Machine Learning, 14, 219-232 (1994)

Flattening and Saturation: Two Representation Changes for Generalization

CÉLINE ROUVEIROL CELINE@LRI.FR
Laboratoire de Recherche en Informatique, U.R.A. 410 of CNRS, Université Paris Sud, bât 490, F-91405 Orsay, France

Abstract. Two representation changes are presented: the first one, called **flattening**, transforms a first-order logic program with function symbols into an equivalent logic program without function symbols; the second one, called **saturation**, completes an example description with *relevant* information with respect to both the example and available background knowledge. The properties of these two representation changes are analyzed as well as their influence on a generalization algorithm that takes a single example as input.

Keywords. Inductive logic programming, generalization with background knowledge

1. Introduction

We describe two representation changes that both play a role in the generalization step. The first one, called **flattening**, transforms a first-order logic program with function symbols into an equivalent program without function symbols. The basic idea of flattening, that is, representing each term in a clause C by a literal of arity n + 1 in the body of C, has been mentioned before, although quite informally, in Logic Programming (Sterling & Shapiro, 1986) and in Machine Learning (Genesereth & Nilsson, 1987; Sammut & Banerji, 1986) and is commonly used (although most of the time manually) as a representation trick in Machine Learning systems that do not handle languages with function symbols. In section 2, flattening as well as its reverse representation change, **unflattening**, are formalized: algorithms are provided, and the properties of flattening with respect to the semantics of logic programs and well-known relations of generality are studied. The second representation change, **saturation**, completes examples given some background knowledge. In section 3, saturation is formalized and compared with other approaches that also make deductive use of background knowledge for generalization. Eventually, the impact of these representation changes on an algorithm that generalizes single flattened clauses is studied.

The representation changes described in this article are implemented in the ITOU system (Rouveirol, 1992), which belongs to the Inductive Logic Programming (ILP) family (Muggleton, 1992), whose main feature is to adopt a logical framework for describing machine learning steps. As with most ILP systems, ITOU deals with a subset of First-Order Logic, definite clauses,[1] i.e., clauses with exactly one head literal (as in PROLOG). Handling function symbols is less common feature in ILP. In this framework, a clause can be interpreted as a partial definition for a concept, where the head literal identifies the defined concept and the literals in the body of the clause represent the conditions for concept membership. In the following, variables are denoted with capital letters, and constants are denoted with lower-case letters.

Different definitions of generality are used throughout this article. The first one is adapted from θ-subsumption introduced in Plotkin (1969). A clause C is θ-subsumed by a clause G if there exists a substitution θ such that $head(C) = head(G)\theta$ and $body(G)\theta \supseteq body(C)$. Generality can also be modeled by logical entailment: a clause G is more general than C if $G \models C$. This definition naturally extends when some background knowledge is available: in this case, a clause G is more general than C, with respect to a logic program T (the background knowledge) if $T \cup G \models C$. θ-subsumption and logical entailment are equivalent for languages without function symbols, but they differ with respect to recursive clauses (Niblett, 1988).

2. Flattening

The basic idea of flattening is to introduce for every function symbol f of arity n a new predicate f_p of arity $n + 1$, where the first n arguments are the same as for the function and the last argument is the result of the function.

2.1. Definitions and algorithms

- **Flattening predicate:** The *flattening predicate* f_p associated with the function symbol f of arity n is the predicate of arity $n + 1$ defined by

$$f_p(X_1, \ldots, X_n, X) \leftrightarrow X =^2 f(X_1, \ldots, X_n)$$

The variable X is called the *output argument* of the flattening predicate.

The set of flattening predicates associated with a clause C is denoted *Flat_Defs*(C). The *flattened clause* C_f corresponding to a clause C (denoted *flat*(C)) is the result of flattening all the terms of C. Notice that in a clause of *Flat_Defs*(C), only one implication out of the original equivalence in the definition of a flattening pedicate is represented. The reason is that clauses of *Flat_Defs*(C) are only used for unflattening *flat*(C) (or some generalization of *flat*(C)) and that this part of the equivalence is sufficient to perform unflattening.

One of the most delicate choices when implementing flattening is when the same term occurs several times in the same clause. The version of flattening presented in figure 1 replaces every occurrence of the same term by the same variable. For example, let E_x be the clause to flatten:

E_x: *member*(*blue*,[*blue*]) $\leftarrow$.

