Extensions of Inversion of Resolution applied to Theory Completion

Céline Rouveirol

Laboratoire de Recherche en Informatique, [1]
Equipe Inference et Apprentissage,
Universite Paris Sud, bat 490
F-91405 Orsay, France
email : celine@lri.lri.fr

Abstract

We present in this paper a decomposition in two elemetary steps of a well known process in Concept Learning, generalisation in presence of background knowledge. The first step, called *Saturation* reformulates an initial example given some background knowledge. Saturation indeed computes by using sound steps only the most specific relevant generalisation of an initial clause, given a domain theory. The second operator, *Truncation*, is a purely inductive generalisation operator that computes, given an initial clause E, all the clauses that θ-subsume E. By combining these two operators, we can compute all the relevant generalisations of an initial clause, given a domain theory. We then propose two constraints, one purely syntactical (i.e., which set a constraint on the *form* of the generalisation we are looking for) but with interesting semantic properties and a second one that express a property rather than just a restritcion on the syntax of the searched generalisation. These two kinds of declarative knowledge allows us to efficiently guide the generalisation process. We then draw comparisons with related works.

1 Introduction.

1.1 Motivations.

duv

We present in this paper a learning system named ITOU that addresses the problem of incomplete domain theory. In ITOU, A explains B iff A logically implies B, denoted $A \models B$, which is equivalent to A is more general than B. We will not handle explictly in this paper negative examples, but T is assumed to be correct. The failure example is assumed to be error free and at last, we assume that the initial vocabulary is sufficient to correctly describe the examples (that means, two instances of different concepts necessarily have different descriptions) and the domain theory.

ITOU is a closed loop system. It takes as input an incomplete domain theory T and an example E such that $T \not\models E$. ITOU looks for a new theory T', such that $T' \models E$, and $T' \models T$. It is thus a monotonic system in that the logical closure of the theories

[1] This paper was written while the author was visiting Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

successively built by ITOU is monotonically increasing. We will describe in this paper the particular case where T' is equal to T plus one single clause C. This clause C is built by applying two operators to the failure example E given the domain theory T. The first operator, called *Saturation*, is a reformulation operator: it computes with a sound method, given E and T, a clause E_s which is a most specific version of E given T: it contains all the possible generalisations for E with respect to T. The second operator, *Truncation*, allows to build all the purely inductive generalisations G_i of a given clause E without use of the background knowledge. The search space for Truncation is very huge, therefore some additional knowledge is needed to constrain the search for appropriate generalisations. First, Truncation is implemented as the generalisation of several input clauses instances of the same concept instead of one, in order to guide Truncation application wiht structural similarities among examples of the same class. Secondly declarative bias is needed to further specify set of constraints on the generalisation. Generalisations G_i that satisfy a set of given constraints are proposed to the user for validation. If one G_i is validated by the oracle, it is added to the initial domain theory T to form one possible T_i'. ITOU then waits for a new unrecognized example to perform its loop again.

In the remainder of the paper, we will first describe in section 2 the Saturation operator. The algorithm of Saturation is detailed, and we present Saturation as a link between inductive inference (inversion of resolution) and sound inference procedures, including skolemisation and resolution. In section 3, we will then detail the Truncation operator and representation changes, $flattening$ and $unflattening$, that allows us to describe Truncation in a uniform way We will then introduce in section 4 a notion of completeness for inductive procedure, called *i-completeness* and we will prove that combination of Saturation and Truncation achieves a restricted form of i-completeness. We finally address the control issue for these operators (section 6). More precisely, we will present two biases, one purely syntactic, called *connection*, and the other one which can be seen as a kind of integrity constraint, the functional relation bias, that allow us to efficiently guide Truncation. At last, we will discuss related works and further directions of research.

1.2 Notations.

In this work, we adopt a logical framework. We deal with definite clauses, i.e., Horn clauses with exactly one head literal. 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 represents the preconditions for an object to belong to this concept. An example C_i is an instance of a concept C iff the clause representing C θ-subsumes the clause representing C_i. We adapt the definition of $\theta-$subsumption given in [Plotkin 70] to the following one restricted to clauses examples of the same concept. A clause $C\theta-$subsumes a clause C_i iff $\exists\theta$ such that:

- $head(C) = head(C_i)$
- $body(C_i) \supseteq body(C)\theta$

that is, the body of the clause representing C_i unifies with a subpart of the body of the clause representing C. Note that the notion of $\theta-$subsumption does not refer to any domain theory that could be available. Variables are denoted with capital letters A,B, etc..., constants are denoted with lowercase letters. We assume the reader is familiar with notations introduced in [Lloyd 87].

2 Saturation

2.1 Loss of information for generalisation

Let us present here the elementary step of generalisation of a single clause C_e in presence of a domain theory T. Such a generalisation step has been described, for example, in MARVIN [Sammut and Banerji 86] under the name of Elaboration.

Definition 1: Let C_e be a clause $C_e : T_e \leftarrow Lc_e$, called the initial or example clause, and a clause $C_t : T_t \leftarrow Lc_t$ belonging to T, such that $Lc_t\theta$-subsumes Lc_e (that is, $\exists\theta$ such that $Lc_e \supseteq Lc_t\theta$). The generalisation of C_e given C_t is the following clause:

$$C_h: \qquad T_e \leftarrow Rel \wedge T_t\theta$$

where $Lc_e = Lc_t\theta \wedge Rel$.

Generalisation consists in replacing in the body of C_e some literals that are matching the body of C_t up to a substitution θ by the head of C_t substituted by θ. Let us take for illustration the well known arch example. We represent examples of arch as definite clauses where the head literal is made of the symbol predicate of arity one, $arch(X)$. The rules of the domain theory, $T_1, \ldots, T_4$, are also expressed as definite clauses.

$$\begin{array}{lll}
E: & arch(X) \leftarrow & \\
 & & part_of(A,X) \wedge part_of(B,X) \wedge part_of(C,X) \wedge \\
 & & is_on(A,B) \wedge is_on(A,C) \wedge is_on(B,D) \wedge \\
 & & is_on(C,E) \wedge ground(D) \wedge ground(E) \wedge \\
 & & left_of(B,C) \wedge does_not_touch(B,C) \wedge \\
 & & lying(A) \wedge wedge(A) \wedge standing(B) \wedge \\
 & & standing(C) \wedge brick(B) \wedge brick(C).
\end{array}$$

If we have in our theory the following rules:

$$\begin{array}{lll}
T_1: & column(X) \leftarrow & \\
 & & brick(X) \wedge standing(X) \wedge \\
 & & is_on(X,Y) \wedge ground(Y). \\
T_2: & column(X) \leftarrow & \\
 & & brick(X) \wedge standing(X) \wedge \\
 & & is_on(X,Y) \wedge column(Y). \\
T_3: & same_height(X,Y) \leftarrow & \\
 & & ground(X) \wedge ground(Y). \\
T_4: & same_height(X,Y) \leftarrow & \\
 & & brick(X_1) \wedge standing(X_1) \wedge brick(X_2) \wedge \\
 & & standing(X_2) \wedge is_on(X_1,Y_1) \wedge \\
 & & is_on(X_2,Y_2) \wedge same_height(Y_1,Y_2).
\end{array}$$

a possible generalisation of E given the domain theory (using T_1 as a rewriting rule on the body of E) is:

$$\begin{array}{lll}
G: & arch(X) \leftarrow & \\
 & & part_of(A,X) \wedge part_of(B,X) \wedge part_of(C,X) \wedge
\end{array}$$

$$is\_on(A,B) \wedge is\_on(A,C) \wedge is\_on(C,E) \wedge$$
$$ground(E) \wedge left\_of(B,C) \wedge does\_not\_touch(B,C) \wedge$$
$$lying(A) \wedge wedge(A) \wedge standing(C) \wedge brick(C) \wedge$$
$$column(B).$$

where the literals $standing(B) \wedge brick(B) \wedge ground(D) \wedge is\_on(B,D)$ have been replaced by $column(B)$.

