# Induction of Recursive Logic Programs

Stéphane Lapointe

Thesis submitted to
the School of Graduate Studies and Research
in partial fulfillment of the requirements
for the Master degree in Computer Science

The Ottawa-Carleton Institute for Computer Science
University of Ottawa

ISBN 0-315-93591-X

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

à ma fiancée, Josée

# Acknowledgment

J'aimerais remercier mon superviseur Stan Matwin pour l'appui fourni et l'intérêt constant qu'il a démontré pour mes recherches et idées quelquefois vagues au départ. Il m'a aussi donné l'occasion de rencontrer un bon nombre de chercheurs oeuvrant en apprentissage automatique et en programmation logique inductive. Le Groupe d'Apprentissage Automatique d'Ottawa et le Groupe "Software Re-use" m'ont fourni un excellent environnement de recherche. J'aimerais particulièrement remercier certains membres avec lesquels j'ai eu des discussions constructives, entre autres, Peter Clark et Robert Holte. J'ai aussi bénificié de rencontres avec certains chercheurs d'autres confins de la terre, i.e. Charles Ling et David Aha. Je remercie Terry Copeck pour l'aide apportée au style du texte et format de la version finale.

Je remercie celle qui partage ma vie, Josée, pour l'encouragement et l'appui moral qu'elle a démontrés tout au long de ce projet. Un gros merci à tout ceux qui ont influencés, soient directement ou indirectement, la réalisation de mon degré de maîtrise.

# Table of Contents

# Glossary

# 1 Introduction

Learning is one of the most interesting and difficult problems in artificial intelligence. Much more difficult than its name would suggest, automated learning is a real challenge. Furthermore, some authors consider it a necessary and sufficient condition for a system to be classed as artificially intelligent [Schank, 1991].

Within the artificial intelligence field, the research community that concerns itself with the task of automating learning is called Machine Learning (ML). A perennial issue in machine learning is how knowledge is to be represented. The methodology employed in this thesis is a type of Logic Programming (LP). This technique has been chosen because it can deal adequately with most of the problems associated with the representation issue. Logic programming is also implemented efficiently in the programming language Prolog.

The research area at the intersection of logic programming and machine learning has recently been labeled Inductive Logic Programming (ILP) [Muggleton, 1990]. This thesis is concerned specifically with the induction of *recursive* logic programs. There are two main reasons for this. First, recursion is the basic paradigm in logic programming and as such is indispensable to any interesting logic program. Second, notwithstanding its central role in LP, the problem of inducing recursive programs has been solved only partially.

To make these notions meaningful, let us consider an example of an ILP task. Suppose that as background knowledge a learner already knows how to append two lists. He is then asked to infer the new concept of a REVERSE LIST and given the following set of positive examples:

reverse([],[]),

reverse([a,b],[b,a])

reverse([a,b,c],[c,b,a])

reverse([d,c],[e,d])

and the following set of negative examples:

reverse([a],[b])

reverse([b,c,d],[d,b,c])

These two sets constitute the training set. The learner's task is to create a definition or logic program that represents the REVERSE LIST concept. His background knowledge can be used in the definition, and often the definition is required to be complete and consistent (a complete definition covers (can prove) all positive examples, while a consistent one does not cover any negative example). In this case the following definition would be satisfactory:

reverse([H|T],RL) ⇐ reverse(T,RT), append(RT,[H],RL).[1]

Note that this definition is complete and consistent with respect to the examples given above.

## 1.1 Motivation

Inductive Logic Programming (ILP) systems [Muggleton & Feng, 1990; Quinlan, 1990] characteristically use large training sets to induce their target clauses. Complex terms are often handled by supplying a so-called *h-easy model.* Such a model is a complete sequence of training examples which essentially unfold the term structure down to the level of constants. For instance, in the example above, an 2-easy model of `reverse`, with the constants `a` and `b` only, would be:

[1] In the term `[H|T]`, `H` represents the first element (the head) of the list and `T` represents the other elements (the tail) in the list.

reverse([a,b],[b,a])

reverse([b],[b])

reverse([],[])

reverse([b,a],[a,b])

reverse([a],[a])

reverse([a,a],[a,a])

reverse([b,b],[b,b])

People however do not require anywhere near this many examples to induce the concept. The goal of our work is to reconcile these two observations by successfully inducing logic programs with a small number of examples which are not strongly related to each other.

Let us illustrate this rather intuitive but quite straightforward objective. For example, suppose we have two examples of an unknown recursive concept:

A base example: unknown(X,[X|Y],Y) where X and Y are variables

A complex example: unknown(d,[a,b,c,d,e],[a,b,c,e]).

A person is able to discover the hidden concept with ease—that is, delete the first occurrence of an element (the first argument) in a list (the second argument) to give the remaining list (the third argument). Could a relational learning system discover such a concept from equally few examples ? Existing systems cannot. Our goal is to design a learning system that performs this task with the same ease as a human being.

Existing learning systems use a generalization technique based on *θ-subsumption* because it is tractable—generalizations can be obtained directly. In θ-subsumption this is accomplished by a performing a single inversion of a deductive rule (in inverse resolution the deductive rule is replaced by a resolution step). However, it has been proved that methods based on θ-subsumption are limited to learning clauses which occur only once in any proof of a training example. In other words they cannot produce all relevant generalizations and, in

the case of recursive clauses, often generate overspecific clauses. For this reason we employ a more general notion of generalization based on logical implication in this thesis. It allows examples to be much less informative but still produce all relevant generalizations.

A second shortcoming of current learning systems is their failure to analyze term structure. In general they do not detect interesting similarities between two terms or within a term. We believe that a capacity to perform deep analysis is an important source of power of a learning approach, and basing our system on this perspective, allows us to achieve generalization based on logical implication.

## 1.2 Overview of our approach

We have designed and implemented a system called LOPSTER (inductive LOgic Programming with Sub-unification of TERms, pron. "lobster") which employs the ideas presented in this thesis. In most cases LOPSTER can induce the appropriate recursive clause of a target logic program from just two examples. It can accomplish this for a interesting and useful class of recursive programs with arguments of arbitrary complexity.

The basic idea behind our method is to make use of a source of knowledge left untapped by most other ILP systems: the structure of the terms in examples. Logic programming solves an important class of problems by recursively simplifying complex terms until a base case is reached. We propose a method to induce such programs from a small number of examples, typically just two. It exploits the structural differences between arguments of two examples, `E1` and `E2`, to infer the likely depth of recursion necessary for a logic interpreter to recursively simplify the arguments of `E1` to obtain those of `E2`. To accomplish this requires a capacity to map one term into a unifiable subterm of another. We define a notion called *sub-unification* which expresses the capacity of one term to be

"embedded" in another and implement a mechanism to perform it. The approach that we propose is especially suitable for recursive logic programs. Any ILP system has to address induction of recursive clauses, since recursion is one of the basic techniques of logic programming.