The final flattened clause, once all the function calls have been replaced, is

E_{xf}: *member*(X,Z) $\leftarrow$

consp(X,Y,Z) $\wedge$ *bluep*(X) $\wedge$ *nilp*(Y).

Procedure flatten(P, P_f, *Flat_Defs(P)*)

% P is a definite program with function symbols, P_f is the corresponding flattened

% program, *Flat_Defs(P)* is a definite program that stores the definitions of flattening predicates

$P_f := \emptyset$, *Flat_Defs(P)* := $\emptyset$

For all clauses C of P

$C_f := C$

For all terms $f(t_1, ..., t_n)$ in C_f % *Term flattening*

If the flattening predicate f_p corresponding to f does not exist in *Flat_Defs(P)*

Then add $f_p(t_1, ..., t_n, X) \leftarrow X = f(t_1, ..., t_n)$ to *Flat_Defs(P)*

endif

Replace **all** occurrences of $f(t_1, ..., t_n)$ in C_f by a new variable X_i, and add the literal $f_p(t_1, ..., t_n, X_i)$ to the body of C_f

endfor

$P_f := P_f \cup C_f$

endfor

return (P_f , *Flat_Defs(P)*)

Figure 1. Flattening algorithm.

with the three following clauses in *Flat__Defs*(E_x):

NIL: $nilp(nil) \leftarrow$
CONS: $consp(X, Y, cons(X,Y)) \leftarrow$.
BLUE: $bluep(blue) \leftarrow$.

Some approaches (Ling, 1991; Nienhuys Cheng, 1991; Banerji, 1992) replace each occurrence of the same term by different and independent variables. The flattened clause in this case would be

E_{xf}: $member(X,Z) \leftarrow$
$consp(X',Y,Z) \wedge bluep(X) \wedge bluep(X') \wedge nilp(Y)$.

This option makes the flattened clause more difficult to read because of the high number of variables occurring in the clause. Moreover, the fact that X and X' may point to the same object is lost (it is only implicitly captured by the usual convention in logic that two different variables may unify).

An alternative consists in storing explicitly in the body of the flattened clause the links between the variables introduced by flattening (Muggleton, 1992). The flattened clause in this case is

$E_{xf''}$: $member(X,Z) \leftarrow$
$consp(X',Y,Z) \wedge bluep(X) \wedge bluep(X') \wedge nilp(Y) \wedge X = X'.$

We will focus in this article on the first and third way to flatten. The first version of flattening generates the simplest (in the sense that the number of variables in the resulting flattened clause is minimal) and most specific flattened clause, which is equivalent to the input clause C with respect to $Flat\_Defs(C)$. This translates the assumption that simultaneous occurrences of the same term in a clause are more likely to be caused by some correlation than by chance. This version of flattening "sticks" to the example as the more specific bindings between arguments of the original clause are transmitted to the flattened clause. Choosing this option for flattening has the consequence that generalization algorithms working on flattened clauses are biased towards the simplest and most specific generalizations of the example(s). In the latter option, a greater number of variables are introduced, but explicit unifications of variables are also kept in the flattened clause. This formulation of flattened clauses is particularly adapted to generalization (see section 4).

- **Unflattening:** *Unflattening* a clause C_f given a set of flattening predicates $Flat\_Defs(C)$ (denoted $unflat(C,Flat\_Defs(C))$ is achieved by applying linear resolution with set of support $Flat\_Defs(C)$ to C_f. If flattening has been performed such that explicit unifications are stored in the body of the flattened clause, resolutions with the axiom $X = X \leftarrow$ first have to be performed, so that all equalities disappear in the body of the unflattened clause.

In the previous example, observe that E_{xf} resolved with *NIL*, *CONS*, and *BLUE* yields E_x. Let us also point out a side effect of unflattening after dropping some literals from a flattened clause (see section 4.1). Literals such that their output variable (i.e., the one standing for the result of the function call) does not occur elsewhere in the clause disappear during the unflattening process. For example, let us consider the unit clause

Ex: $f(g(h(a))) \leftarrow$.