[Muggleton and Buntine] showed that this elementary generalisation step in presence of domain theory can be formalized as an inversion of a resolution step called *Absorption*. Without recalling here the formal proof of this assertion, one can see that G resolved with T_1 yields the clause E. Thus G can be seen as the result of inverting resolution on clauses E and T_1. In [Rouveirol and Puget 90], we have recalled (in an inverse resolution context) a well known feature of this type of generalisation: it is destructive. Following our previous notations, the literals which are replaced by the head of C_t in the body of C_e to yield the generalised clause are lost. This implies, from a theoretical point of view, a *local* incompleteness of the Absorption operator: all the clauses C, which resolved with T_1 would allow to derive E are not obtainable through Absorption [Rouveirol and Puget 90] [2]. More precisely, when there exists in the domain theory partially overlapping concepts, there are several competing generalisations for E, and it is not possible to get all the possible generalisations for the initial clause E by applying sequences of elementary generalisation steps.

In our example, we have the following graph for all the possible generalisations for a part of the initial description of the arch example. For example, the literal $standing(B)$ can be used to introduce both literals $column(B)$ and $same\_height(B,C)$. Only one of these literals only can be introduced since $standing(B)$ disappears after the generalisation has been performed. So, there are two problems: the first one is the impossibility to obtain the most specific generalisation for this example, where both literals $column(B)$ and $same\_height(B,C)$ would appear. The second problem is that the control of this generalisation step becomes a crucial issue as one generalisation forbids all the other generalisations that delete the same literals in the body of the initial clause.

2.2 Elementary Saturation

[Sammut and Banerji 86] provides a solution to this control problem due to the loss of information while generalising: MARVIN asks the expert to validate each generalisation step. When the user does not validate the generalised clause, the system tries to specialize it by introducing in its body concepts that have in their preconditions some of the literals that have disappeared in the previous generalisation step. We have taken a different option to solve this incompleteness and this control issue for generalisation by introducing a reformulation step under the name of *Elementary Saturation*. Basically, elementary saturation does not replace in the body of the clause to generalise E the description of an instance of a concept by the concept identifier, but rather adds the concept identifier to the body of E.

Definition 2: Let C_e be a example clause $C_e : T_e \leftarrow Lc_e$ and let C_s be the *saturant* clause $C_s : Ts \leftarrow Lc_s$, such that $Lc_s\theta$-subsumes Lc_e (that is, $\exists\theta$ such that $Lc_e \supseteq Lc_s\theta$). We call *elementary saturation* of C_e by C_s, the following clause:

[2] There are other incompletenesses of Absorption which are detailed in [Nienhuys-Cheng and Flach 91]

$$C_h: \qquad T_e \leftarrow (Lc_e \wedge T_s\theta)$$

Given the previous example of arch, one possible elementary saturation of E is with the T_1 clause of the domain theory:

$$\begin{aligned} T_1: \qquad & column(X) \leftarrow \\ & \quad brick(X) \wedge standing(X) \wedge \\ & \quad is\_on(X,Y) \wedge ground(Y). \end{aligned}$$

It adds $column(B)$ to the body of E without removing the literals $standing(B) \wedge brick(B) \wedge ground(D) \wedge is\_on(B,D)$. This gives the new clause:

$$\begin{aligned} E_s: \qquad & arch(X) \leftarrow \\ & \quad part\_of(A,X) \wedge part\_of(B,X) \wedge part\_of(C,X) \wedge \\ & \quad is\_on(A,B) \wedge is\_on(A,C) \wedge is\_on(B,D) \wedge \\ & \quad is\_on(C,E) \wedge ground(D) \wedge ground(E) \wedge \\ & \quad left\_of(B,C) \wedge does\_not\_touch(B,C) \wedge lying(A) \wedge \\ & \quad wedge(A) \wedge standing(B) \wedge standing(C) \wedge \\ & \quad brick(B) \wedge brick(C) \wedge column(B). \end{aligned}$$

In this case, the introduction of $same\_height(B,C)$ by saturating E_s with T_2 that was impossible with the previous generalisation scheme, is here still possible. Elementary Saturation is an inverse resolution step, as E_s resolved with T_1 yields the initial clause.

2.3 Control for Elementary Saturation

When there is no information available about the relevance of predicates in a given context (for example, which predicates are interesting to introduce in the body of the clause, or anything that could control how to use of elementary saturation) we have chosen to be exhaustive. Exhaustive application of the elementary saturation operator is refered to as *Saturation*.

2.3.1 Formalization

Definition 3: Saturation operator.

Given a definite program T and a definite clause $C_e : T_e \leftarrow Lc_e$, the Saturation operator is the transitive closure of the elementary saturation operator. More precisely, it builds a clause $C_s : T_s \leftarrow Lc_s$ such that

- $$T_s = T_e$$
- $$Lc_s = (\bigwedge_{h=1}^{n_h} Lcs_h)$$

with Lcs_h defined as :

- $$Lcs_0 = Lc_e$$

- $$Lcs_1 = (\bigwedge_{i=1}^{n} (Ts_{i1}\theta_i))$$

  where the Ts_{i1} are the heads of all the clauses $Cs_{i1} : Ts_{i1} \leftarrow Lcs_{i1}$, such that $Lc_e \supseteq Lcs_{i1}\theta_i$.

- $$Lcs_i = (\bigwedge_{j=1}^{m} (Ts_{ji}\theta_j))$$

  where the Ts_{ji} are the heads of all the clauses $Cs_{ji} : Ts_{ji} \leftarrow Lcs_{ji}$, such that $Lc_{i-1} \supseteq Lcs_{ji}\theta_j$.

The process stops when no more new literals can be derived from the literals that were derived the step before[3].

Theorem 1: Saturation, given a definite logic program P and a definite clause E: $T_e \leftarrow Lc_e$, such that $P \not\models E$, produces all the most specific relevant generalisation of E given P. It is such that $H : T_h \leftarrow Lc_h$ such that:

- $T_e = T_h$
- $P \cup Lc_e \models Lc_h$.[4]

The proof can be found in the annex.

2.3.2 Algorithm and example

The algorithm 1 is constructed from the structure of the proof given in the annex.

Again, let us illustrate the algorithm on our arch example. During the first step,the literals of the body of the example are skolemized. This yields the following set of unit clauses (we only show here the literals that can take part in resolutions with the clauses of the domain theory):

$is\_on(b, d) \leftarrow .$
$is\_on(c, e) \leftarrow .$
$ground(d) \leftarrow .$
$ground(e) \leftarrow .$
$standing(b) \leftarrow .$
$standing(c) \leftarrow .$
$brick(b) \leftarrow .$
$brick(c) \leftarrow .$

[3]it may well be the case that the process never stops because of, for example, an infinite recursion. The depth of the derivation tree then has to be limited.

[4]If H is a generalisation of E given P, this means that $P, H \models E$. Then any clause H' equal to H with any additional literal Rub in its body is also a generalisation for E. We therefore restrict to relevant such generalisations, those that have in their body literals that can be derived through any number of inference steps using *exclusively* clauses from the background knowledge P.

Algorithm 1: Saturation algorithm

PROCEDURE saturation(C_e: a definite clause, T: a domain theory, C_s: a definite clause)
    $C_e : T_e \leftarrow Lc_e$
    $F = \emptyset$
    for all literals Lc_{e_i} of Lc_e **do**
        add $Lc_{e_i}\theta_s \leftarrow$ to F
    endfor
    changed := false
    repeat
        Deductive phase:
        for all clauses C of T **do**
            for all ground facts $Lc_{e_i}\theta_s \leftarrow$ of F **do**
                if all literals of body(c) can be resolved with facts of F
                then add $head(c)\theta_s \leftarrow$ to F
                    changed := true
                endif
            endfor
        endfor
    until not changed or depth-bound exceeded
    $F := \{Lc_{e_i}\theta_s \leftarrow \}$
    $C_{ss} : T_e \leftarrow Lc_{e_i}\theta_s$
    deskolemize C_{ss}, yielding C_s
    return $\{ C_s \}$
endproc

with the skolem substitution: $\{A/a, B/b, C/c, D/d, E/e\}$.