As noted above, our approach in its current implementation is limited to a specific class of logic programs. This class is representative of a broad range of logic programs and the performance of other ILP systems on it is typical of their general power. Experimental results indicate that for the class in question the size of the training set required by LOPSTER is drastically smaller than the number of examples required by GOLEM [Muggleton & Feng, 1990] and FOIL [Quinlan, 1990]. This result in a performance which is an order of magnitude better than that of other systems on the same tasks.

## 1.3 Contributions

The contributions of this thesis are the following:

1) IDENTIFICATION OF A LIMITATION OF EXISTING RELATIONAL LEARNING SYSTEMS. We explain in detail the main limitation of existing systems on the induction of recursive clauses—because these systems operate by inverting the θ-subsumption relation of generalization, they inherit its limitations.

2) DESCRIPTION OF AN IMPROVED METHOD FOR INDUCING RECURSIVE LOGIC PROGRAMS. We propose a method that induces recursive programs on the basis of, in a certain sense, minimum information. Our method requires just two examples: a simple example (often the base case) and a more complex one that needs the first to be proved. This is much more powerful than existing ILP systems which often require tens or hundreds of examples.

3) THE USE OF A POWERFUL KNOWLEDGE REPRESENTATION. The knowledge representation used in LOPSTER is more powerful than most existing systems' in two regards. With some restrictions it accepts clauses in background knowledge. Second, its examples do not need to be ground facts but can incorporate universaily quantified variables.

4) THE USE OF THE NOTION OF GENERALIZATION BASED ON LOGICAL IMPLICATION. Our work is based on an inversion of the implication relation of generalization. We are the first ones to make the leap from $\theta$-subsumption to implication [Muggleton, 1992]. Inverting implication is inherently a more powerful technique.

5) ENUNCIATION OF THE NOTION OF SUB-UNIFICATION. We present a way of analyzing similarity between terms and within a term which is demonstrably useful. In the course of establishing the theoretical foundation of our system, we develop a generalization of unification called *sub-unification* to describe this new technique [Lapointe & Matwin, 1992a; Lapointe & Matwin, 1992b].

6) DEVELOPMENT AND IMPLEMENTATION OF AN EFFICIENT LEARNING SYSTEM. The LOPSTER system implements the ideas set out in this thesis. For the class of problems we have chosen it learns logic programs substantially faster than other systems do.

7) EXTENSIONS OF THE METHOD. We believe it is feasible to extend the learning method presented in this thesis and propose some ways to do so.

8) APPLICATIONS IN PREDICATE INVENTION. We show how our system can solve some difficult even intractable (with existing systems) problems in constructive induction [Ling, 1991b].

## 1.4 Outline

Chapter 2 is an overview of the foundation of existing inductive logic programming systems and includes a brief introduction to our approach. Chapter 3 sets out the theoretical foundations that underpin the LOPSTER system, which is described and illustrated in Chapter 4. Chapter 5 shows how the approach presented here can be extended. In Chapter 6 we apply our system in constructive induction to define new necessary predicates. Finally, we conclude and propose some future interesting research topics in Chapter 7.

# 2 Inductive Logic Programming (ILP)

This chapter explains what ILP is and defines other terms used in this thesis. It discusses two alternative kinds of logic generalization that can be used in ILP and distinguishes them from the rather intuitive coverage notion of generalization. We then present the main requirement of existing ILP systems, that is, their need for basic representative sets. This requirement is a direct consequence of the particular kind of generalization used in all these systems. Last we briefly contrast the existing approaches in ILP with our approach.

## 2.1 What is ILP ?

To answer this, let us first decompose ILP in two parts: *induction* and *logic programs*. A logic program is nothing more than a program represented in a logical formalism (which we will formally define in the next subsection). In this context, induction is the task of automatically generating logic programs from their behaviors. In other words, we generalize from some examples of a target concept, to produce a complete description of the concept; we try to describe what we have never seen. This kind of generalization can also be called induction. Note, that some systems allow examples to be clauses while other systems take only facts (usually ground) as examples. In our system, we take facts with variables universally quantified as examples. For an extended discussion of ILP and particularly inverse resolution and RLGG see [Muggleton, 1990].

## 2.2 Logic programming definitions

Before proceeding it is appropriate to introduce the logic programming terminology and notation used throughout this thesis. For an extended discussion refer to Lloyd [1987] and Genesereth & Nilson [1987].

*Variable*: A variable is represented by a string with the first letter in upper case type.

*Function symbol*: A function symbol is represented by a string with the first letter in lower case type.

*Term*: A term is either a variable or a function symbol followed by bracketed n-tuple of terms. A *constant* is a function symbol. `X, a, f(a) and g(X,f(a))` are terms.

*Predicate symbol*: A predicate symbol is represented by a string with the first letter in lower case type.

*Atom*: An atom is a predicate symbol followed by a bracketed n-tuple of terms. `p(X,f(a))` is an atom if `p` is a predicate symbol.

*Literal*: A *positive literal* is an atom and a *negative literal* is the negation of an atom. Negation is indicated by the symbol "¬".

*Clause*: A clause is a finite set of literals representing the disjunction of these literals with all the variables universally quantified. A *Horn clause* is a clause with at most one positive literal. A *definite clause* is a clause with exactly one positive literal and is represented as $L \Leftarrow L_1, L_2, \ldots, L_n$, where a comma is used to represent the conjunction of literals. L is the *head* of the clause. $L_1, L_2, \ldots, L_n$ is the *body* of the clause. A *unit clause* is a clause with an empty body.

*Logic program*: A logic program is a finite set of definite clauses.

*Fact*: A fact is a unit clause. A *ground fact* is a unit clause without variables.

*Model*: A model of a logic program is a set of ground facts that are true with respect to the logic program.

*h-easy model*: An h-easy-model of a logic program P is a model of P whose ground facts can be derived in at most h steps.

*Substitution*: A substitution is a finite set of pairs $V_i/t_i$, where $V_i$ is a variable and $t_i$ is a term. A substitution is applied to a clause (or a term) by replacing each occurrence of $V_i$ by $t_i$. Given the clause or term C and the substitution $\theta$, the application of $\theta$ to C is noted $C.\theta$. Substitutions can be composed. For instance, {X/g(Y)}.{Y/a} = {X/g(a),Y/a}

*Inverse substitution*: An inverse substitution is a finite set of pairs $t_i/V_i$, where $t_i$ is a term and $V_i$ is a variable. An inverse substitution is applied to a clause (or a term) by replacing each occurrence of $t_i$ by $V_i$. Let C be a clause (or a term) and $\theta^{-1}$ be an inverse substitution, the application of $\theta^{-1}$ to C is noted $C.\theta^{-1}$.