The corresponding flattened clause is

Ex_f: $f(Z) \leftarrow a_p(X) \wedge hp(X,Y) \wedge g_p(Y,Z)$

together with the following set of definitions of flattening predicates $a_p(a) \leftarrow$, $h_p(X,h(X)) \leftarrow$, $g_p(X,g(X)) \leftarrow$. Dropping the literal $h_p(X,Y)$ yields the clause

Ex_{fg}: $f(Z) \leftarrow a_p(X) \wedge g_p(Y,Z)$

which yields, once unflattened,

Ex_g: $f(g(X)) \leftarrow$.

Therefore, unflattening allows us to drop literals such as $a_p(X)$ in the example, since X does not appear anywhere else in Ex_{fg}.

The algorithm in figure 1 always terminates. It introduces a new predicate for each symbol function instead of directly adding the equality $X = f(X_1, \ldots, X_n)$ in the body of the flattened clause. It is interesting when flattened clauses are processed by an algorithm that explores possible matchings between sets of literals with the same predicate symbol (such as *lgg* (Plotkin, 1969), for example). This step will be much more efficient if the flattened clauses are described using several predicates for indexing the equalities with respect to the function they represent, instead of using the one equality predicate.

2.2. *Properties of flattening*

Flattening and unflattening do not change the semantics of a logic program P, and it is therefore equivalent, as far as the logical closure of the theory is concerned, to work on clauses with function symbols or on flattened clauses, given *Flat_Defs*(*P*).

Theorem

$$P \models A \quad \textit{iff } \textit{flat}(P), \textit{Flat\_Defs}(P) \models \textit{flat}(A)$$

where P is a definite clause program, A is a definite clause, and *Flat_Defs*(*P*) denotes the set of flattening predicate definitions for P.

Corollaries:

If C is a definite clause with function symbols,

- *unflat*(*flat*(*C*), *Flat_Defs*(*C*)) = *C*, up to a variable renaming;
- *flat*(*unflat*(*C*, *Flat_Defs*(*C*))) = *C*, up to a variable renaming; and
- *C* θ-subsumes *C'* iff *flat*(*C*) θ-subsumes *flat*(*C'*).[3]

Let us sum up the main features of flattening. First, it gives a uniform syntax to the language used internally by the system. This enables a simple implementation of generalization of a single clause only by dropping literals. More complex hypotheses (that is, with more independent variables) are then considered when the simplest hypothesis fails to explain all the examples. Eventually, we show in the next section how representing constants and terms as predicates in flattened clauses allows us to build pertinent reformulations of the examples.

3. Saturation: relevant use of background knowledge

Saturation is a reformulation operator that enables us to take background knowledge into account during the generalization process. Given an example E (a definite clause) and some

background knowledge T (a definite program), saturation builds a clause that is logically equivalent to E given T by completing E with *pertinent* literals with respect to E and T. Pertinent literals with respect to E and T are literals H_i such that $flat(T)$, $flat(body(E))$ $/\text{-}_{SLD}$ H_i, with the additional constraint that for each step of the SLD derivation yielding one H_i, at least one parent clause must belong to $flat(body(E))$ (linear derivation with set of support $flat(body(E))$).

Saturation introduces a semantic (that is, in term of logical entailment) relevance criterion on literals that complete the example description, as opposed to systems such as GOLEM (Muggleton & Feng, 1990) or CLINT (De Raedt, 1992) that choose the literals to complete the example in a finite submodel of the background knowledge (Rouveirol & De Raedt, 1992) with respect to syntactic restrictions on the hypothesis language (see figure 2).

Let us illustrate this claim in an example after Buntine (1987). The first version of flattening is used when implementing saturation. This is because saturation completes a clause with logical consequences of both literals in the body of this clause and some background knowledge. Although the obtained clause is logically equivalent to the input clause with

---

Procedure saturation(C_e, T, C_s)
% C_e is a definite clause, as well as C_s. T is a definite program
flatten(C_e, C_{ef}, $Flat\_Defs(C_e)$) ; $F := \emptyset$
For all literals Lc_{ei} of $body(C_{ef})$ F:= F $\cup$ (Lc_{ei} $\theta_s \leftarrow$) **endfor**
% θ_s is a substitution that skolemizes every variable of C_e, F contains the set of all
% skolemized literals representing C_e.
Repeat % Deductive phase
changed := false; depth := 0
  For each clause C_k of T
    If every literal of $body(flat(C_k))$ can be resolved with facts of F[4]
    Then
      F := F $\cup$ ($head(C_k)$ $\theta_s \leftarrow$)[5]; changed := true
    endif
    depth := depth + 1
  endfor
until not changed or depth > depth-bound
% F at this point the set of skolemized unit clauses {Lc_{ej} $\theta_s \leftarrow$} with j $\geq$ i
C_{ss}: T_e θ_s $\leftarrow Lc_{ej}$ θ_s % The body of C_{ss} is the conjunction of the facts of F
$C_{sf} := C_{ss}\ \theta_s^{-1}$
unflatten(C_{sf}, $Flat\_Defs(C_e)$,C_s)
return (C_s)

