UNIVERSIDADE DE SÃO PAULO

Inverting resolution: Plotkin's least general generalization

MARIA DO CARMO NICOLETTI
MARIA CAROLINA MONARD

Nº 130

NOTAS

Instituto de Ciências Matemáticas de São Carlos

ISSN - 0103-2577

Inverting resolution: Plotkin's least general generalization

MARIA DO CARMO NICOLETTI
MARIA CAROLINA MONARD

Nº 130

N O T A S D O I C M S C

São Carlos
mar./ 1993

Inverting Resolution: Plotkin's Least General Generalization

Maria do Carmo Nicoletti
Universidade Federal de São Carlos / ILTC
Departamento de Computação

Maria Carolina Monard [1]
Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

Abstract

Inductive Logic Programming — ILP — is a new trend in Machine Learning which attempt to overcome some of the limitations of classical concept learning by intensively using background knowledge and by allowing a richer concept description language — the language of logic programs.

The *inductive resolution methods*, one of the existing approaches to ILP are based on the fact that a correct hypothesis must allow a resolution proof of the examples. Inverse resolution is thus a nondeterministic process that generates all possible hypothesis of resolution proofs for the observations.

This work introduces the basic definitions used in ILP — taking into account the Prolog syntax — and discusses the concepts and ideas which support inverse resolution methods, such as least general generalization and relative least general generalization.

Fevereiro 1993

[1] Trabalho realizado com auxílio do CNPq.

Contents

1 Introduction

Inductive Logic Programming — ILP — is a new trend in Machine Learning which attempts to overcome some of the strong limitations of classical inductive concept-learning, such as:

- **background knowledge expressed in rather limited form,**
- **lack of relational descriptions.**

The research area of ILP is concerned with inducing logic programs from examples in the presence of background knowledge. The learning problem in ILP can be formalised as follows:

> Given some background knowledge B expressed as a set of predicates, some examples E and some negative examples N, find a logic formula H, such that:
>
> $$B \wedge H \models E$$
>
> and
>
> $$B \wedge H \not\models N$$

The *inductive resolution methods* (sneS7approach to ILP) are based on the fact that a correct hypothesis must allow a resolution proof of the examples. Inverse resolution is thus a nondeterministic process that generates all possible hypothesis of resolution proofs for the observations.

The Golem system [Muggleton 90] is based on Plotkin's notion of the relative least general generalization — rlgg — of a set of examples with respect to some background knowledge; by making certain syntatic restrictions, it can be shown that a unique, finite rlgg can be found, which is then simplified to generate a reasonable hyphotesis [Džeroski 92].

In this work it is presented the basic definitions used in Inductive Logic Programming (taking into account the Prolog syntax) and discussed the concepts and ideas which support inverse resolution methods, namely, least general generalization and relative least general generalization.

It also provides the necessary enhancements for a better understanding and analysis of systems based on the inverse resolution approach.

This work is organized as follows: in Section 2 the basic concepts of first order predicate calculus for logic programs are introduced; models of logic programs are presented in Section 3.

[2]Declarative prior knowledge often required in order to solve difficult learning problems.

Section 4 is concerned with unification, substitution and resolution. The unification algorithm is discussed in some detail. Section 5 deals with Plotkin's idea of least general generalization — lgg — (or anti-unification) for inverting resolution, as well as algorithms to compute the lgg of terms, literals and clauses. The absorption, identification and intra-construction operations for inverting resolution are discussed in the framework of propositional logic. Finally, the conclusions are presented in Section 6.

2 First Order Predicate Calculus

A *first order theory* τ consists of an alphabet, a first order language, a set of axioms and a set of inference rules. The first order language consists of the well-formed formulas of the theory. The axioms are designated subset of well-formed formulas. The axioms and rules of inference are used to derive the theorems of the theory.

Next will be presented the seven classes of symbols which define a first-order alphabet. Some notational conventions adopted for these classes, as well as some examples will be introduced.

Definition 2.1 *A first-order alphabet consists of seven classes of symbols:*

1. *variables*
   represented by an upper-case letter followed by a string of lower-case letters and/or digits, such as: *X*, *Y*, *Mother*.

2. *constants*
   start with a lower-case letter or digit, such as: *sea*, *blue*, *3*.

3. *n-ary functions*
   a n-ary function symbol is a lower-case letter followed by a string of lower-case letters and/or digits, together with n slots[3](or argument places) such as: *parent(a,b)*, *mother(X,Y)*, *f(Z)*.

4. *n-ary predicates*
   a n-ary predicate symbol is a lower-case letter followed by a string of lower-case letters and/or digits, together with n slots[3] (or argument places) such as[4]: *less(5,20)*, *mother(ann,Z)*, *f(Z)*.

5. *connectives*
   $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftarrow$,

6. *quantifiers*
   $\forall$ and $\exists$

[3]A n-ary function or predicate symbol will be noted as: $< name > / < arity >$.

[4]It is important to notice that from a syntactic point of view, there is no distinction between a function symbol and a predicate symbol.

7. ***punctuation symbols***
   (,) and ,.

Classes *5.* to *7.* are the same for every alphabet, while classes *1.* to *4.* vary from alphabet to alphabet. For any alphabet only the classes *2.* and *3.* may be empty.

Next it will be introduced the definition of the first order language given by an alphabet.

Definition 2.2 *Terms are defined recursively as follows:*

1. *a constant is a term.*
2. *a variable is a term.*
3. *if f is a functor of arity n (meaning an n-ary function symbol) and $t_1, \ldots t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.*
4. *all terms are generated by applying the above rules.*

Example 2.1 *The variable X and the constant 1 are both terms. If plus is considered a 2-ary function symbol, $plus(X, 1)$ is a term, and so are $plus(plus(X, 1), 1)$, $plus(X, plus(X, 1))$, $plus(plus(X, 1), plus(plus(X, 1)))$ and so on.*

Definition 2.3 *If p is an n-ary predicate symbol and $t_1, t, \ldots, t_n$ are terms, then $p(t_1, t, \ldots, t_n)$ is an atom or atomic formula.*

Definition 2.4 *A ground term is a term not containing variables. Similarly, a ground atom is an atom not containing variables.*

Example 2.2 *The variable X and the constant 1 are both terms. If plus is considered a 2-ary predicate symbol, then $plus(X, 1)$ is an atom.*

It is worth to note that:

- constants denote objects in the domain.
- variables are used to make abstraction of constants and terms.
- terms denote objects in the domain.
- function symbols enable the creation of new terms out of old.
- predicate symbols enable the creation of statements.

Before giving a formal definition of a formula, it is important to distinguish between *bound variables* and *free variables*.

Definition 2.5 *The scope of a quantifier occurring in a formula F is the formula to which the quantifier applies.*

Example 2.3 *The scope of $\forall X$ (or $\exists X$) in $\forall X F$ ($\exists X$) is F.*

Definition 2.6 *An occurrence of a variable in a formula is bound if and only if the occurrence is within the scope of a quantifier employing the variable, or is the occurrence in that quantifier. An occurrence of a variable in a formula is free if and only if this occurrence of the variable is not bound.*

Definition 2.7 *A variable is free in a formula if at least one occurrence of it is free in the formula. A variable is bound in a formula if at least one occurrence of it is bound.*

Definition 2.8 *A closed formula is a formula with no free occurrences of any variable.*

Example 2.4 *In the formula*

$$\exists X p(X,Y) \vee q(X)$$

the first two occurrences of X are bound, while the third occurrence is free, since the scope of $\exists X$ is $p(X,Y)$.