*Unifier*: A unifier of two literals L1 and L2 is a substitution $\theta$ such that $L1.\theta = L2.\theta$. The *most general unifier* (mgu) of two literals L1 and L2 (the mgu exists only if L1 and L2 are unifiable) is a unifier $\theta$ of L1 and L2 such that for any unifier $\sigma$ of L1 and L2 there exists a substitution $\gamma$ such that $\sigma=\theta.\gamma$.

We illustrate the definitions by an example. Here is a logic program P:

p(a).

q(b).

p(X) ⇐ q(X).

r(b) ⇐ p(b).

p(a) and q(b) are unit clauses as well as ground facts. {p(a), q(b), p(b)} is a 1-easy model of P. {p(a), q(b), p(b), r(b)} is a model of P as well as a h-easy

model if h >1 (note that r(b) is obtained in two steps). $\theta$ = {X/b} is a substitution and (p(X) ⇐ q(X)).$\theta$ = p(b) ⇐ q(b). The mgu of q(b) and q(X) is $\theta$ = {X/b}. If $\theta$ = {X/b} then $\theta^{-1}$ = {b/X} is an inverse substitution.

### 2.2.1 Resolution

Inverse resolution plays an important role in our design, but before discussing it we must define the resolution principle from which it is derived. *Resolution* is a consistent but incomplete deductive inference rule first proposed by Robinson [1965]. Its variant *refutation resolution* is both complete and consistent.

Let C1 and C2 be two clauses, L1 ∈ C1 and L2 ∈ C2 literals, and $\theta$ the mgu of ¬L1 and L2, then C defined by (C1-{L1}).$\theta$ ∪ (C2-{L2}).$\theta$ is a resolvent of C1 and C2. Two clauses may have several resolvents for different choices of the resolved literals L1 and L2.

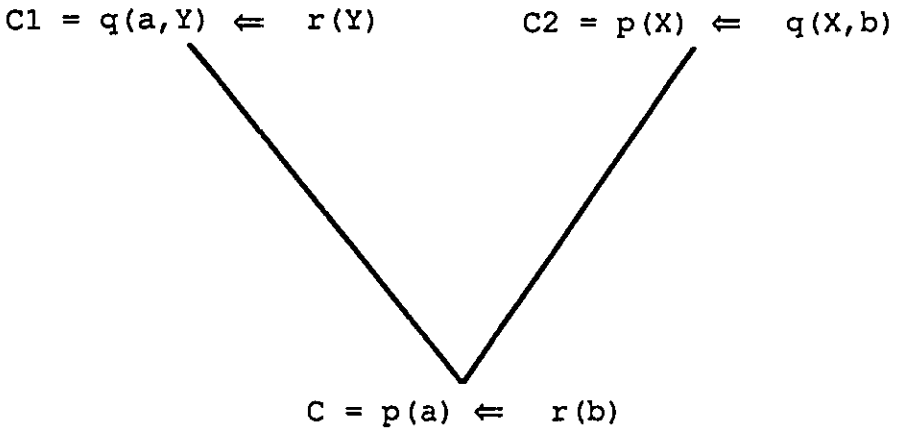


Figure 2.1: An example of resolution.

Fig. 2.1 illustrates an example of the resolution principle where L1 = q(a,Y), L2 = ¬q(X,b) and $\theta$ = {X/a,Y/b} is the mgu of ¬q(a,Y) and ¬q(X,b). The resolvent is

$C = (\{q(a,Y), \neg r(Y)\} - \{q(a,Y)\}).\{X/a, Y/b\} \cup (\{p(X), \neg q(X,b)\} - \{\neg q(X,b)\}).\{X/a, Y/b\} = \{p(a), \neg r(b)\}$

Observe that the "natural" mode of inference (if p and $p \Rightarrow q$ then q) called *modus ponens* can be realized with resolution; the resolvent of p and $\neg p \vee q$ is q.

## 2.3 Generalization

Niblett [1988] provides an extended discussion of generalization from which we will select the ideas which are relevant to our method. There is some disagreement in the AI community about what generalization is. Some people adopt an intuitive coverage notion of generalization while others prefer a definition in formal logic.

According to the coverage notion, a statement S1 is more general than a statement S2 roughly speaking if S1 "covers" S2, i.e. if the set of elements satisfying S1 includes the set of elements satisfying S2. Fig. 2.2 shows this situation.

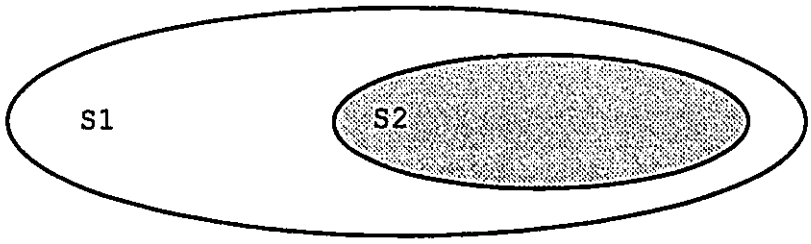


Figure 2.2: Coverage notion of generalization.

For example, the statement `X lives in Canada` is more general than the statement `X lives in Ottawa` because the set of persons living in Canada covers, or contains, the set of persons living in Ottawa[2].

The logic definition of generalization is as follows: `S1` is more general that `S2` if `S1` $\Rightarrow$ `S2`, that is, if we can deduce `S2` from `S1`. In the example above `X lives in Ottawa` is more general than `X lives in Canada`, because if X lives in Ottawa then X lives in Canada. The two definitions are in direct conflict.

The coverage notion of generalization is awkward. Consider this example:

(1) $A \vee B$

(2) $A \vee B \vee C$

Statement (1) is more general than statement (2) according to the logic definition while (2) is more general than (1) according to the coverage notion. Let us analyze these statements:

(3) $A \vee \neg B \equiv A \Leftarrow B$

(4) $A \vee \neg B \vee \neg C \equiv A \Leftarrow B \wedge C$

If (2) is more general than (1) as the coverage definition suggests, then (4) should be more general than (3). However this is not the case. Intuitively speaking, the statement `If Stan plays tennis then Stan beats Stéphane` is more general than the statement `If Stan plays tennis and Stan plays well then Stan beats Stéphane`. In this work, we have adopted the logic definition of generalization which is consistent and formal.