During the second deductive step, all the possible resolutions between clauses of the domain theory and the set of unit clauses that represent the skolemized body of the example, the following literals are deduced:

$$
\begin{array}{l}
same\_height(d,e) \leftarrow . \\
same\_height(b,c) \leftarrow . \\
column(b) \leftarrow . \\
column(c) \leftarrow .
\end{array}
$$

which gives, once unskolemized (the skolem substitution is inverted in the mathematical sense) and again in a conjunctive form: $same\_height(D,E) \wedge same\_height(B,C) \wedge column(B) \wedge column(C)$. The final result of saturating the example clause with the domain theory is the clause:

$$
\begin{array}{lll}
E_s: & arch(X) \leftarrow & \\
 & & part\_of(A,X) \wedge part\_of(B,X) \wedge part\_of(C,X) \wedge \\
 & & is\_on(A,B) \wedge is\_on(A,C) \wedge [is\_on(B,D) \wedge \\
 & & is\_on(C,E) \wedge ground(D) \wedge ground(E)] \wedge \\
 & & left\_of(B,C) \wedge does\_not\_touch(B,C) \wedge \\
 & & lying(A) \wedge wedge(A) \wedge [standing(B) \wedge \\
 & & standing(C) \wedge brick(B) \wedge brick(C)] \wedge \\
 & & same\_height(D,E) \wedge same\_height(B,C) \wedge \\
 & & column(B) \wedge column(C).
\end{array}
$$

Representing this clause as a graph, the root of the graph is the head of the clause, $arch(X)$, the leaves of the graph are the literals in the body of the initial clause, the nodes in between represent literals that have been infered from other literals in the body of the clause. The literals that have been used to deduce other literals are bracketed (like $standing(B)$): strictly, they are redondant in the clause [Buntine 88]. But the aim of Saturation is to make all information explicit *before* the generalisation step, in order to make inductive choices with full knowledge of what the example means. This will allow to avoid backtrack as in MARVIN. Making all information explicit allows to use a more global criterium for generalising. Relationships with other generalisation techniques in presence of background knowledge are studied in section 7. If we drop all the literals that have been used to deduce other ones, we get the following clause, which contains only high level concepts:

$$
\begin{array}{lll}
E_{s_g}: & arch(X) \leftarrow & \\
 & & part\_of(A,X) \wedge part\_of(B,X) \wedge part\_of(C,X) \wedge \\
 & & is\_on(A,B) \wedge is\_on(A,C) \wedge left\_of(B,C) \wedge \\
 & & does\_not\_touch(B,C) \wedge lying(A) \wedge wedge(A) \wedge \\
 & & same\_height(D,E) \wedge same\_height(B,C) \wedge \\
 & & column(B) \wedge column(C).
\end{array}
$$

From a practical point of view, Saturation suppresses control points in the system, more precisely it postpones any inductive choices until application of Truncation.

3 Truncation.

The Truncation is the only inductive operator of our system.

Definition 4: Truncation generalises a clause $C_e : T_e \leftarrow Lc_e$ by:

- dropping literals of Lc_e
- replacing all the occurrences of a term t of C_e by the same variable
- splitting occurrences of a term t of C_e into several independent variables

We have extended CIGOL's Truncation first to inversion of substitutions on non unit clauses [Rouveirol and Puget 89] and then to the full dropping rule [Rouveirol and Puget 90]. More precisely, Truncation can be interpreted as an *inverse θ-subsumption* step, as states the following theorem.

Theorem 2: Let C_e be a definite clause. Truncation provides all the clauses $C_t = Trunc(C_e)$ such that $C_t \theta$-subsumes C_e.

4 Coupling Saturation and Truncation

By coupling Saturation and Truncation, we are able to build a range of generalisations of an input clause given a domain theory in two separate steps, one reformulation step of the input clause given the domain theory (Saturation), and one purely inductive generalisation step (Truncation). Which generalisations can we reach (in terms of search space) by applying those two operators? It is clear that we can reach by using Saturation and Truncation all the generalisations that can be built by an algorithm implementing definition 1 (section 2). But, let us characterize more formally our search space for generalisation.

Theorem 3: Given a domain theory T, an initial clause $E : H_e \leftarrow Lc_e$ and the two operators *Saturation* and *Truncation*, we can build all the most general clauses G_i [5] $G_i : Hg_i \leftarrow Lbg_i$ such that:

- $H_e = Hg_i$
- $T, G_i \models E$

This theorem is a direct consequence of Theorems 1 and 2. Let us explain what it means. The result of Saturation is a clause $C_s : T_s \leftarrow Lc_s$ which is θ-subsumed by all the possible generalisations of C_e, according to the definition of generality given in section 1.1. We can interpret the result of Saturation as being one *relative least general generalisation* of the input clause C_e. Any clause that θ-subsumes C_s is a possible generalisation of C_e. Therefore, coupling Saturation and Truncation, allows us to build all the relevant generalisations of the example clause with respect to a domain theory.

5 Representation change for learning: flattening.

In this section, we will describe a representation change we use, *flattening*, that smoothes implementation of Truncation. Flattening allows to translate a set of clauses with function symbols into an equivalent set of clauses without function symbols.

[5] see preceding footnote

Flattening has been suggested quite often in the Logic Programming [Sterling and Shapiro 86] as well as in the Machine Learning community [Genesereth and Nilsson 87], but it was always in a very unformal way. Its basic idea is to introduce for every function symbol of arity n a new predicate 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. But we want to be sure that the semantic of the corresponding flattened logic program remains unchanged after this representation shift. Moreover, we want this representation change to be reversible. Each new predicate introduced has thus to be defined in regard of the term it replaces, so that the transformation is reversible.

Definitions:

- We call *flattening predicate* f_p associated to the functor f of arity n, the so defined predicate of arity $n+1$:

$$f_p(t_1, \ldots, t_n, X) \leftarrow X =^6 f(t_1, \ldots, t_n).$$

- We call C' the *flattened* clause corresponding to a clause C of a logic program P, the clause C where each occurrence of all a term with function symbol f has been replaced by a new variable X, whereas $f_p(t_1, \ldots, t_n, X)$ is added to the body of the clause. f_p is the flattening predicate correponding to f.

Thus, the algorithm of flattening for a logic program P is Algorithm 2:

Algorithm 2: Flattening algorithm

PROCEDURE flatten(P: definite program with function symbols,
    P_f: corresponding flattened program, $Flat\_Defs(P)$: defs of flattening predicates)

    $P_f = \emptyset$
    $Flat\_Defs(P) = \emptyset$
    for all clauses C of a logic program P **do**
        $C_f = C$
        for all functors of arity n $f(t1, \ldots, tn)$ in C **do**
            if the flattening predicate f_p corresponding to f does not exist
            then add $f_p(t_1, \ldots, t_n, X) \leftarrow X = f(t_1, \ldots, t_n)$ to $Flat\_Defs(P)$
            endif
            Replace **all** occurrences in C_f of $f(t_1, \ldots, t_n)$ by X, and add the literal $f_p(t_1, \ldots, t_n, X)$ to the body of C_f.
        endfor
        add C_f to P_f
    endfor
    return { P_f, $Flat\_Defs(P)$ }
endproc

---

[6] We use a standard equality theory, for instance the axioms defined in (Shepherdson, 1987; pp27-28)

Note that we only represent the equivalence in the definition of the flattening predicate as a simle implication because this implication is the only one that we use during the unflattening step. The set of clauses that define the flattened predicates will not take part in the other steps (i.e. Saturation, Truncation) performed by the system. They are only used to go back to a functional notation (that means, to a program that uses again the function symbols that were defined in the initial program). An unflattened clause C or logic program P is denoted respectively $unflat(C)$ and $unflat(P)$. The algorithm of unflattening is presented in algorithm 3:

Algorithm 3: Unflattening algorithm

PROCEDURE unflatten(P_f: flattened definite program,
    $Flat\_Defs(P)$: defs of flattening predicates,
    P: corresponding program with function symbols)

    $Flat\_Defs(P) = \emptyset$
    for all clauses C_f of the flattened logic program P_f) **do**
        $C = C_f$
        for all flattening predicates $f_p(t_1, \ldots, t_n, X)$ in the body of C **do**
            replace all the occurrences of X in C by the term $f(t_1, \ldots, t_n)$ by
            applying resolution as much as possible between C and clauses of
            $Flat\_Defs(P)$
        endfor
        add C to P
    endfor
    return $\{ P \}$
endproc

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:

$$E''_x: \quad member(X, Z) \leftarrow consp(X, Y, Z) \wedge bluep(X) \wedge nilp(Y).$$

with the three following clauses:

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

Observe that E''_x resolved with NIL, $CONS$ and $BLUE$ yields E_x. We claim that flattening and unflattening do not change the semantic of a logic program P and that therefore, it is equivalent, as far as the logical closure of the theory is concerned, to work on clauses with function symbols or on flattened clauses, given all the definitions of the flattened predicates corresponding to the above function symbols.