In the formula

$$\exists X (p(X,Y) \vee q(X))$$

all occurrences of X are bound, since the scope of $\exists X$ is $p(X,Y) \vee q(X)$.

In the formula

$$(\forall X) p(X,Y)$$

since both the occurrences of X are bound, the variable X is bound. However, the variable Y is free since the only occurrence of Y is free. For example, Y is both free and bound in the formula

$$(\forall X) p(X,Y) \wedge (\forall Y) q(Y)$$

This work is concerned with closed formulas. It follows now a formal definition of a formula.

Definition 2.9 *A well-formed formula — wff — is defined recursively as follows:*

1. *an atom is a formula.*
2. *if F and G are formulas, then so are $(\neg F)$, $(F \vee G)$, $(F \wedge G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$.*

3. *if F is a formula and X is a free variable in F, then $(\forall X\ F)$ and $(\exists X\ F)$ are formulas.*

4. *formulas are generated only by a finite number of applications of 1), 2) and 3).*

It will often be convenient to write the formula $(F \rightarrow G)$ as $(G \leftarrow F)$.

Definition 2.10 *The first order language λ given by an alphabet consists of the set of all wffs constructed from the symbols of the alphabet.*

Definition 2.11 *If F is a formula, then $\forall(F)$ denotes the universal closure of F, which is the closed formula obtained by adding a universal quantifier for every variable having a free occurrence in F. Similarly, $\exists(F)$ denotes the existential closure of F, which is obtained by adding an existential quantifier for every variable having a free occurrence in F.*

Example 2.5 *If F is $p(X,Y) \wedge q(X)$, then*

$$\forall(F) \text{ is } \quad \forall X \forall Y (p(X,Y) \wedge q(X))$$

while

$$\exists(F) \text{ is } \quad \exists X \exists Y (p(X,Y) \wedge q(X))$$

Definition 2.12 *A literal is an atom or the negation of an atom. A positive literal is just an atom. A negative literal is the negation of an atom.*

The literals L and $\neg L$ are said to be each others complements and form, in either order, a complementary pair.

Definition 2.13 *A clause is a formula of the form*

$$\forall X_1 \forall X \ldots \forall X_p (L_1 \vee L \vee \ldots L_q)$$

where each L_i is a literal and $X_1, X, \ldots, X_p$ are all the variables occurring in $L_1, L, \ldots, L_q$.

A clause can also be represented as a finite (possibly empty) set of its literals. Because clauses are so common in logic programming, it will be convenient to adopt a special clausal notation. So, the clause:

$$\forall X_1 \forall X \ldots \forall X_p (A_1 \vee A \vee \ldots \vee A_m \vee \neg B_1 \vee \neg B \ldots \vee \neg B_n)$$

where

$A_1, A, \ldots, A_m, B_1, B, \ldots, B_n$ are atoms and $X_1, X, \ldots, X_p$ are the variables occurring in the atoms, will be denoted by:

$$A_1, A, \ldots A_m \leftarrow B_1, B, \ldots, B_n$$

meaning

$$A_1 \vee A \vee \ldots \vee A_m \leftarrow B_1 \wedge B \wedge \ldots \wedge B_n$$

Thus, in the clausal notation, all variables are assumed to be universally quantified. The following special cases can happen:

1. $m > 1$ the conclusions are indefinite, i.e., there are more than one conclusion.
2. $m = 1, n \geq 0$ known as *definite clause*.
3. $m = 0, n = 0$ known as the *empty clause*, and represented by $\square$.

Definition 2.14 *A definite program clause is a clause which contains exactly one positive literal. It has the form:*

$$A \leftarrow B_1, \ldots, B_n$$

where A, B_1, ...,B_n are atoms. The positive literal A is called the head and the atoms $B_1, \ldots, B_n$ are collectively called the body of the program clause.

Only atoms are allowed in the body of definite program clauses. In Prolog, however, literals of the form *not B*, where B is an atom, are allowed, where *not* is interpreted under the *negation-as-failure* rule.

Definition 2.15 *A program clause is a clause of the form*

$$A \leftarrow L_1, \ldots, L_m$$

where A is an atom and each of $L_1, \ldots, L_m$ is of the form B of $\neg B$, where B is an atom.

Definition 2.16 *A unit clause is a clause of the form*

$$A \leftarrow$$

that is, a program clause (or a definite program clause) with an empty body.

In Prolog terminology such a clause is called a *fact* — and sometimes *unconditional definite clause* — and is denoted simply by

$$A.$$

Definition 2.17 *A set of clauses is called a clausal theory and represents the conjunction of its clauses.*

Definition 2.18 *Let E be a wff or term. Let vars(E) denote the set of variables in E. E is said to be ground if and only if $vars(E) = \emptyset$.*

Definition 2.19 *A goal clause is a clause of the form*

$$\leftarrow B_1, \ldots, B_n$$

that is, a clause which has no head. Each B_i $(i = 1, \ldots, n)$ is called a subgoal of the goal clause. It is the pure negation of $B_1, \ldots, B_n$

If $Y_1, \ldots, Y_r$ are the variables of the goal clause $\leftarrow B_1, \ldots, B_n$ then this clausal notation is a shorthand for

$$\forall Y_1 \ldots \forall Y_r(\neg B_1 \vee \ldots \vee \neg B_n)$$

or equivalently

$$\neg \exists Y_1 \ldots \exists Y_r(B_1 \wedge \ldots \wedge B_n)$$

Definition 2.20 *A finite set of program clauses is called a logic program.*

Consequently, a logic program does not contain any goal clause.

Definition 2.21 *In a logic program, the set of all program clauses with the same predicate p in the head is called the definition of p.*

Definition 2.22 *A Horn clause is a clause which is either a program clause or a goal clause. That means that a Horn clause contains at most one positive literal.*

Definition 2.23 *A normal program is a finite set of program clauses.*

3 Models of Logic Programs

Wffs have meaning only when an interpretation is given for the symbols. In the propositional logic, an interpretation is an assignment of truth values to atoms. In the first-order logic, since there are variables involved, to define an interpretation for a language λ two things must be specified:

- the domain,
- assignment to constants, functions symbols and predicate symbols occurring in the language. It should be assigned to each
  - *constant symbol*, an entity in the domain,
  - *function symbol*, a function in the domain,
  - *predicate symbol*, a corresponding relation in the domain.

What follows is a formal definition of an interpretation of a language in the first-order logic.

Definition 3.1 *An interpretation I of a first order language[5] λ consists of the following:*

1. a non-empty set D, called the domain of the interpretation over which the variables range,
2. the assignment to each
   - constant symbol in λ of an element in D,
   - n-ary function symbol in λ of a mapping from D_n to D,
   - n-ary predicate symbol in λ of a mapping from D_n to {true, false}.

Each interpretation thus specifies a meaning for each symbol in the language. This assignments define the *semantics* of the first-order language.

For a given interpretation, a wff without free variables represents a proposition which is true or false, whereas a wff with free variables stands for a relation on the domain of interpretation which may be satisfied (true) for some values in the domain of the free variables and not satisfied (false) for the others.