---

[2]It is possible to define, for non-propositional statements of 1st order logic, the coverage notion of generalization using the notion of model. If the set of models of statement `S1` is a superset of the set of models of statement `S2` then `S1` is more general than `S2`. The disadvantage is that the intuitive meaning of the coverage notion of generalization is then lost.

There are two main kinds of logic generalization. The first is based on *implication* or subsumption. We say that a clause C subsumes a clause D if C $\Rightarrow$ D, that is, C logically implies D. The second kind of logic generalization is based on *θ-subsumption*. We say that a clause C θ-subsumes a clause D if D $\supseteq$ C.θ for a substitution θ (where $\supseteq$ denotes set inclusion). Niblett [1988] shows that implication is a strictly stronger relation than θ-subsumption.

In the following paragraphs we characterize both kinds of generalization relative to a given background knowledge BK. Let BK be a set of Horn clauses and SE be a set of examples such that BK does not prove SE. We want to induce a clause C such that BK and C prove SE. We say that C is a generalization of SE relative to BK. In the context of generalization under θ-subsumption, the clause C can be used only once during the proof of an example of SE (see Theorem 3.1 in [Niblett, 1988]). This constraint is built into the definition of RLGG (Relative Least General Generalization) as used in GOLEM [Muggleton & Feng, 1990], but it does not apply in generalization based on implication. Consequently this distinction precisely characterizes the difference between θ-subsumption and implication. It is quite clear that generalization under θ-subsumption is inadequate for induction of recursive clauses given that a recursive clause must usually[3] be employed several times to prove an example. For instance, the example `member(3,[1,2,3,4])` is only "useful" if `member(3,[2,3,4])` belongs to BK because then a single use of the clause being induced suffices to reduce the former example to the latter. These two examples form a basic representative set of the recursive clause of `member`.

[3]unless the subgoal directly inferred by the clause is in BK.

## 2.4 Basic representative set

The characteristics of a good set of examples were studied by Ling [1991a]. Intuitively a good set of examples directly represents the clause or the logic program to be induced. A basic representative set, which is a good set of examples, represents an instance of a clause or a logic program.

To expand on this we must first explain what a ground instance of a clause is. If C is a clause and θ is a ground substitution for C, i.e. a substitution that substitutes a ground term for every variable occurring in C, then C . θ is a ground instance of C. For instance,

reverse([a,b,c],[c,b,a]) ⇐
    reverse([b,c],[c,b]),append([c,b],[a],[c,b,a]).

is a ground instance of the clause

reverse([H|T],RL) ⇐ reverse(T,RT),append(RT,[H],RL).

A basic representative set of a logic program P is a set of ground facts obtained by taking, for each clause C of P, all ground facts in any ground instance of C. For example,

{reverse([a,b,c],[c,b,a]), reverse([b,c],[c,b]), append([c,b],[a],[c,b,a])}

is a basic representative set of the clause above.

The relationship between θ-subsumption generalization and basic representative sets is now clear. Generalization under θ-subsumption requires at least one basic representative set of the clause to be induced. If basic representative sets are provided then generalization under θ-subsumption can be used productively.

## 2.5 Existing approaches

Because implication is often considered intractable, recent learning systems [Muggleton & Feng, 1990; Quinlan, 1990; Muggleton & Buntine, 1988; Rouveirol, 1991; Wirth, 1989; Wirth & O'Rorke, 1991] use the weaker generalization based on θ-subsumption. As we have just shown, a drawback of θ-subsumption is its need of at least one basic representative set among the examples[4]. To provide such good sets of examples we require information about the recursive decomposition occurring in the clause, but this information is not available since we do not yet know the clause to be induced. Some systems [Muggleton & Feng, 1990; Quinlan, 1990; Wirth & O'Rorke, 1991] resolve this dilemma by providing an h-easy model, essentially all examples to depth h, and identifying good sets of examples in the learning phase. One may, however, doubt the availability of such models. In many cases a generator would be needed to produce an h-easy model, but in order to build such a generator one would need to know the clause being induced, that is to say, the concept to be learned.

Let us explain in more detail each of a number of well-known approaches and the method they use to select the literals forming the clause. Our claims about θ-subsumption and basic representative sets will then become a matter of fact.

### 2.5.1 Least General Generalization (LGG)

This generalization operator was formulated by Plotkin [1970]. The LGG is applied to a set of clauses or terms. It produces a clause or term C more general under θ-subsumption than every element in the set, such that there is no clause D strictly less general under θ-subsumption than C that is also more general under θ-subsumption than every element in the

[4]this necessary condition is in general not sufficient.

set. The LGG of two clauses C1 and C2 is the greatest lower bound of C1 and C2 within the clause lattice induced by the relation θ-subsumption.

We now give the operational definition of LGG. We first need to define the LGG of two terms e1 and e2, denoted lgg(e1,e2):

if e1 = e2, then lgg(e1,e2) = e1;
if e1 = $f(s_1,s_2,...,s_n)$ and e2 = $f(t_1,t_2,...,t_n)$,
then lgg(e1,e2) = $f(lgg(s_1,t_1),lgg(s_2,t_2),...,lgg(s_n,t_n))$;
otherwise lgg(e1,e2) is a variable V used everywhere throughout the clause as the LGG of these terms.

Note that the following equality holds for n terms or clauses $e_1, e_2, \ldots, e_n$: $lgg(e_1, e_2, \ldots, e_n) = lgg(e_1, lgg(e_2, \ldots, e_n))$. The LGG of two atoms $p(s_1, s_2, \ldots, s_n)$ and $p(t_1, t_2, \ldots, t_n)$ is $p(lgg(s_1,t_1), lgg(s_2,t_2), \ldots, lgg(s_n,t_n))$. For instance, lgg(p(g(a),a),p(g(b),b)) = p(g(X),X). Finally, here is the operational definition of the LGG of two clauses C1 and C2:

lgg(C1,C2) = {l: $l_1 \in C1$
and $l_2 \in C2$
and $l_1$ has the same sign and predicate symbol as $l_2$
and $l = lgg(l_1,l_2)$}

The LGG of two clauses might contain up to $n_1.n_2$ literals, where $n_1$ and $n_2$ are the number of literals in C1 and C2 respectively.

let C1 = {p(a,f(b)),¬q(f(b),a),¬r(a)}
and C2 = {p(a,X),p(c,f(b)),¬q(X,c),r(a)}
then lgg(C1,C2) = {p(a,Y),p(Z,f(b)),¬q(Y,Z)}

where the variable Y represents the LGG of the pair <f(b),X> and the variable Z represents the LGG of the pair <a,c>.