Figure 2. Saturation algorithm.

respect to logical entailment, it is more specific with respect to θ-subsumption. Since the aim of saturation is to complete with relevant information and not to generalize, it seems more adapted when flattening the example clause and the domain theory to replace every occurrence of the same term by the same variable. Moreover, handling equalities literals during the deduction step of Saturation may result in problems. The domain theory is

$$list([]) \leftarrow$$

$$list(X.Y) \leftarrow list(Y)$$

$$member(X,X.Y) \leftarrow list(Y)$$

and the unit clause we want to generalize is E_x: $member(4,[3,4]) \leftarrow$. First, the domain theory and the example are flattened:

T_1: $list(X) \leftarrow nilp(X)$
T_2: $list(Z) \leftarrow list(Y) \wedge consp(X,Y,Z)$
T_3: $member(X,Z) \leftarrow list(Y) \wedge consp(X,Y,Z)$
E_1: $member(Y,U) \rightarrow$
$three(X) \wedge four(Y) \wedge nilp(N) \wedge consp(Y,N,Z) \wedge consp(X,Z,U).$

The corresponding flattening predicates are F_1: $three(3) \leftarrow$, F_2: $four(4) \leftarrow$, F_3: $nilp([]) \leftarrow$, and F_4: $consp(X,Y,cons(X,Y)) \leftarrow$.

Saturation traced on this example provides the following intermediary results (bold literals are the ones introduced by Saturation). First, E_1 is saturated with T_1:

E_{s11}: $member(Y,U) \leftarrow$
$three(X) \wedge four(Y) \wedge nilp(N) \wedge \mathbf{list(N)} \wedge consp(Y,N,Z) \wedge \mathbf{consp}(X,Z,U)$

then saturated twice with T_2:

E_{s12}: $member(Y,U) \leftarrow$
$three(X) \wedge four(Y) \wedge nilp(N) \wedge list(N) \wedge consp(Y,N,Z) \wedge \mathbf{list(Z)} \wedge$
$consp(X,Z,U) \wedge \mathbf{list(U)}$

and, at last, twice with T_3:

E_{s13}: $member(Y,U) \leftarrow$
$three(X) \wedge four(Y) \wedge nilp(N) \wedge list(N) \wedge consp(Y,N,Z) \wedge list(Z) \wedge$
$\mathbf{member(Y,Z)} \wedge \ consp(X,Z,U)\ list(U) \wedge \mathbf{member(X,U)}.$

Saturation then stops, since no more new literals can be deduced from literals of the body of E_{1s}: $member(4,[3,4]) \leftarrow$
$member(4,[4]) \wedge list([4]) \wedge member(3,[3,4]) \wedge list([3,4]).$

Saturation may loop whenever the background knowledge contains one clause that is not *range restricted*, i.e., when one or more variables that occur in the head of the clause do not occur in the body of the clause. This demonstrates one advantage of performing saturation on flattened clauses. The following clause is not range restricted:

$$list(X.Y) \leftarrow list(Y).$$

Flattening rewrites the above clause into

$$T_2\text{: } list(Z) \leftarrow list(Y) \wedge consp(X,Y,Z)$$

which is range restricted again. Saturation on flattened clause will then loop less often than saturation on nonflattened clauses, because representing terms as predicates in the body of the flattened clause can make them range restricted if the isolated variable(s) in the head of the clause occur in a term.

The above example demonstrates that saturation only introduces into the body of the completed flattened example, instances of *member* and *list* predicates built on terms and subterms occurring in the example. This criterion is particularly adapted when the background knowledge is made of rules (especially recursive rules), but it turns out to be too restrictive when the background knowledge contains ground facts (Rouveirol & De Raedt, 1992).