Definition 3.2 *Let I be an interpretation of a first order language λ. A variable assignment (wrt I) is an assignment to each variable in λ of an element in the domain of I.*

[5]Sometimes a definition of an interpretation can be given related to a formula F, instead to a first-order language λ — the definition is the same, except that λ should be replaced by F.

Definition 3.3 *Let I be an interpretation of a first order language λ, with domain D. Let A be a variable assignment (wrt I). The term assignment (wrt I and A) of the terms in λ is defined as follows:*

- each variable is given its assignment according to A,
- each constant is given its assignment according to I,
- if $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ and f' is the assignment of f, then $f'(t'_1, \ldots, t'_n) \in D$ is the term assignment of $f(t_1, \ldots, t_n)$.

Definition 3.4 *Let I be an an interpretation of a first order language λ with domain D. Let A be a variable assignment. Then a formula in λ can be given a truth value true or false, (wrt I and A) as follows:*

1. if the formula is an atom $p(t_1, \ldots, t_n)$, then the truth value is obtained by calculating the value of $p'(t'_1, \ldots, t'_n)$, where p' is the mapping assigned to p by I and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ wrt I and A.
2. if the formula has the form $\neg F$, $F \vee G$, $F \wedge G$, $F \rightarrow G$ or $F \leftrightarrow G$, then the truth value of the formula is given by the table:

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

3. if the formula has the form $\exists X F$, the truth value of the formula is true if there exists $d \in D$ such that F has truth value true wrt I and $A(X/d)$, where $A(X/d)$ is A except that X is assigned to d; otherwise its truth value is false.
4. if the formula has the form $\forall X F$, the truth value of the formula is true if, for all $d \in D$, F has truth value true wrt I and $A(X/d)$, where $A(X/d)$ is A except that X is assigned to d; otherwise its truth value is false.

It should be noted, however, that the truth value of a closed formula does not depend on the variable assignment; consequently, the variable assignment should not be taken into consideration when evaluating the truth value of a closed formula — that is to say that the truth value of a closed formula is always considered only wrt an interpretation.

Example 3.1 *Consider a language λ that has as:*

function symbols:	$f/1$	$g/1$	
predicate symbols:	$p/1$	$q/2$	$r/1$
constant symbols:	a	b	c
variables symbols:	X	Y	Z

Let I be the following interpretation for λ:

- **Domain:**

$$\{1,2\}$$

- **Assignment for constants:**

a	b	c
1	2	1

- Assignment for function symbols:

$f(1)$	$f(2)$	$g(1)$	$g(2)$
2	1	1	2

- Assignment for predicate symbols:

$p(1)$	$p(2)$	$q(1,1)$	$q(1,2)$	$q(2,1)$	$q(2,2)$	$r(1)$	$r(2)$
false	*true*	*true*	*true*	*false*	*true*	*true*	*false*

Obs:- The variable assignment will not be taken into consideration since only closed formulas will be considered.

Consider the formula:

$$F : (\forall X)(p(X) \rightarrow q(f(X), a))$$

If $X = 1$

$p(X)$	$\rightarrow$	$q(f(X), a)$	=
$p(1)$	$\rightarrow$	$q(f(1), a)$	=
$p(1)$	$\rightarrow$	$q(2,1)$	=
false	$\rightarrow$	false	= true

If $X = 2$

$p(X)$	$\rightarrow$	$q(f(X), a)$	=
$p(2)$	$\rightarrow$	$q(f(2), a)$	=
$p(2)$	$\rightarrow$	$q(1,1)$	=
false	$\rightarrow$	false	= true

Since $p(X) \rightarrow q(f(X), a)$ is true for all elements X in the domain D, the formula $(\forall X)(p(X) \rightarrow q(f(X), a))$ is true under the interpretation I. So I is a *model* for F.

Definition 3.5 *Let I be an interpretation of a first order language λ and let F be a closed formula of λ. Then I is a model for F if the truth value of F wrt I is true.*

The axioms of a first order theory are a designated subset of closed formulas in the language of the theory.

Definition 3.6 *Let τ be a first order theory (see Section 2) and let λ be the language of τ. A model for τ is an interpretation for λ which is a model for each axiom of τ.*

Definition 3.7 *Let S be a set of closed formulas of a first order language λ and let I be an interpretation of λ. I is a model for S if I is a model for each formula of S.*

Note that if $S = \{F_1, \ldots, F_n\}$ is a finite set of closed formulas, then I is a model for S iff I is a model for $F_1 \wedge \ldots \wedge F_n$.

Definition 3.8 *Let S be a set of closed formulas of a first order language λ.*

1. *S is satisfiable in λ if λ has an interpretation which is a model for S.*
2. *S is valid if every interpretation of λ is a model for S.*
3. *S is unsatisfiable if it has no models (there is no interpretation of λ which is a model for S).*

Definition 3.9 *Let S be a set of closed formulas and F be a closed formula of a first order language λ. F is said to be a logical consequence of S if, for every interpretation I of λ, I is a model for S implies that I is a model for F.*

If $S = \{F_1, \ldots, F_n\}$ is a finite set of closed formulas, then F is a logical consequence of S iff

$$F_1 \vee \ldots \vee F_n \rightarrow F \text{ is valid}$$

Proposition 3.1 *Let S be a set of closed formulas and F be a closed formula of a first order language λ. Then*

$$F \text{ is a logical consequence of } S \text{ iff } S \bigcup \{\neg F\} \text{ is unsatisfiable}$$

Proof: Suppose that F is a logical consequence of S. Let I be an interpretation of λ and suppose I is a model for S. Then I is also a model for F. Hence I is not a model for $S \bigcup \{\neg F\}$. Thus $S \bigcup \{\neg F\}$ is unsatisfiable.

Conversely, suppose that $S \bigcup \{\neg F\}$ is unsatisfiable. Let I be any interpretation of λ. Suppose I is a model for S. Since $S \bigcup \{\neg F\}$ is unsatisfiable, I cannot be a model for $\neg F$. Thus I is a model for F and so F is a logical consequence of S.

Example 3.2 *Let $S = \{p(a), \forall X(p(X) \rightarrow q(X))\}$ and F be $q(a)$. Let I be any model for S. Thus $p(a)$ is true wrt I. It follows from this and the truth of $\forall X(p(X) \rightarrow q(X))$ that $q(a)$ must be true wrt I.*

Applying these definitions to logic programs, when a goal G is given to the system, with program P loaded, what has been asked to the system is to show that the set of clauses $P \bigcup \{G\}$ is unsatisfiable. In fact, if G is the goal $\leftarrow B_1, \ldots, B_n$, with variables $Y_1, \ldots, Y_r$, then the proposition above states that showing $P \bigcup \{G\}$ unsatisfiable is exacly the same as showing that

$$\exists Y_1 \ldots \exists Y_r (B_1 \vee \ldots \vee B_n)$$

is a logical consequence of P.

Thus the basic problem is that of determining the unsatisfiability of $P \bigcup \{G\}$, where P is a program and G is a goal. According to the definition, this implies showing every interpretation of $P \bigcup \{G\}$ is not a model [Chang 73,Lloyd 84]

That is, logical implication can be reduced to testing unsatisfiability of a formula in conjuctive form. In propositional logic it is possible to just enumerate all truth assigments to test unsatisfiability. On the other side, for predicate logic it will be necessary to enumerate all structures to use that approach. However, there are infinite structures to make this approach feasible. Fortunately there is a way to decide unsatisfiability by examing only some structures.