The most important use of the LGG in any ILP system is to generalize a set of unit clauses. The base case of a logic program can often be obtained by taking the LGG of appropriate ground facts. For instance,

lgg(append([],[a,b],[a,b]),append([],[c],[c])) = append([],[X|Y],[X|Y]).

The LGG operator must be manipulated very carefully to avoid overgeneralization. For instance,

lgg(append([],[a,b],[a,b]),append([c],[],[c])) = append(X,Y,[Z|W]).

### 2.5.2 Relative Least General Generalization (RLGG)

Plotkin [1971] defines the Relative Least General Generalization (RLGG) as a unique clause which covers (or is more general under θ-subsumption than) a given set of examples with respect to a given background knowledge. For simplicity assume an example set composed of a pair of ground facts e1 and e2. Let BK be a set of background clauses such that BK $\nvdash$ e1 and BK $\nvdash$ e2. The clause C is said to be the RLGG of e1 and e2 relative to BK whenever C is the least general clause within the θ-subsumption lattice for which BK∧C $\vdash$ e1∧e2, C being used only once in the derivation of both e1 and e2. The fact that C is used only once highlights the main limitation of this method particularly for recursive concepts. This restriction on C comes from the use of θ-subsumption generalization.

An operational definition of the RLGG follows. Let $a_1 \wedge a_2 \wedge \ldots$ be a model of BK.

rlgg(e1,e2) = lgg(C1,C2) =

$$lgg(e1,e2) \Leftarrow \bigwedge_{\substack{a_i \text{ and } a_j \text{ with} \\ \text{same predicate symbol} \\ \text{and same sign}}} lgg(a_i, a_j)$$

where $C1 = e1 \Leftarrow a_1 \wedge a_2 \wedge \ldots$ and $C2 = e2 \Leftarrow a_1 \wedge a_2 \wedge \ldots$

The model contains the background knowledge and all the examples. When the background knowledge includes a recursive definition then in general the model is not finite. This invoves the use of an h-easy model which is finite.

An example from Quinlan [1991b] illustrates the computation of the RLGG of two scenes s1 and s2 (shown in Fig. 2.3). The examples are:

scene(s1),scene(s2).

The background knowledge is:

on(s1,a,b),on(s2,f,e),
left_of(s1,b,c), left_of(s2,d,e),
circle(a),circle(f),
square(b),square(d),
triangle(c),triangle(e).

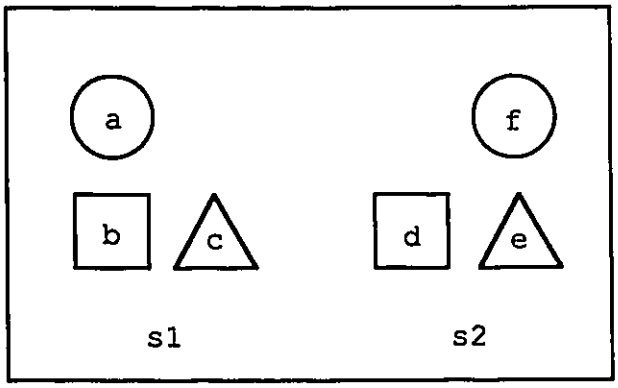


Figure 2.3: Two scenes.

The RLGG of the two scenes would be computed thus[5]:

rlgg(scene(s1),scene(s2)) =

lgg(scene(s1),scene(s2)) ⇐

lgg(on(s1,a,b),on(s2,f,e)),

lgg(left_of(s1,b,c),left_of(s2,d,e)),

lgg(circle(a),circle(f)),

lgg(square(b),square(d)),

lgg(triangle(c),triangle(e)).

Introducing the variables `A, B, C, D` and E for the LGGs of the pairs `<s1,s2>`, `<a,f>`, `<b,e>`, `<b,d>` and `<c,e>` respectively produces:

rlgg(scene(s1),scene(s2)) =

scene(A) ⇐

on(A,B,C),left_of(A,D,E),

circle(B),square(D),triangle(E).

This clause represents the maximal common information conveyed in each scene. It says that in any generalization of these scenes, a circle is on something and a square is left of a triangle.

RLGG performs well in this example. In general it produces satisfactory results when recursive concepts are not involved, because its restriction to a single use of the induced clause in the proof does not then apply. The situation is not as satisfactory with recursive concepts because either basic representative sets must be provided or the induced clause is too specific.

[5]Comma is used as conjunction.

The two ILP systems GOLEM [Muggleton & Feng, 1990] and SERIES [Wirth & O'Rorke, 1991] are based on the RLGG method. In each system background knowledge is represented by a h-easy-model and examples are ground facts. For GOLEM to compute the RLGG of two examples it requires at least two basic representative sets of the target clause. In the example above,

scene(s1) ⇐ on(s1,a,b), left_of(s1,b,c), circle(a), square(b), triangle(c).

scene(s2) ⇐ on(s2,f,e), left_of(s2,d,e), circle(f), square(d), triangle(e).

form the two basic representative sets. In general, but not in the example above, most of the literals in the h-easy-model do not belong to any of the two basic representative sets. That causes an RLGG derivation to contain a large number of irrelevant literals. For example, consider the following subset of an h-easy model of the LIST MEMBERSHIP concept:

member(a,[a])
member(a,[b,a])
member(a,[c,b,a])
member(e,[e])
member(e,[f,e])
member(e,[g,f,e])

There is no background knowledge so the model contains only these examples. The RLGG of `member(a,[c,b,a])` and `member(e,[f,e])` is:

lgg(member(a,[c,b,a]),member(e,[f,e])) ⇐
    lgg(member(a,[b,a]),member(e,[e])),
    lgg(member(a,[a]),member(e,[e])),
    lgg(member(a,[a]),member(a,[b,a])),
    lgg(member(e,[e]),member(e,[g,f,e])),
    lgg(member(a,[a]),member(e,[g,f,e])),

...

There are fifteen literals in the RLGG clause. Introducing variables, we obtain:

member(A,[B,C|D]) ⇐ member(A,[C|D]),
member(A,[A]),
member(a,[E|F]),
member(e,[G|H]),
member(A,[I|J]),
...

Only the first literal in the body of the RLGG clause is needed. The other fourteen literals are irrelevant and therefore harmful to an efficient computation. To eliminate harmful literals and keep relevant literals GOLEM checks whether each literal has a specific property and keeps only those do. Roughly speaking, the criterion is whether WRT the background knowledge there is a unique instantiation of a given output argument in the literal that corresponds to an instantiation of input arguments. If so, the output argument (a term) is determinate and is retained. In `plus(X,Y,Z)`, the output argument `Z` is determinate, but in `bigger_than(X,Y)`, the output argument `Y` is not determinate since many `Y`'s are bigger than a given `X`.