4. Generalization of a flattened clause

Let us now examine one possible algorithm that works on flattened clauses. It builds generalizations of a single clause E with reference to some background knowledge T. The input is a definite clause with function symbols, which is first flattened and saturated given T. Then a purely inductive generalization algorithm can be applied to the saturated clause in order to build the generalization of the input clause given the background knowledge. We will show here that working on flattened clauses allows us to introduce a more elegant formalization for generalization of single clauses.

4.1. Definition

Truncation[6] is a purely inductive operator that builds all the possible generalizations of a single clause with respect to θ-subsumption.[7] That is, given a clause C, Truncation generates all the clauses G_i such that G_i θ-subsume C, or in other words, it builds all the clauses G_i such that $\exists\ \theta$, $head(G_i) = head(C)\ \theta$ and $body(C)\ \theta \supseteq body(G_i)$. Truncation on a clause C_e: $T_e \leftarrow Lc_e$ with function symbols amounts to

- dropping one or more literals of Lc_e
- inverting a substitution θ on C_e, i.e., turning some occurrences of some terms or subterms of C_e into variables

The first point is easy to implement (although less easy to control), and it has been known for quite a long time as the *dropping condition rule* (Michalski, 1983). Therefore, let us rather concentrate on the inverse substitution point.

All the generalizations under θ-subsumption of the term t: $p(a,a)$ are organized in the generality lattice shown in figure 3. Each of those generalizations may be built by inverting a substitution on t. Let σ be a substitution $\{X_i/t_i\}$ where the X_i are the variables and the t_i the terms of the substitution. We call σ^{-1} the inverse substitution of σ the unique mapping[8] such that given any literal L, $L\,\sigma\,\sigma^{-1} = L$ (after Muggleton & Buntine, 1988).

We distinguish two groups of substitutions: *injective*[9] substitutions (in particular renaming substitutions, which are bijections) and noninjective substitutions. Any substitution θ can be expressed as the composition of an injective substitution σ and of some noninjective substitutions ν_i, such that each ν_i substitutes different variables by the same term (Rouveirol, 1992). We therefore distinguish two groups of inverse substitutions. *Simple inverse substitutions* invert injective substitutions. In other terms, given a term t_i in a clause C, a simple inverse substitution replaces **all** occurrences of t_i by the same variable ν_i. In our previous example, inverting an injective substitution on $p(a,a)$ yields $p(X,X)$. The second group of inverse substitutions, called *splitting inverse substitutions*, splits occurrences of the same term t_i in a clause C into several different variables. Applying a splitting inverse substitution to the literal $p(a,a)$ would yield $p(a,Y)$, $p(a,Y)$, or $p(X,Y)$.

4.2. *Generalization of a flattened clause*

Let us first consider the case where the clauses are flattened using the version of flattening that replaces all occurrences of a term by the same variable. The algorithm of figure 4, inspired by De Raedt (1992) and by the smaller step generalization algorithm (Nédellec, 1992), takes into account the structure of the deduction graph of saturation in order to keep the generalization as specific as possible. Initial literals in the body of the example are first dropped; the algorithm thus generalizes as little as possible and asks for user validation before proceeding further in the generalization graph. This algorithm, because of flattening, inverts θ-subsumption for injective substitutions only. The process of inverting injective substitutions on a clause with function symbols is therefore brought down to dropping literals on the corresponding flattened clause where all the occurrences of the same term are replaced by the same variable.

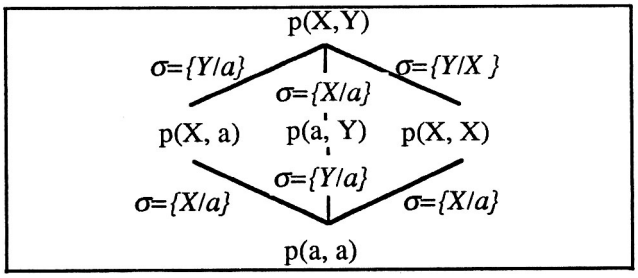

Figure 3. Generality lattice.

```
Procedure dropping_rule(C, S_g)
% C: saturated and flattened clause, S_g : set of candidate generalizations for C
S_g := {C}
For each clause C in S_g
   Delete C from S_g
   For all literals Lb_i that are as low a possible in the derivation graph of body(C)
      Propose the generalization of C, C - Lb_i to the user
      If the generalization is validated
         then add C - Lb_i to S_g
         else
            D(Lb_i) = literals which have been deduced from Lb_i during saturation
            add C - D(Lb_i) to S_g % keeps the most specific literals if generalization fails
      endif
   endfor
endfor
return (S_g)
```

Figure 4. Dropping rule algorithm.