For each universal formula there is a special domain — based on Herbrand universe — and a special constant mapping — based on Herbrand interpretation — such that only structures using that domain and mapping have to be considered.

Herbrand interpretation is used in ILP in order to generate the *h-easy model* with respect to a given logic program P. The notion of *h-easiness* can be seen in more detail in [Muggleton 90].

4 Substitution and Unification

Frequently, when dealing with logical concepts and proving theorems involving quantified formulas, is necessary to *match* certain subexpressions. Finding substitutions of terms for variables to make expressions identical is an extremely important process in AI and is called *unification*. The unification process is based on *substitution*.

The terms of an expression can be variable symbols, constant symbols or functional expressions, the latter consisting of function symbols and terms — as seen in Section 2.

A substitution instance of an expression is obtained by substituting terms for variables in that expression. This process can be formally defined as follows:

Definition 4.1 *A substitution is a finite set of the form*

$$\theta = \{t_1/v_1, \ldots, t_n/v_n\}$$

where every v_i is a variable, every t_i is a term different from v_i and no two elements in the set have the same variables after the stroke symbol.

When $t_1, \ldots, t_n$ are ground terms, the substitution is called a ground substitution.

The substitution that consists of no elements is called empty substitution and is denoted by ε.

Generally substitutions are represented by greek letters.

Definition 4.2 *Let $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ be a substitution and E an expression. Then $E\theta$ is an expression obtained from E by replacing simultaneously each occurrence of the variable $v_i (1 \leq i \leq n)$ in E by the term t_i. $E\theta$ is called an instance of E.*

Example 4.1 *Let $\theta = \{a/X, f(g(c))/Y, g(W)/Z\}$ and $E = p(X, Y, Z)$. Then*

$$E\theta = p(a, f(g(c)), g(W))$$

Definition 4.3 *Let $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ be a substitution. The set $\{v_1, \ldots, v_n\}$ is called the domain of θ, or dom(θ) and the set $\{t_1, \ldots, v_n\}$ is called the range of θ or rng(θ).*

It should be observed that a substitution uniquely maps terms to variables.

Let E be a wff or a term and $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ be a substitution. The instantiation of E by θ, written $E\theta$, is formed by replacing every occurrence of v_i in E by t_i. It is important to remind that no variable can be replaced by a term containing that same variable.

Every sub-term within a given term or literal W can be uniquely referenced by its *place* within W — see definition 4.5.

Example 4.2 *Next table shows four instances of the literal $p(X, f(Y), b)$ obtainned using four different substitutions.*

substitution	*substitution instance*
$\{Z/X, W/Y\}$	$p(Z, f(W), b)$
$\{a/Y\}$	$p(X, f(a), b)$
$\{g(Z)/X, a/Y\}$	$p(g(Z), f(a), b)\}$
$\{c/X, a/Y\}$	$p(c, f(a), b)$

The last of the four substitution instances is an example of a *ground instance*, since none of the terms in the literal contains variables.

Definition 4.4 *Let*

$$\theta_1 = \{t_1/x_1, \ldots, t_n/x_n\}$$

and

$$\theta = \{u_1/y_1, \ldots, u_m/y_m\}$$

Then the composition *of θ_1 and θ is the substitution denoted by $\theta_1\theta$, that is obtained from the set*

$$\{t_1\theta/x_1, \ldots, t_n\theta/x_n, u_1/y_1, \ldots, u_m/y_m\}$$

by deleting any element $t_j\theta/x_j$ for which $t_j\theta = x_j$ and any element u_i/y_i such that y_i is among $\{x_1, x, \ldots, x_n\}$.

Example 4.3 *Composing substitutions*

$$\{g(X,Y)/Z\}\{a/X, b/Y, c/W, d/Z\} = \{g(a,b)/Z, a/X, b/Y, c/W\}$$

Example 4.4 *Let*

$$\theta_1 = \{t_1/x_1, t/x\} = \{f(Y)/X, Z/Y\}$$

and

$$\theta = \{u_1/y_1, u/y, u_3/y_3\} = \{a/X, b/Y, c/Z\}$$

then

$$\{t_1\theta/x_1, t\theta/x, u_1/y_1, u/y, u_3/y_3\} = \{f(b)/X, Y/Y, a/X, b/Y, Y/Z\}$$

However, since $t\theta/x$ and $t\theta = x$, Y/Y should be deleted from the set. In addition, since y_1 and y are among $\{x_1, x, x_3\}$, u_1/y_1 and u/y, that is, a/X and b/Y, should be deleted. Thus it can be written that:

$$\theta_1\theta = \{f(b)/X, Y/Z\}$$

It can be shown that applying θ_1 and θ successively to an expression E is the same as applying $\theta_1\theta$ to E; that is

$$(E\theta_1)\theta = E(\theta_1\theta)$$

It can also be shown that the composition of substitutions is associative; that is

$$(\theta_1\theta)\theta_3 = \theta_1(\theta\theta_3)$$

Definition 4.5 *Places within terms or literals are denoted by n-uples of natural numbers and defined recursively as follows:*

- *the term at place* $< a_1 >$ *within* $f(t_1, \ldots, t_n)$ *is* t_{a1},
- *the term at place* $< a_1, \ldots, a_m >$ *within* $f(t_1, \ldots, t_n)$ *is the term at place* $< a, \ldots, a_m >$ *in* t_{a_1}.

Example 4.5 *Let* L *be the literal*

$$f(X, a, g(Y, h(c,d), k(m,n,o)), l(u,v), Z)$$

Then

the term at place $< 3 >$ *within* L *is* $g(Y, h(c,d), k(m,n,o))$.

the term at place $< 3,3,2 >$ *within* L
is the term at place $< 3,2 >$ *within* $g(Y, h(c,d), k(m,n,o))$,
which is the term at place $< 2 >$ *within* $k(m,n,o)$
which is n.

The definition of place can easily be extended to cover places within clauses by assuming a fixed ordering on literals within a clause.

Definition 4.6 *Let* t *be a term found at place* p *in literal* L, *where* L *is a literal within clause* C. *The place of* t *in* C *is denoted by the pair* $< L, p >$.

Example 4.6 *Let the clause*

$$C = \{p(X,Y) \leftarrow q(a, h(X,b), Y), m(a, h(l))\}$$

where $h(X,b)$ *is the term at place* $< 2 >$ *in the literal* $q(a, h(X,b), Y)$. *So the place of* $h(X,b)$ *in* C *is the pair*

$$< q(a, h(X,b), Y), 2 >$$

Definition 4.7 *Let* E *be a clause or a term and*

$$\theta = \{t_1/v_1, \ldots, t_n/v_n\}$$

be a substitution. The corresponding inverse substitution θ^{-1} *is*

$$\{v_1/< t_1,\{p_{1,1},\ldots,p_{1,m_1}\}>,\ \ldots,\ v_n/< t_n,\{p_{n,1},\ldots,p_{n,m_n}\}>\}$$

An inverse substitution is applied by replacing all t_i at places $p_{i,1},\ldots,p_{i,m_i}$ within E by v_i. Clearly $E\theta\theta^{-1} = E$. It is important to note that an inverse substitution is not strictly a substitution but rather a rewrite. Whereas the substitution θ maps terms to variables within t, the inverse substitution θ^{-1} maps variables in t to terms in $t\theta$.