SERIES [Wirth & O'Rorke, 1991] also requires two basic representative sets and computes the RLGG of two examples. It successively adds literals in the body of the clause that are the LGG of two ground facts extracted from the h-easy model. When doing so SERIES stresses critical inputs/outputs, that is, it tries to add a literal that uses input/output variables not yet used in the clause[6]. When an appropriate literal cannot be added to the clause it tries to invent a new predicate using critical inputs/outputs as arguments. Most of the time however SERIES fails to induce the definition of this new predicate because it is not

[6] to prune the search, it uses dependency graphs.

often that an h-easy model, or even two appropriate basic representative sets, of the new predicate can be collected.

### 2.5.3 Inverse Resolution (IR)

As its name suggests, the inverse resolution is the inversion of a resolution step. Here we focus on the V-operator which inverts a binary resolution step. Fig. 2.4 is a diagrammatic representation of this operator. The V-operator derives the clause on one arm of the "V" given the clause on the other arm and the clause at the base. The literal resolved on is positive (+) in C1 and negative (-) in C2. The *abstraction operator* constructs C2 given C1 and C. Conversely given C2 and C, the *identification operator* constructs C1. We derive the equation of the absorption operator from the resolution equation:

$$C = (C1-\{L1\}).\theta_1 \cup (C2-\{L2\}).\theta_2$$

Let $D = (C1-\{L1\}).\theta_1 - (C2-\{L2\}).\theta_2$

then $\underline{C2 = ((C - D) \cup \{\neg L1\}\theta_1).\theta_2^{-1}}$

With the resolved literal excluded, D represents those literals in C1 which are distinct from the literals in C2 once substitutions are made. There is a whole range of possible solutions depending on the choice of D, $(C1-\{L1\}).\theta_1 \supseteq D \supseteq \emptyset$, the choice of $\theta_1$ and the choice of $\theta_2^{-1}$. The most specific solution for the absorption operator is obtained when both D and $\theta_2^{-1}$ are empty. This yields:

$$\underline{C2 = C \cup \{\neg L1\}.\theta_1}$$

Note that $\theta_1$ can be partly determined from C and C1.

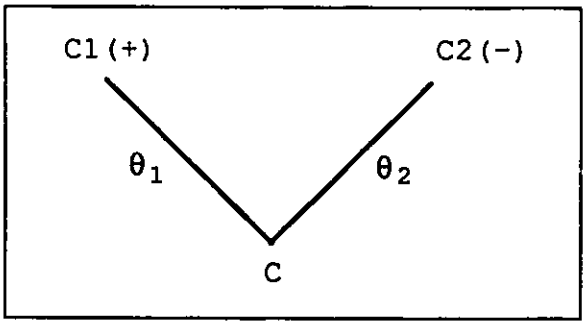


Figure 2.4: The V-operator.

Application of the most specific absorption operator on C1 = {p(a,X),¬q(X)} and C = {¬q(b),r(c)} gives C2 = {¬q(b),r(c),¬p(a,b)} with L1 = p(a,X) and $\theta_1$ = {X/b}. If C1 is a definite clause there is a unique choice for the positive literal L1 in C1.

The inversion operator can only deal with θ-subsumption generalization because it inverts resolution steps one at a time and, as we might expect, the induced clause has exactly one occurrence in the proof of the base of the "V". For an extended discussion and comparison of IR operators, refer to Ling & Narayan [1991]. All current implementations of IR in the literature, CIGOL [Muggleton & Buntine, 1988], ITOU [Rouveirol, 1991; Rouveirol & Puget, 1990] and LFP2 [Wirth, 1989], rely on the V-operator as their basic inference rule. Fig. 2.5 illustrates this situation. These systems all use generalization based on θ-subsumption and consequently inherit the shortcomings of θ-subsumption in inducing recursive clauses.

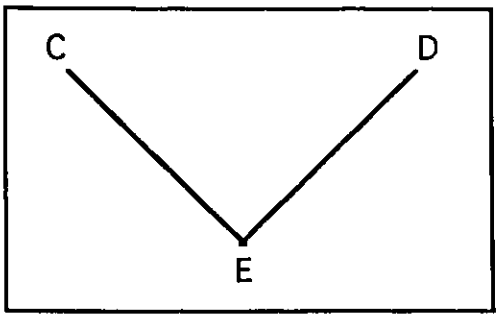


Figure 2.5: Induction with the V-operator.
D is a background clause, E is an example and C is the clause to be induced.

A typical proof by resolution involving a recursive clause C requires several occurrences of C in the proof. Fig. 2.6 illustrates this situation. To make things clearer, substitute `append([A|B],C,[A|D]) ⇐ append(B,C,D)` for C, `append([],L,L)` for D, and `append([a,b,c],[1,2],[a,b,c,1,2])` for E in the figure respectively. C and D yield E (up to a substitution) by three of the binary resolution steps shown. However, the V-operator cannot be used to obtain C from D and E—the situation described in this figure is beyond the scope of IR methods based on θ-subsumption. Our goal is this thesis to deal specifically with this situation, because the induced clause C in the Fig. 2.6 is more general and complete than the induced clause C in Fig. 2.5. For instance, for D and E as defined above, the induced clause C in Fig. 2.5 could be:

append([a,b,c], [1,2], [a,b,c,1,2]) ⇐ append([], L, L).

or in the best case:

append([A,B,C|D], E, [A,B,C|F] ⇐ append(D, E, F).

which are less general than the clause C in Fig. 2.6.

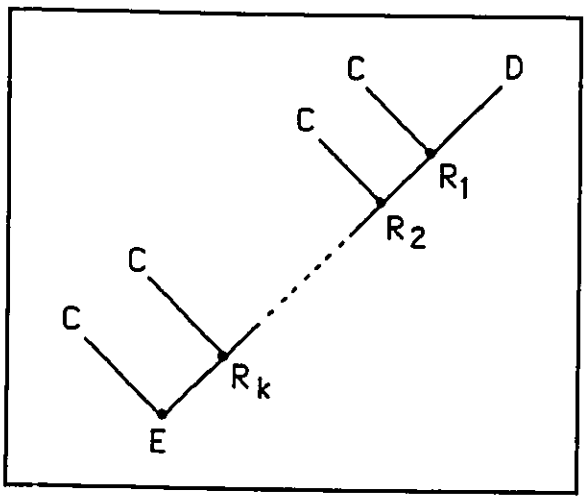


Figure 2.6: Beyond The V-operator.

