad \emptyset_{th} \models_{ctx} g$$

where $\emptyset_{th}$ represents the empty context (i.e., the empty logic theory) where each formula of $\mathcal{L}_{ind}$ is by default evaluated. A proof procedure exists that is sound with respect to $\models_M$. For a formal description of multi-theory entailment and the proof procedure see (Omicini 1996).

Object-Oriented Modelling and ILP

In this section, we first formally define the learning problem in a multi-theory logic framework, then provide a learning algorithm for this framework.

Problem Definition

In the following, we define the learning problem in a multi-theory environment. Let B and P be consistent $\mathcal{L}_{ind}$ programs, E^+ and E^- are instance properties represented by $\mathcal{L}_{ind}$ facts, such that $\forall o : e \in E^+ \cup E^-$, o denotes an instance theory. The problem can then be formulated as follows.

Given:
a set E^+ of positive examples
a set E^- of negative examples
a background knowledge $B = \langle T_B, IsA_B \rangle$
Find:
a program $P = \langle T_P, IsA_B \rangle$ such that
$\forall o : e^+ \in E^+$, $\langle T_B \cup T_P, IsA_B \rangle \models_M o \not\vdash e^+$
$\forall o : e^- \in E^-$, $\langle T_B \cup T_P, IsA_B \rangle \not\models_M o \not\vdash e^-$

In the problem definition we consider, the set $IsA_\mathcal{P}$ does not change since we cannot learn new *parent* relations, i.e., the class hierarchy is fixed. An interesting extension of our framework considers also the learning of *parent* relations and of new theories.

```
procedure LearnInstance($p/n, Theory; H, E^-$);
    let $E^+_{Theory}$ and $E^-_{Theory}$ be
        the positive and negative examples for $p/n$
            in $Theory$
    $H := \emptyset$
    while $E^+_{Theory}$ is not empty do
        pick randomly $m$ couples of positive examples
        compute their $\tau$-rlgg
        evaluate them on positive examples
        selects the $\tau$-rlgg $C'$
            that covers most positive examples
        repeat
            $C := C'$
            $Gen := \{\tau\text{-}rlgg(C, (e^+ \leftarrow)) \mid e^+ \in E^+_{Theory}$
                and $\tau\text{-}rlgg(C, (e^+ \leftarrow))$ is consistent$\}$
            let $C'$ be the clause in $Gen$
                that covers most positive examples
        until $Gen$ is empty
        add $C$ to $H$
        remove from $E^+_{Theory}$ the positive examples
            covered by $C'$
    endwhile
    return $H, E^-_{Theory}$
```

Figure 1: Learning in Instances

Learning Algorithm

We describe an algorithm for learning with an object oriented data model. The algorithm adopts an overall bottom-up strategy which first learns in instances and then generalises the results on classes along the hierarchy by means of two operators: the relative least general τ-generalisation (τ-*rlgg*) and the least general τ-generalisation (τ-*lgg*) operators, that extend the notions of relative least general generalisation (*rlgg*) and least general generalisation (*lgg*) defined by Plotkin (Plotkin 1970) to the case of multi theory logic programs.

The formal definition of τ-*lgg* of two clauses and of two atoms is given in the Appendix. Intuitively, the τ-*lgg* extends the *lgg* by taking into account hierarchical information in the background: thus, two terms belonging to different classes are generalised by their least upper bound. The τ-*rlgg* operator is obtained from the notion of τ-*lgg* in the same way as *rlgg* is obtained from the notion of *lgg*.

We assume that the instances from which we want to learn are complete on examples. Thus, every instance for which a predicate is relevant contains examples about that predicate. Also, if no example for a predicate is available in an instance, then that predicate has not to be learned in the instance and in the classes above it.

The algorithm starts recursively from the root of the hierarchy down to the instances, by calling *LearnClass*$(p/n, root; H, E^-)$. Clauses in a class C are learned after the learning in each descendent of C has been completed. For the sake of simplicity we consider the learning of a single predicate p/n. Multiple predicate learning can be performed by iterating the learning of a single predicate.

The learning process on instances in performed by the procedure *LearnInstance* in Figure 1. This algorithm is similar to GOLEM (Muggleton & Feng 1990): it differs because no post-processing generalisation phase is performed.