Example 4.7 *Let L be the literal $f(X, g(X, Y))$. Let the substitution*

$$\theta = \{t_1/v_1, t/v\} = \{a/X,\ b/Y\}$$

Then

$$L\theta = f(a, g(a, b))$$

The inverse substitution is

$$\theta^{-1} = \{X/< a,\{< 1 >,< 2,1 >\}>, Y/< b,\{< 2,2 >\}>\}$$

$< 1 >$ and $< 2,1 >$ are the places within L at which variable X is found and $< 2,2 >$ is the place within L at which variable Y is found.

In the resolution proof procedure, it is common to have to unify (match) two or more expressions. Therefore, next is considered the unification of expressions.

Definition 4.8 *A substitution θ is called a unifier of a set $\{E_1, \ldots, E_k\}$ iff*

$$E_1\theta = E\theta = \ldots = E_k\theta$$

The set $\{E_1, \ldots, E_k\}$ is said to be unifiable if there is a unifier for it.

Example 4.8 *$\theta = \{a/X, b/Y\}$ unifies*

$$\{p(X, f(Y), b), p(X, f(b), b)\}$$

to yield

$$\{p(a, f(b), b)\}$$

Example 4.9 *The set $\{p(a, Y), p(X, f(b))\}$ is unifiable since the substitution*

$$\theta = \{a/X, f(b)/Y\}$$

is a unifier for the set.

Definition 4.9 *A unifier δ for a set $\{E_1, \ldots, E_k\}$ of expressions is a most general unifier — mgu — if and only if for each unifier θ for the set there is a substitution γ such that $\theta = \delta\gamma$. Two (or more) expressions are unifiable if they have a mgu.*

Example 4.10 *$\theta = \{a/X, b/Y\}$ is a unifier of the set*

$$\{p(X, f(Y), b), p(X, f(b), b)\}$$

but in some sense it is not the simplest unifier. There was no need to substitute X by a in order to achieve unification.

Definition 4.10 *Let C_1 and C be two wff's and θ the substitution $\{t_1/v_1, \ldots, t_n/v_n\}$, in which for $i \neq j$, $t_i \neq t_j$.*

$C_1\theta$ and $C\theta$ are said to be standardised apart whenever there is no variable which occurs in both $C_1\theta$ and $C\theta$.

4.1 Recursive Procedure UNIFY

There are many algorithms that can be used to unify a finite set of unifiable expressions and which report failure when the set cannot be unified. The following recursive procedure, found in [Nilsson 80], is useful for establishing a general idea of how to unify a set of two list-structured expressions[6].

```
UNIFY(E1,E2)
1   if either E1 or E2 is an atom (that is a predicate symbol, a function symbol, a constant symbol, a negation
    symbol or a variable), interchange the arguments E1 and E2 (if necessary), so that E1 is an atom, and do:
    begin
3     if E1 and E2 are identical, return NIL
4     if E1 is a variable, do:
5       begin
6         if E1 occurs in E2, return FAIL
7         return {E1/E2}
8       end
9     if E2 is a variable, return {E2/E1}
10    return FAIL
11  end
1   F1 ← the first element of E1, T1 ← the rest of E1
13  F2 ← the first element of E2, T2 ← the rest of E2
```

[6]The literal $p(X, f(a, Y))$ is written as $(pX(faY))$ in this list-structured form.

14 $Z_1 \leftarrow$ **UNIFY**(F_1, F_2)

15 **if** $Z_1 =$ **FAIL, return FAIL**

16 $G_1 \leftarrow$ **result of applying** Z_1 **to** T_1

17 $G_2 \leftarrow$ **result of applying** Z_1 **to** T_2

18 $Z_2 \leftarrow$ **UNIFY**(G_1, G_2)

19 **if** $Z_2 =$ *FAIL* **return FAIL**

0 **return the composition of** Z_1 **and** Z_2

It can be proven that **UNIFY** finds a *most general unifier* of a set of unifiable expresions or reports failure when the expressions are not unifiable. It follows a Prolog implementation of the algorithm **UNIFY** and some examples of its execution.

4.2 Prolog Implementation of algorithm UNIFY

```
% unifying two variables
unify(X,Y):-
   var(X),
   var(Y),
   X = Y.

% unifying a variable with a non-variable
% the unification only can be realized  if the variable does not appear
% in the non-variable
unify(X,Y):-
   var(X),
   nonvar(Y),
   ((atomic(Y),!) ; not_appear(X,Y)),
   X = Y.

% the same as before
unify(X,Y):-
   var(Y),
   nonvar(X),
   ((atomic(X),!) ; not_appear(Y,X)),
   X = Y.

% atomic arguments
unify(X,Y):-
   nonvar(X),
   nonvar(Y),
   atomic(X),
   atomic(Y), !,
   X = Y.

% structure arguments
unify(X,Y):-
   nonvar(X),
   nonvar(Y),
   X =..[F,X1|Xs],
   Y =..[F,Y1|Ys],
   unify_args([X1|Xs],[Y1|Ys]).

unify_args([],[]).
unify_args([X|Xs],[Y|Ys]):-
   unify(X,Y),
   unify_args(Xs,Ys).

not_appear(X,Y):-
```

```
    Y =..L,
    not_appear1(X,L).

not_appear1(X,[]).

not_appear1(X,[L|Ls]):-
    atomic(L),!,
    not_appear1(X,Ls).

not_appear1(X,[L|Ls]):-
    var(L),!,
    not (X == L),
    not_appear1(X,Ls).

not_appear1(X,[L|Ls]):-
    not_appear(X,L),
    not_appear1(X,LS),!.
```

Examples of execution:

```
?- unify(f(X,b,Y),f(a,b,c)).
X = a
Y = c ->;
no

?- unify(f(g(h(i(c),X,Y))),f(g(h(i(Z),W,a)))).
X = _00FC
Y = a
Z = c
W = _00FC ->;
no

?- unify(X,f(X)).
no
```

4.3 Resolution

Resolution is an important rule of inference that can be applied to *clauses*. As mentioned before, a clause is defined as a wff consisting of a disjunction of literals. The resolution process, when applicable, is applied to a pair of *parent* clauses to produce a derived clause called the *resolvent* of the two clauses.

In order to apply resolution to clauses containing variables, it is necessary to find a substitution that can be applied to the parent clauses so that they contain complementary literals.

Let the prospective parent clauses be given by the following sets of literals $\{L_i\}$ and $\{M_i\}$ and let us assume that the variables occurring in these two clauses have been standardized apart[7].