D is a background clause, E is an example and C is the clause to be induced. $R_1$ is the resolvent of C and D, and $R_i$ is the resolvent of C and $R_{(i-1)}$, for $i = 2, 3, \ldots, k$.

### 2.5.4 Information gain

FOIL [Quinlan, 1990] is a system based on information gain. Literals to be included in the solution clause are chosen on the basis of the gain in information they provide. This system begins operation with a description that covers every example. It successively adds literals in the body of the clause until no negative examples are covered. When making an addition FOIL evaluates the amount of information to be gained by adding each candidate literal and picks the one that offers the best gain. The gain calculation is based on the improved positive and negative coverage of the clause with the selected literal added. To compute the gain FOIL relies on negative examples substantially. In fact, it needs large sets of negative examples or the gain calculation lose its significance. However in practice large sets of negative examples are not available. Furthermore, because it deals only with

generalization under θ-subsumption, FOIL needs at least one basic representative set among the examples. In order to get such sets FOIL is used with an h-easy-model.

## 2.6 Our approach

We believe there is no reason why generalization based on implication should not be used as the basis for a learning system. The shortcomings of θ-subsumption encouraged us to work with the straightforward notion of implication. So, we do not require basic representative sets at all for induction of recursive clauses. In fact, we show how to induce a recursive clause with only two examples such that one is reducible to the other.

### 2.6.1 Overview of our method

Before weighing in with technical details in chapter 3, we would like to introduce the general idea of our method using the schema in figure 2.7. We begin with two examples of the target concept, `E1` and `E2`. The task is to induce a recursive clause that can prove `E2` using `E1`. The first operation is to sub-unify `E1` with `E2`, that is, to find a substructure (*subterm*) in `E2` that can be unified with `E1`. Once this is done the term in `E2` surrounding the unified subterm (*embedding term)* is inspected for repetitive structure (*generating term*; represented by a layer of bricks in the figure). A recursive clause is induced from the generating term directly. The induced clause removes layers one at a time from the examples (see botom part of the figure). In the case described in the figure, the induced clause is used three times to prove `E2`. The proof ends on the example `E1`, which is why `E2` is said to be (recursively) reducible to `E1`.

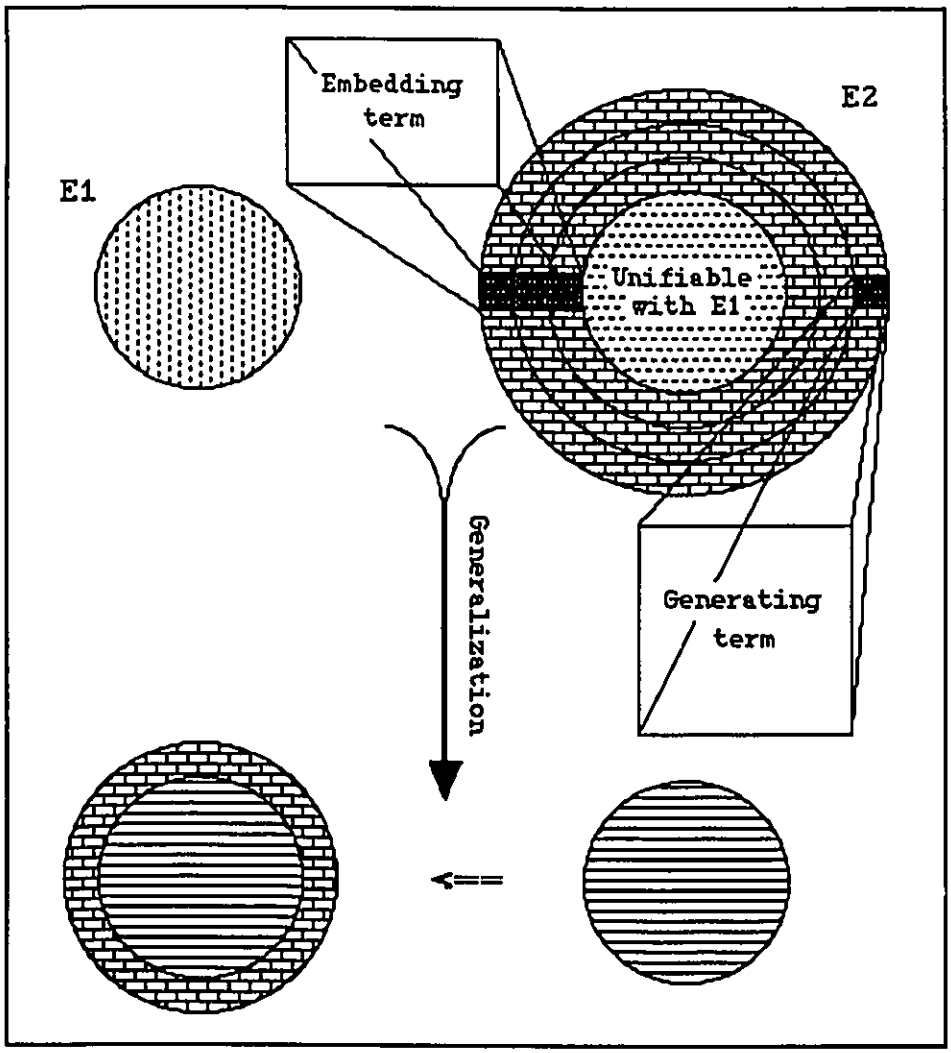


Figure 2.7: Overview of our method.

### 2.6.2 Criterion for induction

It may be possible to induce more than one clause for a given pair of examples. There are usually several subterms in each example which could be unified with the other example. And, while the number is usually small in practice, there may be more than one generating

term for a given sub-unification. Which clause or set of clauses are most interesting from our point of view?

Let BK be a set of background Horn clauses and E an example such that BK does not prove E. Let C be a clause such that together BK and C prove E. The depth of C with respect to BK and E is the number of times C is used in the proof of E or, if C is recursive, the depth of the recursion of C in the SLD-refutation proof of E. The recursive case is the one we are interested in.

Our criterion for the selection of a clause to be induced is this: we want to induce a clause C such that BK and C prove E and the *depth* of C is maximum. Note that, for a given depth there may be more than one such clause. We are interested in the clause with the *most predictive power*, i.e. it will perform correctly on as large a class of examples as possible. We believe this last condition will be satisfied by the clause which is used the greatest number of times in the proof of E.