Learning in a class is performed by the procedure *LearnClass* in Figure 2. If the theory on which *LearnClass* is called is an instance, then *LearnInstance* is called. Otherwise, the procedure is called recursively on each son of the class. The theories returned by each son are stored in a list H_{Sons}, and negative examples in all the instances of the sons are collected in a set E^-_{Sons}.

If each theory in H_{Sons} contains a definition for the target predicate, we try to learn a definition for this predicate in the class. Otherwise, the recursive process stops and the theories in H_{Sons} are asserted in their respective son.

In order to learn a definition for the target predicate in a class, we start with an empty set of clauses H and we iteratively add a clause that generalises those in its sons. To this purpose, in each iteration of the loop, a tuple of clauses is obtained by picking one clause from each theory in H_{Sons} and the τ-*rlgg* of the clauses in the tuple is considered. The resulting clause is then tested on all the negative examples E^-_{Sons}: if it is consistent, it is added to H and all the clauses in the tuple are removed from the corresponding theories in H_{Sons}. If it is not consistent, the clauses in the tuples are left in H_{Sons} and a new iteration is started. In case one of the theories H_S in H_{Sons} becomes empty, the clause that is included in the tuple from H_S is the most specific clause $\perp$[1].

The learning process on classes continues recursively,

[1] $\perp$ is such that, for any clause C, τ-*rlgg*$(C, \perp) = C$

```
procedure $LearnClass(p/n, Theory; H, E^-)$
    if $Theory$ is an instance then
        $LearnInstance(p/n, Theory, H, E^-)$
    else ($Theory$ is not an instance)
        $H_{Sons} := \emptyset$ (list of theories learned in sons)
        $E^-_{Sons} := \emptyset$ (set of negative examples in instances)
        for each son $S$ of $Theory$ do
            $LearnClass(p/n, S; H_S, E^-)$
            Add $\{H_S\}$ to $H_{Sons}$
            $E^-_{Sons} := E^-_{Sons} \cup \{E^-\}$
        endfor
        $H := \emptyset$
        if no set $H_S$ in $H_{Sons}$ is empty do
            while every $H_S \in H_{Sons}$ is not empty do
                pick a n-tuple $T$ of clauses from $H_{Sons}$
                compute the $\tau$-rlgg $C$
                test $C$ on $E^-_{Sons}$
            if $C$ consistent then
                remove all the clauses of $T$ from $H_{Sons}$
                add $C$ to $H$
            endwhile
        endif
        assert all the clauses remained in $H_{Sons}$
            in the respective sons
    endif
    return $H, E^-_{Sons}$
end procedure
```

Figure 2: Learning in Classes

level by level as far as root is reached.

Examples

The following example should give an intuition of the behaviour of the algorithm. Consider a multi-theory background knowledge B, whose class hierarchy is depicted in Figure 3. The set T_B contains the following facts:

$buck : likes(chappy).$	$buck : likes(doggy).$
$kitty : likes(wiskas).$	$kitty : likes(kitkat).$
$fufy : likes(wiskas).$	$fufy : likes(gourmet).$

Consider the following examples:

$$E^+ = \{toby\!:\!eats(chappy), toby\!:\!eats(doggy), buck\!:\!eats(chappy), buck\!:\!eats(doggy), kitty\!:\!eats(wiskas), kitty\!:\!eats(kitkat), fufy\!:\!eats(wiskas), fufy\!:\!eats(gourmet)\ \}$$

$$E^- = \{toby\!:\!eats(kitkat), buck\!:\!eats(gourmet)\ kitty\!:\!eats(gourmet), fufy\!:\!eats(kitkat)\ \}$$

The procedure *LearnClass* is recursively called starting from the *root* class down to the instances. The instances of *dog* are first considered. The τ-*rlgg* of the positive examples in the instance *toby* is computed, obtaining the clause:

$$eats(X) :- X \ltimes dog\_food$$

Since this clause does not cover any negative example in *toby*, it is returned to the call of *LearnClass* on *dog*.