[7] To standardize variables apart means to rename variables symbols such that the same variable symbol does not appear in more than one clause — definition 4.10. In logic programming notation, this means that if there are two clauses

$p(X) \leftarrow \ldots$	they should be transformed	$p(X) \leftarrow \ldots$
$q(X) \leftarrow \ldots$	for example into	$q(Y) \leftarrow \ldots$

Suppose that $\{l_i\}$ is a subset of $\{L_i\}$ and that $\{m_i\}$ is a subset of $\{M_i\}$ such that a most general unifier θ exists for the union of the sets $\{l_i\}$ and $\{\neg m_i\}$. It can be said that the two clauses $\{L_i\}$ and $\{M_i\}$ *resolve* and that the new clause

$$\{\{L_i\} - \{l_i\}\}\theta \cup \{\{M_i\} - \{m_i\}\}\theta$$

is a *resolvent* of the two clauses.

If two clauses resolve, they may have more than one resolvent because there may be more than one way in which to choose $\{l_i\}$ and $\{m_i\}$. In any case, they can have at most a finite number of resolvents.

Figure 1 shows an example of the aplication of the resolution rule to a pair of clauses from the propositional calculus, in order to obtain a resolvent[8].

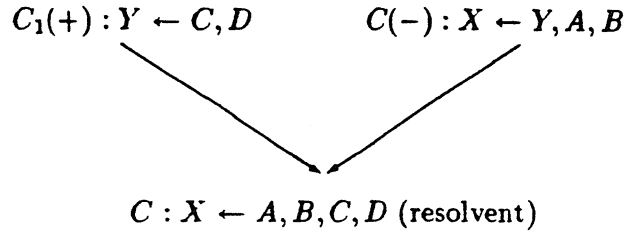

Figure 1: Resolution (Input: $C_1(+)$ & $C(-)$)

4.4 Resolution Proofs

Resolution is used for deriving the consequences of a logical theory. Given a theory T, C is a consequence of T (*or* $T \rightarrow C$) if and only if $T \wedge \neg C$ can be shown to be false.

Resolution-based systems are designed to produce proofs by contradiction or *refutation*. A resolution proof consists of a series of resolution inference steps which generate the empty or false clause, given $T \wedge C$ as input. Such a proof is often represented graphically as a binary tree [Chang 73].

5 Inverting Resolution - Plotkin's Least General Generalization

Although it is apparent to many researchers in Machine Learning that there is a strong relationship between deductive theorem-proving mechanisms and inductive inference,

[8] $C_1(+)$ means that Y is a positive literal (or atom) within this clause. $C_2(-)$ means that Y is a negative literal within this clause.

this idea has rarely been investigated to any greater depth in order to notice that the ideas of *logical subsumption* or *logical implication* are central to both. One exception to this is Plotkin [Plotkin 71], who investigated the idea that

> *just as unification was fundamental to deduction, so might a converse be of use in induction* [Muggleton 88].

From this idea Plotkin went on to develop the concept of *least general generalization* - lgg - or *anti-unification* of literals and clauses.

Resolution provides an efficient mean of deriving a solution to a problem, giving a set of axioms which define the task environment. Whereas resolution takes two terms and resolves them into a *most general unifier* — *mgu* — the anti-unification finds the *least general generalization* — *lgg* — of two terms.

Inverse resolution is an approach which operates on theories rather than clauses. The name of this approach reflects its basic idea, which is to invert the resolution rule of inference. Next some concepts which support the process of inverting resolution are introduced.

Definition 5.1 *A clause C_1 is more general than (θ-subsumes) a clause C iff there is a substitution θ such that $C_1\theta \subseteq C$.*

$C_1\theta$-subsumes C is noted by $C_1 \preceq C$.

In this context, clauses are to be regarded as sets of literals related in a disjunctive way. For example, the clause

$$f(X,Y) \leftarrow g(X) \wedge h(X,Y)$$

corresponds to the set

$$\{f(X,Y), \neg g(X), \neg h(X,Y)\}$$

The θ-subsumption can be thought of as a partial ordering of generality over clauses in the absence of background knowledge.

Example 5.1 *Let the clauses*

$$C_1 = \{father(X,Y) \leftarrow parent(X,Y) \wedge male(X)\}$$

$$C = \{father(a,b) \leftarrow parent(a,b) \wedge parent(a,c) \wedge male(a) \wedge female(c)\}$$

respectively. $C_1\theta$-subsumes C, with $\theta = \{a/X, b/Y\}$

Example 5.2 *Let the clauses*

$$C_1 = p(X) \quad \textit{and} \quad C = p(a) \vee q(a)$$

If $\theta = \{a/X\}$ *then* C_1 *subsumes* C *since* $C_1\theta = p(a) \subseteq C$

The following properties hold for *θ-subsumption*:

1. if C_1 is more general than C then C_1 logically entails C.
2. the relation $\preceq$ induces a lattice on the set of all clauses. This means that for any two clauses there is a least upper bound and a greatest lower bound, and both are unique up to equivalence under *θ-subsumption*.

Definition 5.2 *The least general generalization — lgg — C of two clauses* C_1 *and* C *is the greatest lower bound whithin the clause lattice induced by the relation* $\preceq$.

A least general generalization is a generalization which is less general than any other such generalization. This concept is important for concept-learning as it forms the basis of cautious generalization algorithms. Cautious generalization assumes that if C_1 and C are true, it is very likely that $lgg(C_1, C)$ will also be true.

Example 5.3 $lgg(p(g(a), a), p(g(b), b)) = p(g(X), X)$

It should be noted that under the substitution $\{X/a\}$

$$p(g(X), X) \quad \textit{subsumes} \quad p(g(a), a)$$

and that under the substitution $\{X/b\}$

$$p(g(X), X) \quad \textit{subsumes} \quad p(g(b), b)$$

Subsumption served as a very useful tool in several learning systems. However, subsumption alone has its limitations. In particular, it is unable to take advantage of background information which may assist generalization. Consider next the example showed in [Buntine 86].

Example 5.4 *Suppose there are given two instances of a concept cuddly_pet,*

$$cuddly\_pet(X) \leftarrow fluffy(X) \wedge dog(X)$$

$$cuddly\_pet(X) \leftarrow fluffy(X) \wedge cat(X)$$

which have as lgg

$$cuddly\_pet(X) \leftarrow fluffy(X)$$

Suppose it is known that:

$pet(X) \leftarrow dog(X)$

$pet(X) \leftarrow cat(X)$

In the light of the background knowledge, the lgg is an over-generalization and potentially dangerous since any fluffy object could be considered a cuddly pet. A more appropriate one would be:

$$cuddly\_pet(X) \leftarrow fluffy(X) \wedge pet(X)$$

5.1 The LGG of Terms, Literals and Clauses

Next, three algorithms which compute the lgg of terms, lgg of literals and lgg of clauses respectively are presented [De Raedt 92].