Theorem 4 [7]**:**

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

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

Corollaries:

For a definite clause C and C':

- if C is a clause with function symbols, $unflat(flat(C)) = C$.
- if C is a flat clause, $flat(unflat(C)) = C$.
- $C\theta$-subsumes C' iff $flat(C)\theta$-subsumes $flat(C')$.

At this point of description of flattening, one issue needs to be raised: do we replace each occurence of the same term by the same variable, or do we replace each occurrence of the same term by different (and independent) variables? The second approach has been developed in [Ling 89], but it introduces a lot of variables, which overloads the generalisation process. So, in our version of flattening, we replace all the occurrences of the same term throughout a clause by the same variable (in our example above, all the occurrences of the *blue* constant are replaced by the same constant X). This means that our version of flattening "sticks" to the example (no generalisation occurs while flattening), as all the bindings between arguments of the original clause are transmitted to the flattened clause. This makes the generalisation step far less combinatorial. On the other hand, this also makes the system more sensitive to coincidences in the examples, but Truncation will give the possibility to split several occurrences of the same variable into several independant variables, when for example the generalisation covers a negatice example.

To sum up, flattening has two purposes: first, it gives a uniform syntax to the internal language of the system, and that allows a simple implementation of generalisation, although of course it does not solve any of the control problems for generalisation. Secondly, our hypothesis to replace each occurrence of the same term with the same variable strongly encourages the system to find the simplest hypothesis to complete the domain theory. More complex hypothesis (with more independent variables) will be considered afterwards when the simplest hypothesis fail to complete the domain theory. At last, representing constants and terms as predicates in flattened clauses allows during saturation to introduce new predicates instantiated with constants and terms that used to appear in the initial example only (see sections 7.2 and 7.3).

5.1 Truncation on flattened clauses

Truncation amounts to drop literals in the body of the corresponding flattened clause. For example, generalising the clause:

[7] The proof of this theorem can be found in the annex

$$Ex: \quad member(blue, [blue]) \leftarrow .$$

is done by dropping the literals in the body of the clause:

$$Ex''_f: \quad member(X, Z) \leftarrow consp(X, Y, Z) \wedge bluep(X) \wedge nilp(Y).$$

Dropping *bluep(X)* and *nilp(Y)* yields the clause once unflattened:

$$Ex'': \quad member(X, [X \mid Z]) \leftarrow .$$

A possible application of the splitting variable rule here is to split the two occurrences of X into two possibly different variables x and W, yielding the clause:

$$Ex_{tg}: \quad member(X, [W \mid Z]) \leftarrow .$$

Here, it leads to a too general clause and it should not be applied.

6 Elements of Control for ITOU

6.1 ITOU algorithm

Given an incomplete but complete domain theory (see section 1.1), ITOU first performs a subsumption test : does the theory subsumes (in the logical implication meaning) the example? The subsumption test has in fact the same structure as Saturation (it exploits the subsumption theorem whose proof can be found in the annex). Therefore the subsumption test, given an example and a domain theory, starts skolemising the body of the example, then it performs forward chaining on the equivalent set of unit clauses. If the skolemised head of the example can be deduced, the example is then subsumedotherwise not. This subsumption test suffers from the same limitations as Saturation: it terminates if there are no infinite recursive calls in the theory. If there are some, it is always possible to bound the depth of the derivation graph, and once this bound is reached, the example is considered unexplained.

If the example is not subsumed by the theory (we only address this issue here), the system generates a new theory T' such that T' logically entails both the initial theory and the failure example. There are of course plenty of theories that satisfy these two conditions. The simplest one is the initial theory plus just the unrecognized example. It is clear that we are looking for some more predictiveness for the new theory. Therefore, ITOU generalises the failure example before integrating it to the domain theory, by applying first Saturation and then Truncation. The search space for Truncation even for a simple input clause is very large, so Truncation is not applied on a single clause but takes several clauses as input. The other clauses that will be generalised with the saturated input clause are either other saturated input clauses instances of the same concept, in case the system has access to several examples at the same time, or other clauses of the domain theory that describes the same concept has the example. Truncation is therefore limited to dropping literals that are not common to the input clauses.

Algorithm 4: ITOU algorithm

PROCEDURE ITOU(E : pos. example = definite clause,
    T : background knowledge = definite program)

    Call $flatten(T, T_f, Flat\_Defs(T))$
    Call $flatten(E, E_f, Flat\_Defs(E))$
    Call $saturation(E_f, T_f, E_s)$
    if the skolemised head of the example has been derived during Saturation
    then Stop
    else Look for clause(s) C_i of T_f that have the same head as E_s
        repeat
            Choose k clauses Cg_k among C_i
            Compute G, a purely inductive generalisation of Cg_k and E_s
            Provide G to user for validation
        until user validates G or no more choices for Cg_k
        Replace Cg_k by G in T_f call $unflatten(T_f, Flat\_Defs(T), T)$ **return** { T }
    endif
endproc

6.2 Generalisation algorithm

We first started to use the *least general generalisation* algorithm [Plotkin 70] (denoted *lgg*) as a typical purely inductive generalisation step. But, most of the time, *lgg* is too conservative a strategy for generalisation, the so obtained generalisation is too specific. There are a lot a cases where some additional information is available in the form of bias (such as those described in sections 6.3.1 and 6.3.2) that allows us to generalise more with a higher degree of confidence. So, we have adapted the *lgg* algorithm to the top down generalisation algorithm that follows (algorithm 5). First, the heads of the clauses are generalised. Then, each selection in the body of the two input clauses are considered. The literal which is the generalisation of the above selection is added to the current partial generalisation if the resulting new current generalisation satisfies a set of constraints. If the obtained partial generalisation satisfies the stopping condition the algorithm stops, else the generalisation iterates on the next selection. Each generalisation satisfying a given set of constraints is proposed to the user for validation. A validated generalisation is added to the initial domain theory, otherwise the system backtracks to another possile generalisation. If none of the possible generalisations ha sbeen validated by the user, the system will just add the example to the domain theory and it will wait for further information (i.e., instances of the same concept) to generalise.

6.3 Control for the generalisation step

This Truncation operator has of course to be carefully controlled, because it operates on saturated clauses, that can be very large. There have been a lot of biases for generalisation, mostly syntactic, such as constraints on the form of generalisation (simplicity evaluated

Algorithm 5: Generalisation algorithm

PROCEDURE gen(C_1, C_2, G: definite flattened clause
    S_c: set of constraints, $Stop\_Cond$: stopping condition)
    $C_1 : H_1 \leftarrow Lb_{1i}$
    $C_2 : H_2 \leftarrow Lb_{2j}$
    compute $H_g = lgg(H_1, H_2)$
    $G_c = \emptyset$ partial generalisation of the bodies of C_1 and C_2
    for all pairs of literals (Lb_{1_i},Lb_{2_j}) of $body(C_1)$ and $body(C_2)$ **do**
        compute $Lb_{gk} = lgg(Lb_{1_i}, Lb_{2_j})$
        if $G_c \cup Lb_{gk}$ satisfies S_c
        then $G_c := G_c \cup Lb_{gk}$
        endif
        if G_c satisfies $Stop\_Cond$
        then return $\{ G_c \}$
        endif
    endfor
endproc

in regard of the number of variables, number of literals, etc...), evaluation in terms of generality (least or most general generalisation) or any domain dependent characteristic (see [Kodratoff 88]) for a survey about generalisation techniques and biases). Sometimes, counter examples are also used to restrict the space of possible generalisations. We present here two biases, based on semantic properties of the described domain. Let Sc be a set of constraints that the final generalisation should satisfy. Sc can be split into Sc_c, that is, constraints that can be evaluated while the generalisation is build, on each generated partial generalisation, and into Sc_a that are stopping conditions for the generalisation algorithm.