The version of flattening that maintains equality literals in the body of the flattened clause (Muggleton, 1992), although it provides more complex clauses, enables us to invert θ-subsumption for arbitrary substitutions. Purely inductive generalization of a single flattened clause *flat*(C) can then be divided into two steps:

- dropping literals of *flat*(C) that are not equalities, which amounts to both dropping literals and inverting injective substitutions on C
- dropping *equality literals of flat*(C) in conjunction with some instances of flattening predicates, which inverts noninjective substitutions on C

The algorithm in figure 5 realizes this kind of generalization. Used in combination with the algorithm of figure 4, it forms the basis of a complete algorithm for inverting θ-subsumption.

For example, let C be the unit clause C: $p(a,a) \leftarrow$. The flattened clause corresponding to C is C_f: $p(X,Y) \leftarrow a(X) \wedge a(Y) \wedge X = Y$. Dropping the literals $a(X)$ and $a(Y)$ inverts an injective substitution on C_f and replaces all the occurrences of a by the same variable in the corresponding unflattened clause, $p(X,X) \leftarrow$. If the literals $X = Y$ and $a(Y)$ are dropped, the generalization $p(a,Y) \leftarrow$ is obtained, dropping $X = Y$ and $a(X)$ yields $p(X,a) \leftarrow$, and $p(X,Y) \leftarrow$ is obtained by dropping $a(X)$, $a(Y)$ and $X = Y$.

```
Procedure dropping_ruleb(C, S_g)
% C: saturated and then flattened clause, S_g : set of candidate generalizations for C
S_g := {C}
For each clause C in S_g
   Delete C from S_g
   For each triplet (X=Y, fp(V,X), fp(V,Y)) in body(C) % V is a vector of variables
      the four possible generalizations are obtained by dropping any two literals or the
      three literals of the triplet
   Propose each of these generalizations to the user
   For each possible generalization
      If the generalization is validated
         then add the generalization to S_g
         else
            D(Lb_i) = literals which have been deduced from Lb_i during saturation
            add C - D(Lb_i) to S_g % keeps the most specific literals if generalization fails
      endif
   endfor
endfor
return (S_g)
```

Figure 5. Dropping rule algorithm on flattened clauses with equality literals.

5. Related works and conclusion

We have presented in this article two representation changes that are used in a bottom-up generalization system called ITOU. Their main advantage is that they are independent from any learning algorithm. If we consider flattening, many systems (Ling, 1989; De Raedt, 1992; Sammut & Banerji, 1986; Quinlan, 1990; Banerji, 1992) make the assumption that their input clauses are without function symbols and therefore need to transform their examples using techniques similar to flattening in order to simulate some function symbols. Flattening plus some syntactic restrictions or some adequate heuristic to limit the search of the learning algorithm in the space of flattened literals may be a way to extend algorithms that handle propositional or DATALOG languages only. Saturation is as well a very general mechanism that allows us to take background knowledge into account in one pass only. It can be added as a front end to learning algorithms (Bisson, 1992) that traditionally do not use background knowledge. Saturation coupled with flattening allows us to deal in a nonheuristic way with some types of background knowledge with infinite models.

Flattening was first designed (Rouveirol & Puget, 1989) to solve the problem of building all possible inverse substitutions in CIGOL (Muggleton & Buntine, 1988). We have first

provided an algorithm that computes absorptions for definite clauses with function symbols. The flattening/unflattening process transferred the problem of inverting substitutions in Absorption (the only substitutions are renaming substitutions, which are much easier to invert) to the one of dropping literals in Truncation, where all the inductive choices took place. In the version of flattening from Rouveirol and Puget (1989), each occurrence of the same term was replaced by a different variable.