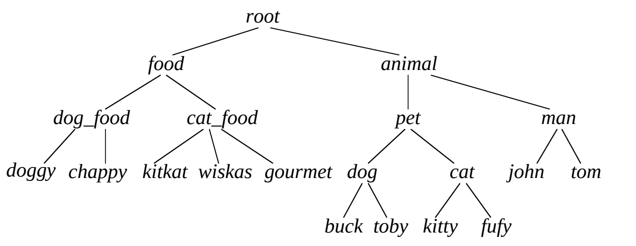

Figure 3: Class hierarchy

The τ-*rlgg* of the positive examples in the instance *buck* produces the clause:

$$eats(X) :- X \ltimes dog\_food, likes(X)$$

This clause is consistent with negative examples on *buck* and it is also returned to *LearnClass* on *dog*.

All the instances of *dog* have been considered and the clauses generated in each instance have been collected in H_{Sons}. The τ-*rlgg* of the learned clauses is computed, yielding:

$$eats(X) :- X \ltimes dog\_food$$

This clause is tested on all the negative examples in dog instances *buck* and *toby* and found to be consistent. Therefore, it is returned by *LearnClass* to the recursive call on *pet*.

Then the *cat* instances are considered. In both the *cat* instances *kitty* and *fufy*, the following clause:

$$eats(X) :- X \ltimes cat\_food, likes(X)$$

is the τ-*rlgg* of the positive examples and is consistent. Therefore, it is returned by *LearnInstance* to the call of *LearnClass* on *cat*.

In the *cat* class, the τ-*rlgg* of the clauses generated in *kitty* and *fufy* (that are actually equal) is the clause itself which is consistent on all the negative examples in the *cat* instances. Thus, it is returned to *LearnClass* on *pet*.

All the sons of *pet* have been considered and clauses generated in them for the *eats* predicate have been collected in H_{Sons}. The generalisation of these clauses produces

$$eats(X) :- X \ltimes food$$

which is then tested on negative examples in *pet* instances. This clause covers all the negative examples. As a consequence, an empty set of clauses is returned to the *animal* and then to the *root* class (i.e., the learning process stops) and the clauses learned for *cat* and *dog* are asserted in the respective class. Therefore, the final theory will contain the clauses:

$$cat\ :\ eats(X) :- X \ltimes cat\_food, likes(X)$$
$$dog\ :\ eats(X) :- X \ltimes dog\_food$$

expressing the general knowledge that dogs are usually less "picky" that cats.

Now, consider the following examples:

$E^+ = \{toby\!:\!barks\_at(john), toby\!:\!barks\_at(fufy),$
$toby\!:\!barks\_at(tom), toby\!:\!barks\_at(buck),$
$buck\!:\!barks\_at(kitty), buck\!:\!barks\_at(fufy)\ \}$
$E^- = \{toby\!:\!barks\_at(dick)\ \}$

Starting from the root, the recursive calls for the algorithm reach the instances. The instance *toby* is considered and the τ-*rlgg* of the positive examples *toby* : *barks_at*(*kitty*), *toby* : *barks_at*(*fufy*) is computed, obtaining the clause:

$$R_1 = barks\_at(X) :- X \not< cat$$

Then the τ-*rlgg* of this clause and one of the other positive examples for *toby* *toby* : *barks_at*(*tom*) or *toby* : *barks_at*(*buck*) is computed, producing

$$barks\_at(X) :- X \not< animal$$

which, however, is inconsistent on the negative example *toby*:*barks_at*(*john*). Therefore, clause R_1 is added to the current theory H. Then, a new iteration of the covering loop is started: the τ-*rlgg* of the remaining examples *toby*:*barks_at*(*tom*), *toby*:*barks_at*(*buck*) is:

$$barks\_at(X) :- hates(X), X \not< animal$$

This clause is consistent and is also added to H.

In the instance *buck*, there are only two positive examples and their τ-*rlgg* is also clause R_1.

Now, all the instances of *dog* have been considered, and the set H_{Sons} contains all the theories generated in the instances. Couples of clauses are now picked from H_{Sons} and their τ-*rlgg* is computed.

The first couple of clauses is

$$\langle (barks\_at(X) :- X \not< cat), (barks\_at(X) :- X \not< cat)\rangle$$

Their τ-*rlgg* is obviously the clause itself and is tested on the only negative example: since it is consistent, it is added to the H set for *dog* and the clause is removed from the two sets in H_{Sons}. A new iteration is started and the following couple is considered

$$\langle \perp, (barks\_at(X) :- X \not< animal, hates(X))\rangle$$

Their τ-*rlgg*, $barks\_at(X) :- X \not< animal, hates(X)$, is then found to be consistent and added to H. At this point, no clause is left in H_{Sons}, thus the procedure ends by returning the theory H.