6.3.1 Connection

Definitions:

- Let there be a clause $C : T(V_t) \leftarrow C_o(Vc_o)$. A variable $X$is connected to a variable T in C iff:
    1. $Y \in Connd(X)$, or
    2. $Z \in Connd(X)$ and $Z \in Connd(Y)$.
- A literal L is connected to a variable X if at least one of its arguments is connected to X.
- A clause C is *strongly connex* if both following conditions hold:
    1. All the literals in its body are connected to a variable which is one of its head arguments.

2. If the head predicate has more than one argument, and if C is not a unit clause, all the variables in the head of te clause must be connected to each other through a path of predicates in the body of the clause.

- A clause C is *connex* iff only the first of the two above conditions holds. We will refer to only connex clauses in the remainder of the paragraph.

The connection principle states that variables in the body of a clause must eitherappear in the head of the clause, or be linked to a variable of the head of the clause through a path of predicates. This ensures that all literals in the body of a clause are related to its head, or that all the conditions for an instance to belong to a concept are related to the concept itself. [Morik 89] lists some common errors in knowledge modeling that justify the use of connection. We have proved [Rouveirol 91b] that for each non connex clause, there exists a connex clause with the same least Herbrand model, which gives a formal justification to our restriction to connex clauses.

The input of the generalisation algorithm is restricted to connected clauses, and the generalisation of two connected clause has to be connected. This allows to limit the number of variables in the generalised clause. Formalization and examples of use of connection can be found in [Rouveirol 91b]. There exists a Directed Acyclic Graph for the variables of a connex clause. This DAG is used for the incremental computation of generalisation of connex clauses.

Heuristics similar to the connection have been already used: linked clauses of [Helft 89]. [De Raedt 91] developed this idea or constraining generalisation by restricting the language of the generalisation by defining a series of language. in the first and simplest language, all the variables appearing in the body of clauses must appear in the head (no existential variables allowed in the body of the clauses), and then in the next languages of the series, the number of existential variables allowed in the clauses increases. This can be seen as a generalisation of the connection heuristic.

6.3.2 Functional relation

The functional relation principle is a semantic bias. For some predicates, we know that instantiation of some of its arguments uniquely fixes instantiations for its other arguments. For example, an object has only one position, people have only one social security number, one father and one mother, etc... The formal definition close to the one after [Puget 89] is the following one.

Definition: Let P be a logic program, let $pred(X,Y)$ be a predicate where X and Y are two variables vectors defined in P. $pred(X,Y)$ is a functional relation for X if $pred(X,Y)$ verifies :

$$T + equalitytheory \models \\ (\forall X, Y, Z, pred(X,Y) \wedge pred(X,Z) \Leftrightarrow Y = Z)$$

The notion of functional relation is related to that of the functional determination [Russell 89]. Functional relations are indeed frequent: for example, in object oriented languages, a mono-valued slot defines a functional relation of a slot value and the object (for example, the position of a piece on the board for board games, etc...).

Let us see on a small example how knowledge of functional relationship can guide the generalisation process. $plus(X,Y,Z)$ is a functional relation of two of its arguments. Indeed, we can verify, for example, that given values for X and Y, there is only one possible value for Z. If we know that two clauses to be generalised are instances of a concept that is a functional relation, each candidate generalisation for these two clauses will have to fulfill these constraints.

For example, let us imagine that a candidate generalisation for two clauses that describe addition instances is:

$$plus(X,U,V) \leftarrow plus(X,Y,Z) \wedge succ(Y,U).$$

We wish to check whether this generalisation verifies the functional relation of addition. Let us take any of the three functional relation constraint given for addition, for example:

> Given $plus(X,Y,Z)$, if Y and Z are instantiated, there is only one possible instantiation for X.

We also need the knowledge that $succ(X,Y)$ is a functional relation for one any of its arguments.

To check the above functional relation contraint on our example, we will instantiate U and V (more precisely, mark U and V as instantiated, because we are not interested in the real values of U and V, but rather on the uniqueness of these values) and then observe the propagation of these instantiations on the rest of the clause. Nothing can be propagated through the first literal in the body of the candiate generalisation, $plus(X,Y,Z)$, because there is no variable marked as instantiated in $plus(X,Y,Z)$, so no other variables can be marked. Shifting to the second literal of the body, $succ(Y,U)$, U is known as being instantiated; as $succ$ is a functional relation of any of its arguments, if U is instantiated, there is only one possible value for V. But this is not sufficient for us to know about the unicity of value for Z, because nothing more can be propagated through the clause (the only one variable is marked instantiated in $plus(X,Y,Z)$ is Y, and we need at least two marked variables to infer anything more).

Another candidate generalisation is:

$$plus(X,U,V) \leftarrow plus(X,Y,Z) \wedge succ(Y,U) \wedge succ(Z,V).$$

If we check the same functional relation constraint, it starts as in the previous example, but for $succ(Z,V)$, V is set and sets Z as well. Therefore, are two marked arguments in $plus(X,Y,Z)$, Y and Z, and then X becomes marked. The functional relation is verified as well as the two others for addition, this generalisation is thus proposed to the user. This is one of the recursive definitions for addition.

Just to give an idea about the power of these two biases, on this example for synthesizing addition from examples (see [Rouveirol 90]), our basic generalisation algorithm computed approximately 1100 candidate generalisations without using any bias, 70 candidate generalisations after using the connection bias, and only one of them were left after applying also the functional relation bias.

7 Related works

7.1 CIGOL

Our starting point when designing the predecessor of ITOU, IRES [Rouveirol and Puget 89] was to extend the CIGOL system [Muggleton and Buntine] which was the first system based on inversion of resolution (I.R.). The idea of I.R. seems really elegant because it provides a single mechanism that covers many aspects of induction such as purely inductive generalisation (Truncation), generalisation in presence of a background knowledge (Absorption), introduction of new predicates (Intraconstruction). The idea of inverting a deductive process to perform induction was not new [Plotkin 71] but was for the first time so clearly stated as the basis for an inductive system.

However, CIGOL suffered from many limitations. In order to give an hint of the complexity of CIGOL's method to solve inversion of resolution, we give, after [Muggleton and Buntine], the formula used for inverting a single step of resolution, namely Absorption.

C_1 and R are the two given clauses, the formula that computes all the clauses C_2 (there can be more than one solution to this formula, that's why the term inversion must not be taken in all its mathematical interpretation) such that R is the resolvent of C_1 and C_2 is: $C_2 = (R - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{L_2\}$ (1)

L_1 and L_2 are the two resolved literals in C_1 and C_2(that disappear during the resolution step) θ_1 and θ_2 are the associated substitutions ($L_1\theta_1 = L_2\theta_2$), and θ_2^{-1} is the inverse substitution of θ_2. The syntactic form of C_1, C_2 or R has to be constrained so that solving this equation becomes manageable in a reasonable time. On the one hand, CIGOL accepted clauses with function symbols but restricted the input C_1 to be a unit clause. In this case, $(C_1 - \{L_1\}) = \emptyset$, and the formula (1) looks more friendly.

In our first re-implementation of CIGOL [Rouveirol and Puget 89], we chose to solve inversion of resolution for first order languages without function symbols. In these languages, substitutions are pure variable substitutions only, which makes the computation of inverse substitutions a lot easier. To get rid of the limitation of working with a function free first order languages only, we have introduced flattening and unflattening in order to work indifferently in functional or flattened representation. Combination of flattening/unflattening with our simplified I.R. operators allowed us to get rid of the unit clause limitation of CIGOL. Secondly, concerning Absorption and Intraconstruction, the hypothesis of separability was made in CIGOL: parent clauses cannot have non trivial factors (because of the incompleteness of Absorption described in section 2.1). By implementing Saturation, we got rid of the separability assumption of CIGOL.