Nienhuys-Cheng, and Flach (1991) and Nienhuys-Cheng (1991) independently developed a representation change similar to flattening, called *tree coding* of clauses. They use this special coding to define term partitions to build all the generalizations of a given clause, with respect to the θ-subsumption relation. If we do not consider this difference that the formalization of Nienhuys-Cheng is algebraic whereas ours is logical, it achieves results similar to our Truncation operator with the splitting occurrence rule with respect to generalization of a single clause without reference to background knowledge. However, the main difference between our approaches is that Nienhuys-Cheng does not consider the problems of generalization in presence of background knowledge.

Some work has been done as well on generalization of the set of clauses (Rouveirol, 1992). In this case, flattening is still useful for constraining saturation, but if some lgg-like algorithm (Plotkin, 1969) is to be used, it may be more efficient to unflatten the clauses before generalization in order to prevent the lgg algorithm from forming irrelevant selections for subterms that have the same function symbols.

Appendix: Proof of the theorem

Theorem:

$$P \models A \text{ if and if } flat(P), Flat\text{-}defs(P) \models flat(A),$$

where $Flat\text{-}defs(P)$ denotes the sets of flattening predicates definition clauses for P.

Proof: The theorem can be proved by induction on the complexity of terms of P: induction on the number of arguments of P, induction on the depth of terms of P. We propose here the proof for the simplest case only.

Let us suppose that P is made of one unit clause: $pred(c)$. $Pred$ is a predicate symbol of arity one, and c is a constant. The flattening predicate of the constant c is

$$c_p(X) \leftrightarrow X = c.$$

We have to prove, in order to prove the theorem that

(1) $pred(X) \leftarrow c_p(X), c_p(X) \leftrightarrow X = c$ implies $pred(c)$.
(2) $pred(c), c_p(X) \leftrightarrow X = c$ implies $pred(X) \leftarrow c_p(X)$.

(1) This is the simplest part to prove. c_p is the flattening predicate defined by $c_p(X) \leftrightarrow X = c$. We restrict the equivalence to the only implication:

(i) $c_p(X) \leftarrow X = c$.

If we resolve $pred(X) \leftarrow c_p(X)$ with **(i)**, we get $pred(X) \leftarrow X = c$.
By applying one of the equality axioms:

$$x1=y1 \wedge \ldots xn=yn \rightarrow f(x1, \ldots, xn) = f(y1, \ldots, yn)$$

we get the following clause: $pred(c)$. ■

(2) We reduce this to the absurd. Let us suppose we have $pred(c)$ and that we do not have $pred(X) \leftarrow c_p(X)$.

This would mean that $\$ a , \neg pred(a) \wedge c_p(a)$. However, by definition, $c_p(X) \leftrightarrow X = c$, and therefore $a = c$. From two previous assertions, we can derive $\neg pred(c)$, which is in contradiction with $pred(c)$. ■

Acknowledgments

I would like to thank Yves Kodratoff and all the Inference and Learning Group at Orsay. Thanks to J.F. Puget, since most of the ideas developed in this article had their source in discussions with him, and to Maurice Bruynooghe, Luc de Raedt, and all the members of the A.I. group of Katholieke Universiteit Leuven. Special thanks to the anonymous reviewers; the last version of the article benefitted significantly from their comments (especially the discussion concerning the flattening), and to Katharina Morik for her enthusiasm. This work has been partially supported by CEC through ESPRIT-2 contract ECOLES (n°3059) and BRA ILP (n°6020).

Notes

1. The reader may refer to Lloyd, (1987) for basic definitions of Logic Programming.
2. We use a standard equality theory, for instance, the axioms defined in Shepherdson (1987, pp. 27–28).
3. This does not generalize to logical implication.
4. There may be several ways to resolve facts of F with all the literals of $body(C_k)$.
5. If $head(C_k)$ contains variables that are not instantiated by θ_S, then θ_S is extended to skolemize these variables as well.
6. Truncation as presented here is more general than the Truncation operator (Muggleton & Buntine, 1988) in that it covers the dropping condition rule (Rouveirol, 1992).
7. The reader should refer to Lapointe and Matwin (1992) for an extension of Truncation to inversion of implication.
8. We only consider here idempotent substitutions (Lassez et al. 1987).
9. A substitution σ is *injective* iff $\forall\ v_1, v_2$ belonging to the domain of σ, if $v_1 \neq v_2$ then $\sigma(v_1) \neq \sigma(v_2)$.

References

Banerji, R.B. (1992). Learning theoretical terms. In S. Muggleton (Ed.) *Inductive logic programming*. New York: Academic Press.