The class *pet* is now considered. The only subclass that returns a theory containing clauses for *barks_at* is *dog*, therefore the learning process terminates by asserting the theory learned for *barks_at* in the class *dog*.

Discussion

Some aspects of the algorithm are worth to be discussed concerning the generalisation process in instances and in classes. When learning in an instance, only a small subset of examples is typically available. Since the number of negative examples may be small, we choose to rely on a notion of least general generalisation so as to avoid the risk of overgeneralisation. For example, consider the instance *toby* in the example, where we have only the two positive examples $eats(chappy), eats(doggy)$ and no negative example. We prefer to learn the most specific clause

$$toby\ :\ eats(X) :- X \not< dog\_food$$

rather than the most general clause

$$toby\ :\ eats(X) :- X \not< root$$

which is not informative.

At a first glance, when learning in classes this problem seems not to occur because negative example from all the instances are considered. However, consistency on negative examples is not sufficient to avoid overgeneralisation. In fact, in a partitioned domain, some predicates may be relevant only for some parts of the hierarchy. As an example, consider a hierarchy where the class *root* has three subclasses: *animal*, *plants* and *inanimate_objects*. Clearly the predicates *eats* is not relevant for inanimate objects and plants and no negative examples for *eats* are provided in their instances. Thus negative information does not prevent us from overgeneralising and we could learn a definition for *eats* in the root. To avoid this, we learn a definition of a predicate in a class only when all its sons contain a definition for that predicate.

An alternative approach would consider a bias that explicitly defines which instances/classes should be taken into account when learning a certain predicate. Thus, the user would be in charge of partitioning domain knowledge in two set of classes: those for which the predicate is relevant and those for which it is not.

Related Works

To the best of our knowledge, the use of an object-oriented data model in learning has been investigated only in the field of description logics. Relevant works (Kietz & Morik 1994; Cohen & Hirsh 1994; Lambrix & Maleki 1996) have discussed the problem of learning class definitions expressed in a particular description logic formalism. The learning process that is considered in these works differs from ours: there, the learning task is to build a class hierarchy starting from the descriptions of a number of instances of those classes. Instead, we learn definitions for class properties, in a given class hierarchy of those classes.

The system KLUSTER (Kietz & Morik 1994) uses a formalism that is a subset of the BACK description logic. The learning problem can be described in this way: given a set of assertions in the ABox (the examples) and an empty TBox, find a TBox (i.e., a hierarchy of concept definitions) such that the concept definitions correctly describe the examples. Examples consist in a number of assertions about concept memberships of instances and about role relations between instances. The learned theory will contain intensional definitions for the concepts and roles such that all the extensional assertions (examples) are true.

In (Cohen & Hirsh 1994) the authors consider the C-CLASSIC description logic as the representation language. They present the system LCSLEARN that takes as input a set of concept descriptions and computes the least common subsumer of the descriptions (LCS), that is a least general generalisation.

In (Lambrix & Maleki 1996) the authors use a description logic containing special construct for handling *part-of* relations. The user is allowed to give to the learning system several kinds of information on concepts that subsume (are subsumed by) the target concept, concepts that are parts of the target concept, and concepts that are collections of parts that must occur in the target concept. The system learns the definition of the target concept by iteratively reducing two version spaces, one for the is-a relation and one for the part-of relation.

Another work related to ours is (Page & Frisch 1992) where the operator *constraint generalisation* is presented that is able to generalise atoms by taking into account hierarchical relations among Herbrand universe constants. This operator computes the least general generalisation of *constrained atoms* that are atoms of the form H/C, where H is an atom and C is a set of constraints on the terms of H. Hierarchical relations can be represented by means of clauses of the form $Class(X) \leftarrow Subclass(X)$ where X is an instance.

By representing hierarchical relations by means of a first order theory, multiple inheritance is allowed: an object may thus be classified along different hierarchies. The generalisation of two constrained atoms will then contain the conjunction of least upper bounds of the terms w.r.t. all the hierarchies.

Conclusion and Future Work

The adoption of an object-oriented data model in ILP allows to exploit the benefits of object-oriented knowledge representation in learning. Complex and structured domains can be modelled straightforwardly and the resulting knowledge base can be more easily maintained.