Working on flattened representations allowed us to implement I.R. operators that were simpler than CIGOL's ones. Some other limitations of I.R. then became apparent: the first one was that the dropping rule was not covered by the I.R. scheme. The second one was the incompleteness and the difficulty to control the Absorption step, this control point being crucial as Absorption is destructive. We then introduced a new definition to the generality relation, and as a consequence a new framework for induction of definite programs, Inversion of logical implication [Rouveirol 91a]. This new framework can account for the dropping rule way of generalising and it allowed us as well to elegantly introduce a non destructive Absorption that simplifies control for generalisation. In CIGOL, and in our first rewriting of the CIGOL system [Rouveirol and Puget 89], there were choices about which absorptions to perform given one example clause and a domain theory, and then other choices for the Truncation operator. With Saturation, no choice is performed until the application of Truncation. Instead of choosing which Absorption and then which Truncation to perform among all those that are possible for two clauses, the only choice point will be that of Truncation, with all information available, which allows to make the right inductive choice.

7.2 P.G.A.

The Plausible Generalisation Algorithm [Buntine 87] generalises a pair of ground facts with respect to a given background knowledge. The algorithm first builds a h-easy ground finite model of the background knowledge. P.G.A. then builds the generalisation of all the pairs of these facts, provided that there are relevant to the concept to be learned and that they don't introduce in the generalisation existential variables of depth greater than a fixed bound. The obtained generalised clause is then pruned so that there are no redundant literals in its body. P.G.A. is close to our approach (even it is even closer to the GOLEM approach) but the main difference is that we only add to the body of the example clause when performing generalisation literals that are derivable both from the background knowledge *and* from the body of the example. This is one possible non heuristic criterion for relevance. By that, we restrict the number of literals that will appear in the body of the generalisation clause.

7.3 GOLEM

GOLEM [Muggleton and Feng 92] builds *relative least general generalisations* (denoted *rlgg*), that is lgg in presence of background knowledge, of ground atoms. To perform this generalisation in presence of background knowledge, GOLEM, as P.G.A., computes a finite model of the background knowledge. The atoms belonging to this model form the body of the example clauses, whose head are the ground input facts. Then, the usual lgg algorithm is performed. The problem when applying this method is that the body of the example clauses as well as the body of the resulting lgg may grow very large if not infinite ([Buntine 87] reports such problems). GOLEM therefore works with strongly generative domain theories [8] that have the properties that the h-easy model of such theory is finite. but stil, the length of the *rlgg* of two atoms grows polynomially in the size of the completed example clauses. In order to have the size of lgg grows, GOLEM builds ij-determinate subpart of the *rlgg* that covers the most examples (we refer to [Muggleton and Feng 92] for more details about ij-determination). Ij-determination, as our funtional relation bias, uses the property that some predicates in the body of the generalised clause, given one instantiation of their input arguments, there is only one possible instantiation of their output arguments. Ij-determination also includes a bound on the depth of existential quantification for a variable and a bound on the number of literals needed for one literal in the body of the generalised clause to be determined. Given the ij-determination heuristic, GOLEM performs efficiently on large amount of data.

Muggleton [Muggleton 92] made a theoretical link between *lgg* and inverse resolution by proposing the ν^n operator. The ν^n operator maintains a set of clauses that are obtainable by applying at most n most specific inverse resolution steps (a most specific inverse resolution step is similar to our elementary saturation). The difference is that our Saturation operator is that elementary saturation is applied exhaustively: therefore it provides only one clause as a result that contains implicitly all the clauses maintained by the ν^n operator. It is by applying Truncation that ITOU can build all the clauses built by ν^n, although it will of course never build them all. [Muggleton 92] reports a completeness result for the ν^n operator similar to ours but restricted to strongly generative clauses.

[8] i.e., made of clauses where each variable appears in at least two literals of the clauses

7.4 Structural Matching

Our approach is also related to that of Structural Matching [Kodratoff and Ganascia 86]. Let us compare ourselves to latest implementations of it, namely OGUST [Vrain 90] and [Bisson 90]. OGUST uses a generalisation algorithm of the same type as the one described in destrgen, but it was guided by the similarities between objects of several examples, whereas MARVIN performs on a single example only. The main difference between OGUST and ITOU is the clear separation between the reformulation step and the generalisation step while generalising in presence of background knowledge. Another system, KBG, based on structural matching [Bisson 90] performs as ITOU a saturation step. ITOU and KBG differs in the way to control the purely inductive step of generalisation: ITOU uses constraints describing properties of the concept to learn or generalisation language restrictions, whereas KBG chose the literals to pair during the matching step of the generalisation driven by the computation of a numeric similarity measure between objects in the examples to generalise.

7.5 Top down approaches to completion of domain theories

The approaches we have presented up to now are bottom-up views of induction (it is particularly clear with Saturation that builds a deduction graph from the skolemized body of the example). They all take an example and try to integrate it to the domain theory. They generalise the example, rewrite it in terms of concepts already defined in the theory, introduce new terms in the vocabulary and use them to reformulate the domain theory. They are however different systems that have been more influenced by the Explanation Based Learning and that perform top down induction by completing partial proof trees [Wirth 88], [Wirth 89], [Duval and Kodratoff 90].

They proceed in two steps: the first step is the construction of the partial proof tree, i.e., the choice of the failed proof tree that is going to be completed among all those possible ones. This choice determines the RHS of the rule to be added to the theory. It has to be guided by domain specific heuristics or it requires access to an oracle. The second part is the completion of the failed proof tree. Wirth's system, LFP, uses inverse resolution, whereas [Duval and Kodratoff 90] do not. Both systems assumes that the proof can be completed by adding one clause only to the theory.

The choice of the failed proof tree to complete is one the major difficulty of the method. However, coupling this kind of method with ours would allow to get a more substantial completeness result as the one we have for Saturation. Our Saturation produces a clause that has the same head as the example clause. Performing backward chaining on the head of the example (which is done by the top down approaches) would allow to overcome this limit. The control for coupling the two approaches would on the other hand be very complex.

7.6 CLINT

We find similar ideas as those developed in the paper in the CLINT system [De Raedt 91]. CLINT has a step similar to Saturation, called computation of a starting clause, that builds most specific generalisations of a given clause in a given language. The main difference is that it does not build this starting clause as we do by making inferences on the body of the initial example using the background knowledge. In the beginning (that is before seing any positive or negative example), the starting clause is the most specific clause in the language (this clause is finite as CLINT only deals with finite language). This clause is generalised by dropping as much literals as possible so that the new generalised clause covers more positive examples but still does not cover any negative examples. CLINT also uses integrity constraints (these contraints are represented as general clauses) to guide the search for generalisation: clauses in the knowledge base must satisfy the integrity constraints. If a constraint is not satisfied, CLINT will use it to generate new examples that will be classified as positive or negative by the user. Those examples are then used to generalise or specialise the faulty concepts. CLINT gives a very good demonstration of how additional information (syntax of the target language, integrity contraints) can be used to guide simple generalisation and specialisation operators.

8 Conclusion

We have presented a decomposition in two elemetary steps of a well known problem in Concept Learning, generalisation in presence of background knowledge. the first step, called *Saturation* reformulates an initial example given some background knowledge. Saturation indeed computes by using sound steps only the most specific relevant generalisation of an initial clause, given a domain theory. The second operator, *Truncation*, is a purely inductive generalisation operator that computes, given an initial clause E, all the clauses that θ-subsume E. By combining these two operators, we can compute all the relevant generalisations of an initial clause, given a domain theory. We then propose two constraints, one purely syntactical (i.e., which set a constraint on the form of the generalisation we are looking for) but with interesting semantic properties and a second one that express a property rather than just the syntax of the searched generalisation. These two kinds of declarative knowledge will allow to efficiently guide the generalisation process.

There are many further directions for future research. First, we would like to extend our approach to other tasks in Machine Learning than generalisation, such as specialisation and introduction of new terms. This framework for induction is a sound basis to study learnability results, in particular, what kind of programs are learnable (PAC-learnable or identifiable in the limit), given a set of elementary operators and a set of constraints.