Bisson, G. (1992). Conceptual clustering in a first order logic representation, *Proceedings of the Tenth European Conference on Artificial Intelligence* (pp. 459–462). New York: Wiley.

Buntine, W. (1987). Induction of Horn Clauses: methods and the plausible generalization algorithm. *International Journal of Man & Machine Studies*, *26*, 499–520.

Buntine, W. (1988). Generalized subsumption and its applications to redundancey. *Artificial Intelligence*, *36*, 149–176.

De Raedt, L. (1992). *Interactive theory revision.* New York: Academic Press.

Genesereth, M.R. & Nilsson, N. (1987). *Logical foundations of artificial intelligence.* Los Altos, CA: Morgan Kaufmann.

Lapointe, S., Matwin, S. (1992). Sub unification: a tool for efficient induction of recursive programs. *Proceedings of the Ninth International Machine Learning Conference.* Los Altos, CA: Morgan Kaufmann.

Lassez, M.J., Maher, M.J., & Marriot, J.L. (1988). Unification revisited. In J. Minker (Ed.) *Foundations of deductive databases and logic programming.* Los Altos, CA: Morgan Kaufmann.

Ling, C.X. (1989). Learning and inventing horn clauses theories. In Z.W. Ras (Ed.), *Methodologies for intelligent systems*, 4. Amsterdam: North Holland.

Ling, C.X. (1991). A critical comparison of various methods based on inverse resolution. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 168–172). Evanston, IL: Morgan Kaufmann.

Lloyd, J.W. (1987). *Foundations of logic programming*, 2nd extended edition. New York: Springer-Verlag.

Michalski, R.S. (1983). A theory and methodology of inductive learning. In R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 1). Tioga.

Muggleton, S., & Buntine, W. (1988). Machine invention of first order predicates by inverting resolution. *Proceedings of the Fifth International Machine Learning Workshop* (pp. 339–352). Ann Arbor, MI: Morgan Kaufmann.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. *Proceedings of the First Conference on Algorithmic Learning Theory.* Tokyo: Ohmsha.

Muggleton, S. (1992). Inverting implication. *Proceedings of the 2nd International Workshop on Inductive Logic Programming* (technical report), Tokyo: ICOT.

Nédellec, C. (1992). How to specialize by theory refinement. *Proceedings of the European Conference on Artificial Intelligence* (pp. 474–478). Vienna: Wiley.

Niblett, T. (1988). A study of generalization in logic programs. *Proceedings of the Third European Working Session on Learning* (pp. 131–136). Glasgow, Pitman.

Nienhuys-Cheng, S.H. (1991). Consistent term mapping, term partition and inverse resolution. *Proceedings of the IJCAI'92 Workshop on Evaluating and Changing Representation.* Sydney.

Nienhuys-Cheng, S.H., & Flach, P.A. (1991). Consistent term mapping, term partition and inverse resolution. In *Machine Learning: EWSL-91: European Working Session on Learning* (pp. 361–374). Porto: Springer-Verlag.

Plotkin, G.D. (1969). A note on inductive generalization. In B. Meltzer and D. Michie (Eds.) *Machine intelligence 5.* Edinburgh: Edinburgh University Press.

Quinlan, J.R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Rouveirol, C. (1992). ITOU: Induction of first order theories. In S. Muggleton (Ed.), *Inductive logic programming.* New York: Academic Press.

Rouveirol, C., & De Raedt, L. (1992). The use of background knowledge. *Proceedings of the ECAI92 Workshop on Logical Approaches to Learning.* Vienna.

Rouveirol, C., & Puget, J.F. (1989). A simple solution for inverting resolution. *Proceedings of the Fourth European Working Session on Learning* (pp. 201–211). Montpellier, Pitman.

Sammut, C., & Banerji, R.B. (1986). Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell, & R.M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). Morgan Kaufmann.

Shepherdson, J.C. (1987). Negation in Logic programming. In J. Minker (Ed.), *Logical foundations of deductive databases.* San Mateo, CA: Morgan Kaufmann.

Sterling, L., & Shapiro, E. (1986). *The art of Prolog: Advanced programming techniques.* Cambridge, MA: MIT Press.

Received February 10, 1992

Accepted June 4, 1992

Final Manuscript September 15, 1992