We define an algorithm that learns definitions of properties in classes having different levels of generality, by starting from instances and then "climbing" the class hierarchy towards the root. It is worth noting that by introducing a very simple modification to classical ILP algorithms, i.e., the extension of *rlgg* operator, we obtain a substantial improvement from a knowledge representation viewpoint. We have implemented the algorithm in SICStus Prolog (SICS 1997), and tested it on structured knowledge examples.

We are currently extending the language with multiple inheritance by exploiting linearisation algorithms transforming theory graphs into a single logic theory. Future research will face the problem of extending the hierarchy by adding new classes.

References

Bergadano, F., and Gunetti, D. 1996. *Inductive Logic Programming.* MIT Press.

Brachman, R. 1979. On the epistemological status of semantics networks. In Findler, N., ed., *Associative Networks: Representation and Use of Knowledge by Computers.* Academic Press.

Bugliesi, M.; Lamma, E.; and Mello, P. 1994. Modularity in Logic Programming. *Journal of Logic Programming* 19-20:443–502.

Cohen, W., and Hirsh, H. 1994. Learning the CLASSIC description logic: Theoretical and experimental results. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference - KR94*, 121–133.

De Raedt, L., and Bruynooghe, M. 1993. A theory of clausal discovery. In Bajcsy, R., ed., *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1058–1063. MK.

Kietz, J.-U., and Morik, K. 1994. A polynomial approach to the constructive induction of structural knowledge. *Machine Learning* 14:193–217.

Lambrix, P., and Maleki, J. 1996. Learning composite concepts in description logics: A first step. In Raś, Z. W., and Michalewicz, M., eds., *ISMIS96.*

McCabe, F. 1992. *Logic and Objects.* Prentice Hall International, London.

Minsky, M. 1975. A framework for representing knowledge. In Winston, P., ed., *The Psychology of Computer Vision.* McGraw Hill.

Morik, K.; Wrobel, S.; Kietz, J.-U.; and Emde, W. 1993. *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications.* Academic Press.

Muggleton, S., and Feng, C. 1990. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, 368–381. Ohmsma, Tokyo, Japan.

Muggleton, S. 1991. Inductive Logic Programming. *New Generation Computing* 8(4):295–317.

Omicini, A. 1996. A general framework for multi-theory logic languages. Technical Report DEIS-LIA-009-96, University of Bologna (Italy). LIA Series no. 16.

Page, C., and Frisch, A. 1992. Generalization and learnability: A study of constrained atoms. In Muggleton, S., ed., *Inductive Logic Programming.* AP. 29–62.

Plotkin, G. 1970. A note on inductive generalization. In *Machine Intelligence*, volume 5. Edinburgh University Press. 153–163.

SICS. 1997. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, Kista, Sweden.

Touretzky, D.; Horty, J.; and Thomason, R. 1988. A Clash of Intuitions: The current State of Nonmonotonic Multiple IHS. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, 476–482.

Woods, W., and Schmolze, J. 1991. The KL-ONE family. *Computer & Mathematics with Applications.* Special Issue on Semantic Networks in Artificial Intelligence.

Least General τ-Generalisation

The notion of least general generalisation (Plotkin 1970) has to be extended in order to cope with the variable types determined by the class hierarchy of a program. For this purpose, in this section we will introduce the notion of τ-subsumption, and define the least general τ-generalisation (τ-*lgg*) accordingly.

We first formally define the notion of τ-expansion of a formula with respect to a program $\mathcal{P}$ which makes parent relations explicit in a logic formula. Then, τ-subsumption and least general τ-generalisation can be obtained from the definitions of θ-subsumption and *lgg*, respectively, by applying them to the τ-expansion of clauses.

For the sake of commodity, we will denote clauses as set of literals, which have obviously to be read as disjunctions of literals.

Then, τ-subsumption and least general τ-generalisation can be defined by applying the definitions of θ-subsumption and least general generalisation applied to τ-expansions. In particular, if we denote with $\tau_{\mathcal{P}}(f)$ the τ-expansion of a formula f with respect to a program $\mathcal{P} = \langle T_{\mathcal{P}}, IsA_{\mathcal{P}}\rangle$: $\tau_{\mathcal{P}}(f)$ is a clause obtained by properly rewriting f as a set of literals according to the following rules.