Acknowledgements: I would like to thank Yves Kodratoff for his support and the suggestions he has provided to this work. Special thanks to all the careful readers of my PhD thesis, which is the basis of this paper, in particular J.G. Ganascia and M. Bruynooghe. At last, I would like to thank more particularly J.F. Puget for his advice and help during all the stages of this work and L. de Raedt for many helpful discussions. This work has been partially supported by CEC, through the ESPRIT-2 contrat MLT (n2154), ECOLES (n3069).

ANNEX
Proofs of

- Theorem 1 and soundness of skolemization
- Theorem 4: semantics conservation for flattening/unflattening

Proof of theorem 1:

The proof of theorem 1 requires an intermediary result stating that skolemisation of definite clauses (that is, replacing each variable in a clause by a new constant) and it reverse process deksolemisation are sound processes.

Theorem (Soundness of skolemization): Let T be a definite logic program and $\mathcal{L}$ the language of T . Let $F(X_1, \ldots, X_n)$ be a n-ary formula of $\mathcal{L}$, where $X_1, \ldots, X_n$ are the variables of F.

$$T \models \forall X_1, \ldots, X_n F(X_1, \ldots, X_n) \text{ iff}$$
$$T \models F(c_1, \ldots, c_n)$$

where $c_1, \ldots, c_n$ are constants that do not appear in T.

For the proof, we take F as unary. By induction on the number and complexity of the arguments of F, we will get the proof in the general case.

(1) We first prove that

$$(T \models \forall X F(X)) \Rightarrow (T \models F(c))$$

c being a constant which does not belong to the vocabulary of $\mathcal{L}$.

Let us reduce this first implication to the absurd. Let us suppose we have $T \models \forall X F(X)$ and $(T \not\models F(c))$. $T \models \forall X F(X)$ **(assertion 1)** means that each model $\mathcal{I}$ with domain $\mathcal{D}$ of T is also a model of F, that is for any assignation of a variable $X$to an element d of $\mathcal{D}$, the image of $F(X)$ by $\mathcal{I}$ has for truthvalue $True$.

$(T \not\models F(c))$ **(assertion 2)** means that there exists at least one model of T which is not a model for $F(c)$. In other words, there exists at least one interpretation $\mathcal{I}_\wr$ for T such that the assignation of c to an element d_o of the domain $\mathcal{D}_\wr$ of $\mathcal{I}_\wr$ is such that the image of $F(c)$ by $\mathcal{I}_\wr$ has for truthvalue $False$.

The second assertion is impossible, given the first one. Let us consider the $\mathcal{I}_\wr$ interpretation previously quoted and the $\mathcal{I}'_\wr$ interpretation, which is $\mathcal{I}_\wr$ restricted to $\mathcal{L}$. It is clear that $\mathcal{I}'_\wr$ is a model of T and of $\forall X F(X)$. Therefore, any assignation of a variable X to an element d_o in $\mathcal{D}_\wr$ makes the image of $F(X)$ true in $\mathcal{I}'_\wr$. In the same way, any assignation of a new constant c to an element d_o of $\mathcal{D}_\wr$ by $\mathcal{I}'_\wr$ makes the image of $F(c)$ by $\mathcal{I}'_\wr$ true, which contradicts the second assertion. □.

(2)

$$(T \models F(c)) \Rightarrow (T \models \forall X F(X))$$

We reduce this implication to the absurd as well. So let us suppose that $T \models F(c)$ and that $T \not\models \forall X F(X)$

$T \models F(c)$ means that in any model $\mathcal{M}$ of T, there is a constant assignation that associates c with an element d of the domain $\mathcal{D}$ of $\mathcal{M}$ such that the image F' of F by $\mathcal{M}$ applied to d has for truth value $True$.

$T \not\models \forall X F(X)$ means that there exists a model of T, $\mathcal{M}_\wr$ which is not a model of $\forall X F(X)$, that is, in this $\mathcal{M}_\wr$ model, there exists one constant c which is assigned to an element d_o of the domain $\mathcal{D}$ such that the image F' of F by $\mathcal{M}_\wr$ applied to d_o is false.

Let us consider the interpretation $\mathcal{M}'$ identical to $\mathcal{M}_{\wr}$ except for the following constant assignation: c is associated to d_o in $\mathcal{D}$. $\mathcal{M}'$ is always a model for T (because c does not belong to the set of constants of T, the assignation of the constants of T remains unchanged) but is not anymore a model for $F(c)$, which contradicts the fact that all models of T are models of $F(c)$. $\square$

Proof of theorem 1 (continued)

As we have shown just before, skolemization and its reverse process, unskolemization of the example are logically sound due to the previous theorem.

We can proceed now to the theoretical study of saturation. We have a clause:

$$C_e : T_e \leftarrow Lc_{e_1} \wedge Lc_{e_2} \wedge \ldots \wedge Lc_{e_n}$$

and a domain theory T such that $T \not\models C_e$. Let us suppose for simplicity that saturation looks for an hypothesis in the form of a single clause :

$$C_h : T_h \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m} \text{ such that } T, C_h \models C_e,$$

$$T, (T_h \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m}) \models T_e \leftarrow Lc_{e_1} \wedge Lc_{e_2} \wedge \ldots \wedge Lc_{e_n}$$

can be rewritten into:

$$T, \neg(T_e \leftarrow Lc_{e_1} \wedge Lc_{e_2} \wedge \ldots \wedge Lc_{e_n}), (T_h \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m}) \models \square$$

that gives, when developing the first clause:

$$T, \exists(\neg T_e \wedge Lc_{e_1} \wedge Lc_{e_2} \wedge \ldots \wedge Lc_{e_n}), (T_h \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m}) \models \square$$

which yields, by skolemization, (θ_s is a skolem substitution):

$$T, (\neg T_e \wedge Lc_{e_1} \wedge Lc_{e_2} \wedge \ldots \wedge Lc_{e_n})\theta_s, (T_h \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m}) \models \square$$

The negation of the skolemized example can be replaced by a set of unit clauses, the previous formula then becomes:

$$\begin{array}{l} T, \leftarrow T_e\theta_s, Lc_{e_1}\theta_s \leftarrow, Lc_{e_2}\theta_s \leftarrow, \ldots, Lc_{e_n}\theta_s \leftarrow, \\ \qquad (T_h \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m}) \models \square \ (2) \end{array}$$

We are looking for an hypothesis which has the same head as the example. We obtain such an hypothesis by unifying the example head and the negated hypothesis head, which is done by performing one and only one resolution between the negated head of the example and the head of the searched hypothesis. In other words, the refutation we are looking for starts with the resolution between $\leftarrow T_e\theta_s$ and $(T_h \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m}\ )$.

(2) becomes:

$$T, Lc_{e_1}\theta_s \leftarrow, Lc_{e_2}\theta_s \leftarrow, \ldots, Lc_{e_n}\theta_s \leftarrow, \neg(Lc_{h_1} \wedge \ldots \wedge Lc_{h_m})\theta_s \models \square$$

that can be rewritten into:

$$T, Lc_{e_1}\theta_s \leftarrow, Lc_{e_2}\theta_s \leftarrow, \ldots, Lc_{e_n}\theta_s \leftarrow) \models \exists^9(Lc_{h_1} \wedge \ldots \wedge Lc_{h_m})\theta_s$$

Let us split this implication into m ones:

$$T, Lc_{e_1}\theta_s \leftarrow, Lc_{e_2}\theta_s \leftarrow, \ldots, Lc_{e_n}\theta_s \leftarrow) \models \exists Lc_{h_1}\theta_s$$

$$T, Lc_{e_1}\theta_s \leftarrow, Lc_{e_2}\theta_s \leftarrow, \ldots, Lc_{e_n}\theta_s \leftarrow) \models \exists Lc_{h_i}\theta_s$$

$$T, Lc_{e_1}\theta_s \leftarrow, Lc_{e_2}\theta_s \leftarrow, \ldots, Lc_{e_n}\theta_s \leftarrow) \models \exists Lc_{h_m}\theta_s$$

and let us study the i-th one.

We use the following lemma, whose proof can be found in (Rouveirol, 1991).