Algorithm 5.1 *LGG of Terms*

procedure lgg(t_1 :term,t_2 :term)
  if $t_1 = t_2$
  then t_1
  else if $t_1 = f(u_1, \ldots, u_k)$ and $t_2 = f(v_1, \ldots, v_k)$
    then $f(lgg(u_1, v_1), \ldots, lgg(u_k, v_k))$
    else variable V
    endif
  endif
endproc

Example 5.5 *LGG of Terms*

T_1	T_2	$lgg(T_1, T_2)$
$[1,2,3]$	$[1,4,5]$	$[1,X,Y]$
$[a,b,c,d]$	$[a,c,d]$	$[a,X,Y\|Z]$
$tree(nil,a,nil)$	$tree(tree(nil,a,nil),a,nil))$	$tree(X,a,nil)$
$f(a,b,c,a)$	$f(x,y,x,x)$	$f(E,F,G,E)$

Algorithm 5.2 *LGG of Literals*

procedure lgg(L_1 :literal,L_2 :literal)
  if $L_1 = signp(t_1, \ldots, t_n)$ and $L_2 = signp(u_1, \ldots, u_n)$
  then $signp(lgg(t_1, u_1), \ldots, lgg(t_n, u_n))$
  else undefined
  endif
endproc

Example 5.6 *LGG of Literals*

L_1	L_2	$lgg(L_1, L_2)$
$f(a,b)$	$f(a,c)$	$f(a,X)$
$f(a,b)$	$\neg f(a,c)$	$undefined$
$\neg f(a,b)$	$\neg f(a,c)$	$\neg f(a,X)$
$append([1,2],[3,4],[1,2,3,4])$	$append([a],[b],[a,b])$	$append([X\|T],[Y\|U],[X,V\|W])$

Algorithm 5.3 *LGG of Clauses*

procedure lgg(C_1 :clause,C_2 :clause)
  if $C_1 = \{L_1,\ldots,L_n\}$ and $C_2 = \{K_1,\ldots,K_m\}$
  then $\{U_i \mid U_i = lgg(L_j, K_k)$ for $1 \leq j \leq n$ and $1 \leq k \leq m$ and $lgg(L_j, K_k) \neq undefined\}$
  endif
endproc

Example 5.7 *LGG of Clauses*

Let the clauses C_1 and C be

$$C_1 = father(tom,an) \leftarrow parent(tom,an) \wedge male(tom) \wedge female(an)$$

$$C = father(jef,paul) \leftarrow parent(jef,paul) \wedge male(jef) \wedge male(paul)$$

then

$$lgg(C_1,C) = father(X,Y) \leftarrow parent(X,Y) \wedge male(X) \wedge male(Z)$$

5.2 Inverting Resolution in Propositional Logic

In this section the discussion will be limited to propositional Horn clauses. In future works this analysis will be extended to deal with first-order representations.

Let C_1 and C be the two following clauses, where α and β stand for conjuctions of literals.

$$C_1 = (h_1 \leftarrow \alpha h)$$

$$C = (h \leftarrow \beta)$$

The resolvent or resolved product of C_1 and C can be written as

$$C = C_1.C = (h_1 \leftarrow \alpha\beta)$$

The resolved quotient can be defined as follows

$$C_1 = C/C$$

Alternatively, C_1 is called the *identificant* of C and C. Similarly

$$C = C/C_1$$

Again, C is called the *absorbant* of C and C_1.

It is now straightforward to define the absorption and identification operators.

Definition 5.3 *Given a propositional Horn clause program $P \supseteq \{C,C_1\}$, the absorption operator — Abs — transform P to*

$$P' = (P - \{C\}) \cup \{C/C_1\}$$

Figure 2 shows an example of the absorption operator applied to two clauses of the propositional calculus.

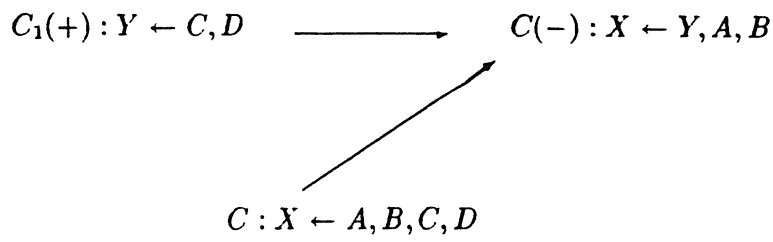

Figure 2: Absorption (Input: $C_1(+)$ & C)

Definition 5.4 *Given a propositional Horn clause program $P \supseteq \{C,C\}$, the identification operator — Ident — transform P to*

$$P' = (P - \{C\}) \cup \{C/C\}$$

Figure 3 shows an example of the identification operator applied to two clauses of the propositional calculus.

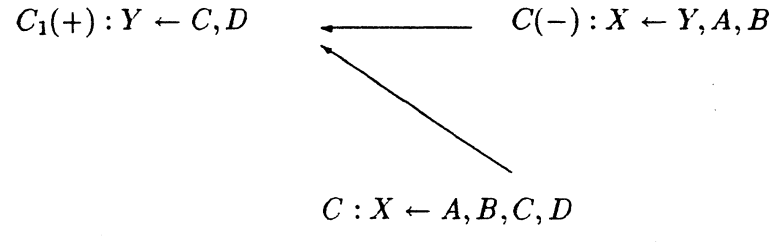

Figure 3: Identification (Input: $C(-)$ & C)

For the definition of the operator *intra-construction*, it will be considered

$A = (h_3 \leftarrow \gamma h_4)$,

$BB = \{B_1, \ldots, B_n\} = \{(h_4 \leftarrow \delta_1), \ldots, (h_4 \leftarrow \delta_n)\}$,

$CC = \{C_1, \ldots, C_n\} = \{(A.B_1), \ldots, (A.B_n)\} = \{(h_3 \leftarrow \gamma\delta_1), \ldots, (h_3 \leftarrow \gamma\delta_n)\}$

Definition 5.5 *Given a propositional Horn clause program $P \supseteq \{CC\}$, the intra-construction operator — Intra — transform P to*

$$P' = (P - \{CC\}) \cup \{A\} \cup BB$$

Figure 4 shows an example of the intra-construction operator applied to two clauses of the propositional calculus.

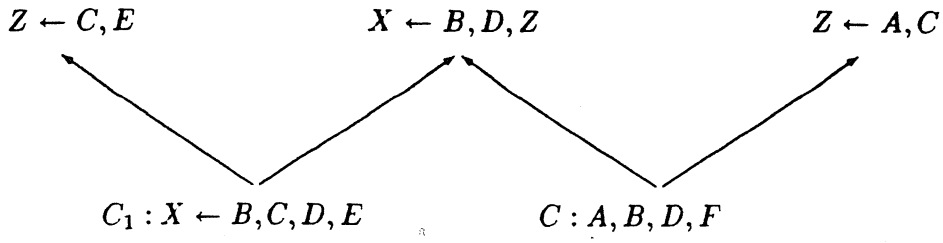

Figure 4: Intraconstruction (Input: C_1 & C)

The inverse of the three operators can be uniquely defined as follows.

Definition 5.6 *Given a propositional Horn clause program $P \supseteq \{C_1, C\}$, the inverse absorption operator — Abs^{-1} — transform P to*

$$P' = (P - \{C\}) \cup \{C_1.C\}$$

Definition 5.7 *Given a propositional Horn clause program $P \supseteq \{C_1, C\}$, the inverse identification operator — $Ident^{-1}$ — transform P to*

$$P' = (P - \{C_1\}) \cup \{C_1.C\}$$

Definition 5.8 *Given a propositional Horn clause program $P \supseteq (\{A\} \cup BB)\}$, the inverse intra-construction operator — $Intra^{-1}$ — transform P to*

$$P' = (P - (\{A\} \cup CC)) \cup BB$$

As a special case of Plotkin's *least general generalization* of clauses, γ is said to be the lgg of the bodies of clauses within CC ($bodies(CC)$) if and only if γ is the common intersection of propositional symbols of $bodies(CC)$.