Given a hierarchic formula $t \mathrel{|\!\!<} o$, where t is a term and o is a ground term such that $o \mathrel{\|\!\!<} o' \in IsA_{\mathcal{P}}$,

$$\tau_{\mathcal{P}}(t \mathrel{|\!\!<} o) ::= \{t \mathrel{|\!\!<} o\} \cup \tau_{\mathcal{P}}(t \mathrel{|\!\!<} o')$$

given that

$$\tau_{\mathcal{P}}(t \mathrel{|\!\!<} \mathtt{root}) ::= \emptyset$$

by definition. Given instead a hierarchic formula $t \mathrel{|\!\!<} t'$, where t is a term and t' is a non-ground term, its τ-expansion is simply given by the formula itself, that is

$$\tau_{\mathcal{P}}(t \mathrel{|\!\!<} t') ::= \{t \mathrel{|\!\!<} t'\}$$

Given an equality formula $X = o$, where X is a variable and o is a ground term such that $o \mathrel{\|\!\!<} o' \in IsA_{\mathcal{P}}$, its τ-expansion is defined such that

$$\tau_{\mathcal{P}}(X = o) ::= \{X = o\} \cup \tau_{\mathcal{P}}(X \mathrel{|\!\!<} o')$$

Given instead an equation $X = t$, where X is a variable and t is a non-ground term, its τ-expansion $\tau_{\mathcal{P}}(X = t)$ is simply given by the formula itself, that is

$$\tau_{\mathcal{P}}(X = t) ::= \{X = t\}$$

Given a literal $(\neg)p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms, its τ-expansion is defined as follows:

$$\tau_{\mathcal{P}}((\neg)p(t_1, \ldots, t_n)) ::= \{(\neg)p(X_1, \ldots, X_n)\} \cup$$
$$\cup \tau_{\mathcal{P}}(X_1 = t_1) \cup \ldots \cup \tau_{\mathcal{P}}(X_n = t_n)$$

Given a demo formula $t \mathrel{/\!\!-} p(t_1, \ldots, t_n)$, where $t, t_1, \ldots, t_n$ are terms, its τ-expansion is defined as follows:

$$\tau_{\mathcal{P}}(t \mathrel{/\!\!-} p(t_1, \ldots, t_n)) ::= \{t \mathrel{/\!\!-} p(X_1, \ldots, X_n)\} \cup$$
$$\cup \tau_{\mathcal{P}}(X_1 = t_1) \cup \ldots \cup \tau_{\mathcal{P}}(X_n = t_n)$$

As obvious, given a clause $\{c_1, \ldots, c_n\}$, its τ-expansion is given by the union (i.e., disjunction) of the τ-expansions of its literals:

$$\tau_{\mathcal{P}}(\{c_1, \ldots, c_n\}) ::= \bigcup_{i=1}^{n} \tau_{\mathcal{P}}(c_i)$$

Definition 3. (τ-*subsumption*) A clause C τ-*subsumes* D if there exist a substitution θ such that $\tau_{\mathcal{P}}(C)$ θ-subsumes $\tau_{\mathcal{P}}(D)$. We write then $C \preceq_{\tau_{\mathcal{P}}} D$. □

Definition 4. (*least general τ-generalisation*) We say that clause C is a *least general τ-generalisation* (τ-*lgg*) of clauses C_1 and C_2 if $C \preceq_{\tau_{\mathcal{P}}} C_1$, $C \preceq_{\tau_{\mathcal{P}}} C_2$ and, for every clause E such that $E \preceq_{\tau_{\mathcal{P}}} C_1$ and $E \preceq_{\tau_{\mathcal{P}}} C_2$, it holds that $E \preceq_{\tau_{\mathcal{P}}} C$. □

Therefore, the algorithm for computing the *lgg* can be used for computing the τ-*lgg*, too.

In general, the least general τ-generalisation of two clauses may not be unique, given the redundancy introduced by the τ-expansion. In fact, given two different clauses C and C' having the same τ-expansion, if C is a least general τ-generalisation for clauses C_1 and C_2, then C' is a least general τ-generalisation for the same clauses, too. As a result, least general τ-generalisation defines an equivalence class, rather than a single clause. However, with an abuse of notation, we still speak of the least general τ-generalisation of two clauses as a single clause, meaning the minimal clause with respect to set inclusion of the equivalence class.