Lemma: Let S be a satisfiable definite program and $\leftarrow B$ a unit goal, such that $S \cup \leftarrow B$ is unsatisfiable. There exists a refutation of $S \cup \leftarrow B$ where there is only one resolution with $\leftarrow B$.

By applying this lemma, we can conclude that there exists for all $Lc_{h_i}\theta_s$ a refutation of $T, Lc_{e_1}\theta_s \leftarrow, Lc_{e_2}\theta_s \leftarrow, \ldots, Lc_{e_n}\theta_s \leftarrow, \leftarrow Lc_{h_i}\theta_s$ where $\leftarrow Lc_{h_i}\theta_s$ is used only once, in other words, $Lc_{h_i}\theta_s$ can be derived by forward chaining on the skolemized body of the example, given the domain theory T.

To get back to a non skolemized form of the hypothesis body, we add on both sides of the logical implication the literal $\neg T_e\theta_s$, we get the following implication:

$$T, (\neg T_e \wedge Lc_{e_1} \wedge Lc_{e_2} \ldots \wedge Lc_{e_n})\theta_s \models \exists(\neg T_e \wedge Lc_{h_1} \wedge \ldots \wedge Lc_{h_m})\theta_s$$

that gives, the LHS and RHS of the logical implication rewritten as clauses:

$$T, \neg(T_e \leftarrow Lc_{e_1} \wedge Lc_{e_2} \ldots \wedge Lc_{e_n})\theta_s \models \exists\neg(T_e \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m})\theta_s$$

The LHS of the implication is itself logically implied by:

$$T, \neg(T_e \leftarrow Lc_{e_1} \wedge Lc_{e_2} \ldots \wedge Lc_{e_n})$$

So,

$$T, \neg(T_e \leftarrow Lc_{e_1} \wedge Lc_{e_2} \ldots \wedge Lc_{e_n}) \models \exists\neg(T_e \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m})\theta_s$$

At last, by applying the subsumption theorem, we can deskolemize the RHS of the logical implication, which provides us with the following result:

$$T, \neg(T_e \leftarrow Lc_{e_1} \wedge Lc_{e_2} \ldots \wedge Lc_{e_n}) \models \neg(T_e \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m})$$

in other words:

$$T, T_e \leftarrow Lc_{h_1} \wedge \ldots \wedge Lc_{h_m} \models T_e \leftarrow Lc_{e_1} \wedge Lc_{e_2} \ldots \wedge Lc_{e_n}. \ \Box$$

Theorem 4:

$$P \models A \text{ iff } flat(P), Flat\_defs(P) \models flat(A),$$

where $Flat\_defs(P)$ 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, c is a constant. The flattening predicate of the constant c is:

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

In order to prove the theorem, we have to prove that:

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

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

(1): It 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:

$$c_p(X) \leftarrow X = c. \text{ (i)}$$

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:

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

we get the following clause:

$pred(c)$. □

(2): We reduce this to the absurd. Let us suppose we have $pred(c)$ and that we don't have $pred(X) \leftarrow c_p(X)$. That would mean that:

$$\exists a, \neg pred(a) \wedge c_p(a).$$

However, by definition, $c_p(X) \leftrightarrow X = c$, therefore $a = c$. From two previous assertions, we can derive $\neg pred(c)$, which contradicts $pred(c)$. □

References

[Bisson 90] G. Bisson. Kbg: A knowledge base generalizer. In B. Porter and R. Mooney, editors, *Proceedings of the seventh International Conference on Machine Learning*, pages 9–16. Morgan Kaufmann, 1990.

[Buntine 87] W. Buntine. Induction of horn-clauses : methods and the plausible generalization algorithm. *International Journal of Man-Machine Studies*, 26:499–520, 1987.

[Buntine 88] Wray Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375–399, 1988.

[De Raedt 91] L. De Raedt. *Interactive concept learning.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1991.

[Duval and Kodratoff 90] B. Duval and Y. Kodratoff. A tool for management of incomplete theories: Reasoning about explanations. In P.B. Brazdil and K. Konolige, editors, *Machine Learning, Meta-Reasoning and Logics*, pages 135–158. Kluwer Academic Publishers, 1990.

[Genesereth and Nilsson 87] M. Genesereth and N. Nilsson. *Logical foundations of artificial intelligence.* Morgan Kaufmann, 1987.

[Helft 89] Nicolas Helft. Induction as nonmonotonic inference. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 149–156. Morgan Kaufmann, 1989.

[Kodratoff 88] Y. Kodratoff. *Introduction to Machine Learning.* Pitman, 1988.

[Kodratoff and Ganascia 86] Y. Kodratoff and J.G.G. Ganascia. Improving the generalization step in learning. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*, volume 2, pages 215–244. Morgan Kaufmann, 1986.

[Ling 89] X. C. Ling. Learning and inventing of horn clause theories - a constructive method. In Z.W. Ras, editor, *Methodologies for Intelligent Systems, 4*. North Holland, 1989.

[Lloyd 87] J.W. Lloyd. *Foundations of Logic Programming, second extended edition.* Springer Verlag, 1987.

[Morik 89] K. Morik. Sloppy modeling. In Katharina Morik, editor, *Knowledge Representation and Organization in Machine Learning*, volume 347 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1989.

[Muggleton 92] S. Muggleton. Inverting implication. In *Proceedings of the Second Inductive Logic Programming Workshop*, pages 19–39, 1992. ICOT (Technical report TM-1182).

[Muggleton and Buntine] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In Morgan Kaufmann, editor, *Proceedings of the 5th International Workshop on Machine Learning*, pages 339–352.

[Muggleton and Feng 92] S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

[Nienhuys-Cheng and Flach 91] S. H. Nienhuys-Cheng and P. A. Flach. Consistent term mapping, term partitions and inverse resolution. In Kodratoff Y., editor, *Machine Learning - EWSL91: European Working Session on Learning.* Springer Verlag (482), 1991.

[Plotkin 70] G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5. Edinburgh University Press, 1970.

[Plotkin 71] G. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6. Edinburgh University Press, 1971.

[Puget 89] J.F. Puget. Learning concept negation. In *Proceedings of the 4th European Working Session on Learning*, pages 179–188. Pitman, 1989.

[Rouveirol 90] C. Rouveirol. Saturation: Potsponing choices for inverting resolution. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 557–562. Pitman, 1990.

[Rouveirol 91a] C. Rouveirol. Completeness of inductive procedures. In *Proceedings of the 8th International Workshop on Machine Learning.* Morgan Kaufmann, 1991.

[Rouveirol 91b] C. Rouveirol. *ITOU: Induction de Theories en Ordre Un.* PhD thesis, Laboratoire de Recherche en Informatique, Universite Paris XI, 1991. in French.

[Rouveirol and Puget 89] C. Rouveirol and J.-F. Puget. A simple solution for inverting resolution. In K. Morik, editor, *Proceedings of the 4th European Working Session on Learning*, pages 201–210. Pitman, 1989.

[Rouveirol and Puget 90] C. Rouveirol and J.-F. Puget. Beyond inversion of resolution. In *Proceedings of the 7th International Conference on Machine Learning.* Morgan Kaufmann, 1990.

[Russell 89] S.J. Russell. *The use of knowledge in analogy and induction.* Pitman, 1989.

[Sammut and Banerji 86] C. Sammut and R. Banerji. Learning concepts by asking questions. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*, volume 2, pages 167–192. Morgan Kaufmann, 1986.

[Sterling and Shapiro 86] L. Sterling and E. Shapiro. *The Art of Prolog, Advanced Programming Techniques.* Cambridge, MA:MIT Press, 1986.

[Vrain 90] C. Vrain. Ogust: A system that learns using domain properties expressed as theorems. In Y. Kodratoff and R.S. Michalski, editors, *Machine Learning : an artificial intelligence approach*, volume 3, pages 360–382. Morgan Kaufmann, 1990.

[Wirth 88] R. Wirth. Learning by failure to prove. In D. Sleeman, editor, *Proceedings of the 3rd European Working Session on Learning.* Pitman, 1988.

[Wirth 89] R. Wirth. Induction of logic programs. In K. Morik, editor, *Proceedings of the fourth European Working Session on Learning.* Pitman, 1989.