Given γ and a new predicate symbol h_3, it is straightforward to construct A and BB. It is only through reversal of multiple resolution steps that the introduction of new predicate symbols becomes possible.

It is important to note that intra-construction automatically creates a new term in its attempt to simplify descriptions.

The lgg operations as showed before are purely syntactical as they do not take into account any background knowledge. Taking into account background knowledge would give the system a notion of semantical generalization.

Buntine [Buntine 86] has produced a theory for induction of horn-clauses called *generalized subsumption*. In this framework, all subsumption is done relative to a logic program. In this case, the logic program is the set of clauses which represent the learner's background knowledge.

The *relative least general generalization* — *rlgg* — is a lgg constructed with respect to the given background knowledge B.

Definition 5.9 *The rlgg of two examples e_1 and ϵ (ground facts) relative to the theory T (the background knowledge) is the least general clause C under the θ-subsumption lattice such that*

$$T \wedge \models e_1 \wedge e$$

A method to construct rlgg's can be derived as follows:

$$\begin{array}{rcl} \hline T \wedge C & \models & e_1 \\ C & \models & T \rightarrow e_1 \\ & \models & C \rightarrow (T \rightarrow e_1) \\ & \models & C \rightarrow (\neg T \vee e_1) \\ & \models & C \rightarrow (\neg(a_1 \wedge \ldots \wedge a_n) \vee e_1) \\ & \models & C \rightarrow (\neg a_1 \vee \ldots \vee \neg a_n \vee e_1) \\ \hline \end{array}$$

Doing the same for ϵ:

$$\models C \rightarrow (\neg a_1 \vee \ldots \vee \neg a_n \vee e)$$

If

$$C_1 = (\neg a_1 \vee \ldots \vee \neg a_n \vee e_1) \quad \text{and} \quad C = (\neg a_1 \vee \ldots \vee \neg a_n \vee e)$$

respectively, it follows that

$$\models C \rightarrow lgg(C_1.C)$$

In fact by definition $C = lgg(C_1, C)$. This constitutes the operational definition of rlgg.

Example 5.8 *Suppose the theory consists of the following facts [De Raedt 92]:*

$parent(paul, an) \leftarrow$

$parent(tom, jef) \leftarrow$

$male(paul) \leftarrow$

$male(tom) \leftarrow$

and the examples are:

$e_1 = father(paul, an)$

$e_2 = father(tom, jef)$

Then:

$C_1 = father(paul, an) \leftarrow parent(paul, an) \wedge parent(tom, jef) \wedge male(paul) \wedge male(tom)$

$C_2 = father(tom, jef) \leftarrow parent(paul, an) \wedge parent(tom, jef) \wedge male(paul) \wedge male(tom)$

$$rlgg(C_1, C_2) = father(X, Y) \leftarrow \begin{array}{l} parent(X, Y) \wedge male(X) \wedge parent(paul, an) \\ \wedge parent(tom, jef) \wedge male(paul) \wedge male(tom) \end{array}$$

It is important to note that models of a theory are not always finite. If the theory contains arbitrary clauses, the model can be infinite — e.g. if the theory contains recursive clauses. Infinite models result in infinite clauses C_1 and C and in an infinite rlgg.

When the model is very large, the rlgg can also become very large, which is undesirable. Therefore a means to restrict the size of the clauses and model is needed.

Muggleton and Feng [Muggleton 90] solve the problem in three different steps: they replace the model by a finite approximation — *h-easy model* —, they remove redundant literals from the computed rlgg and they bias the clause C_1 and C to take only into consideration relevant literals.

A discussion of the concept of h-easy model, considered within the framework of the Golem system will be treated in a future work.

6 Conclusions

As pointed out before, among the major drawbacks of the traditional machine learning systems, there are those related to

- limited expression of background knowledge, and
- lack of relational descriptions.

Inductive Logic Programming is an approach to Machine Learning which, at certain level, solves those problems by intensively using background knowledge and by allowing a richer concept description language — the language of logic programs.

Although being a new tendency in Machine Learning which attempts to overcome some of the limitations of traditional inductive concept learning methods, ILP has some problems which need to be solved, mostly those related to the intractability of the search space.

What is intended next is to investigate how the existing ILP methods deal with those problems and what kind of approach can be advocate in order to solve them.

To accomplish that, the Golem ILP based system will be used and experienced in order to analyse its scope and evaluate its performance on learning in different domains.

References

[Buntine 86] Buntine W. *Generalised Subsumption.* In Proceedings of the European Conference on Artificial Intelligence, London, 1986.

[Chang 73] Chang, C.L.; Lee, R.C.T. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[De Raedt 92] De Raedt, L. *Personal Communication*, 1992.

[Džeroski 92] Džeroski, S.; Muggleton, H.S.; Russell, S. *PAC - Learnability of Determinate Logic Programs.* In Proceedings of ISSEK Workshop'92, Slovenia, September, 1992.

[Lloyd 84] Lloyd J.W. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Mendelson 79] Mendelson, E. *Introduction to Mathematical Logic.* (2nd Ed.) Van Nostrand, Princeton, 1979.

[Muggleton 88] Muggleton, S.H. *Inverting the Resolution Principle.* Machine Intelligence 12, Oxford University Press, pp 93-102, 1988.

[Muggleton 90] Muggleton, S.H.; Feng, C. *Efficient Induction of Logic Programs.* In Proceedings First Conference on Algorithmic Learning Theory, Tokyo, pp 368-381, 1990.

[Muggleton 91] Muggleton, S.H. *Inductive Logic Programming.* New Generation Computing, Vol. 8, pp 295-318, 1991.

[Nilsson 80] Nilsson, N.J. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, CA, 1980.

[Plotkin 71] Plotkin G.D. *Automatic Methods of Inductive Inference.* Ph. D. thesis, Edinburgh University, August 1971.

[Shoenfield 67] Shoenfield J. *Mathematical Logic.* Addison-Wesley, Reading, Mass., 1967.

N O T A S D O I C M S C

Nº 129/93 - FRANCO, J.L.; MONARD, M.C. - Programacao logica e fluxo em redes

Nº 128/93 - CARVALHO, A.N; RUAS Fº, J.G. - Global attractors for parabolic problems in fractional power spaces

Nº 127/92 - LEME, E.C. - A note on "estimating the fraction of population in an income bracket using pareto distribution"

Nº 126/92 - MORABITO, R.N.; ARENALES, M.N. - Staged and constrained two-dimensional cutting problems: a new approach

Nº 125/92 - RUAS, M.A.S. - On the equisingularity of families of corank 1 generic germs

Nº 124/92 - BIASI, C.; MENDES, M.D.C. - Generalization of Fibonacci sequence and application

Nº 123/92 - MOREIRA, E.S.; TRINDADE JR., O.; BATAGIN JR.; F. - Processamento paralelo atraves da distribuicao em ambientes SunOS

Nº 122/92 - MOCHIDA,D.K.H.; FUSTER, M.C.R.; RUAS, M.A.S. - Geometric characterization of the singularities of height functions on surfaces in 4 - spaces

Nº 121/92 - ACHCAR, J.A. - Some aspects of reparametrization for inferences of K-Out-of-M system reliability in the independent exponenetial case

Nº 120/92 - RODRIGUES, J. - Weighted balanced loss function for the exponential mean time to failure