# 3 Theoretical foundations

This chapter introduces two basic components of our learning system. These are the sub-unification mechanism and the notion of a generating term. To facilitate understanding we present examples immediately after the definitions that employ them rather than in a group at the end. *Literal* always means a positive literal. A letter with a superscript, $t^V$ or $GT^V$, represents a term in which there is exactly one occurrence of the superscripted variable. Thus $t^V$ could be `s(s(V))` but not `f(V, V)`.

## 3.1 Sub-unification

DEFINITION 1:

We recursively define *subterms* of a term `t` as:

- t is a subterm of t.
- If $t=f(t_1,t_2,...,t_n)$, subterms of $t_1,t_2,...,t_n$ are subterms of t.

For instance, there are four subterms of `f(g(a),X)`. They are `f(g(a),X)` itself, `g(a)`, `a` and `X`.

DEFINITION 2:

A *sub-unifier* of a term `t1` in a term `t2` is a pair $<\theta, t^V>$ where $\theta$[7] is a substitution and $t^V$ is a term such that $t^V.\{V/s\} = t2$ for a subterm `s` of `t2` and $s.\theta = t1.\theta$.

[7] $\theta$ does not refer to V, it only refers to variable in `s` and `t1`.

If a sub-unifier of t1 exists in t2 then t1 is *sub-unifiable* in t2. We call $t^V$ the *embedding term* of the sub-unification. Thus a term t1 is sub-unifiable in a term t2 if and only if t1 is unifiable with a subterm s of t2.

Example 1:

Let t1 = g(a,X) and t2 = f(g(Y,Z),W,c). Some of the sub-unifiers $<\theta, t^V>$ of t1 in t2 are:

| | | |
|---|---|---|
| 1) $\theta = \{Y/a, X/Z\}$ | $t^V = f(V,W,c)$ | where s = g(Y,Z) |
| 2) $\theta = \{Y/a, X/b, Z/b\}$ | $t^V = f(V,W,c)$ | where s = g(Y,Z) |
| 3) $\theta = \{W/g(a,X)\}$ | $t^V = f(g(Y,Z),V,c)$ | where s = W |
| 4) $\theta = \{Y/g(a,X)\}$ | $t^V = f(g(V,Z),W,c)$ | where s = Y |

In general, and in this example in particular, certain variables will accept any instantiation. This means there are, in some cases, an infinite number of sub-unifiers. For example, the constant b in the substitution of sub unifier 2 can be replaced by anything. The following definition characterizes the finite set of sub-unifiers in which we are interested.

DEFINITION 3:

A sub-unifier $<\theta, t^V>$ of a term t1 in a term t2 is called a *most general sub-unifier* (mgsu) of t1 in t2 if and only if for each sub-unifier $<\sigma, t^V>$ of t1 in t2, there exists a substitution $\gamma$ such that $\sigma = \theta.\gamma$.

Consider Example 1 in the light of this definition. Sub-unifier 2 is not a mgsu of t1 in t2 because sub-unifier 1 is more general; there is no substitution $\gamma$ such that {Y/a,X/Z} = {Y/a,X/b,Z/b}.$\gamma$

The notion of mgsu is very similar to the notion of mgu (most general unifier) . In fact, $<\theta, t^V>$ is a mgsu of t1 in t2 if $\theta$ is a mgu of t1 in s, where $t^V.\{V/s\} = t2$.

DEFINITION 4:

A sub-unifier <θ, $t^V$> of a term t1 in a term t2 is *trivial* if and only if the unified subterm s of t2 is a variable and s is a proper subterm of t2 (i.e. s is not t2). Otherwise it is *non-trivial.*

The concept of triviality requires explanation. Any variable subterm s of a term t2 is unifiable with any term t1. However this kind of sub-unification is useless to us unless s is t1 itself, i.e. the sub-unification is also a unification[8]. In our context it does not make sense for any other subterm of t2 unifiable with t1 to be a variable, because that would mean that we could keep decomposing the term t2 forever.

Consider Example 1 again. The sole non-trivial mgsu is sub-unifier 1. We already know that sub-unifier 2 is not a mgsu even though it is non-trivial. Furthermore, sub-unifier 3 and sub-unifier 4 are trivial because in each s is both a variable and a proper subterm of t2.

How many mgsu's can there be for a pair of terms ? Observe that if t1 is any term and t2 = X, there is only one mgsu: θ = {X/t1} and $t^V$ = V (θ is an unifier of t1 and t2). In general for a pair of terms there may exist several mgsu's (a finite number) employing different embedding terms. Here we are only interested in the set of non-trivial mgsu's which is of finite (usually very small) size.

The notion of sub-unification can be extended to the case of two literals with the same predicate symbol and the same arity. This is just an extension of sub-unification of a pair of terms to several pairs of terms. It will allow us to sub-unify two examples that are facts containing variables.

[8]In fact unification is an interesting kind of sub-unification that we need to consider.

DEFINITION 5:

A *sub-unifier* of a literal $p(t_1, t_2, \ldots, t_n)$ in a literal $p(s_1, s_2, \ldots, s_n)$ is a (n+1)-tuple $<\theta, t_1^V, t_2^V, \ldots, t_n^V>$ where $\theta$ is a substitution such that $<\theta, t_i^V>$ is a sub-unifier of $t_i$ in $s_i$, for i = 1,2,...,n. The sub-unification is *non-trivial* if each $<\theta, t_i^V>$ is a non-trivial sub-unifier of $t_i$ in $s_i$.

DEFINITION 6:

A sub-unifier $<\theta, t_1^V, t_2^V, \ldots, t_n^V>$ of a literal $p(t_1, t_2, \ldots, t_n)$ in a literal $p(s_1, s_2, \ldots, s_n)$ is a *most general sub-unifier* (mgsu) if and only if for each sub-unifier $<\sigma, t_1^V, t_2^V, \ldots, t_n^V>$ of $p(t_1, t_2, \ldots, t_n)$ in $p(s_1, s_2, \ldots, s_n)$ there exists a substitution $\gamma$ such that $\sigma = \theta.\gamma$.

Example 2:

Although Prolog list notation is used for convenience, it should be remembered that lists are terms built with a function such as `cons`.

Let L1 = append([], X, X)
and L2 = append([a,b,Y], [1,2], [a,b,Y,1,2])

There are exactly five non-trivial mgsu's $<\theta, t_1^V, t_2^V, t_3^V>$ of `L1` in `L2`:

| | | | | |
|---|---|---|---|---|
| 1) | $\theta$={X/[1,2]} | $t_1^V$=[a,b,Y\|*V*] | $t_2^V$=*V* | $t_3^V$=[a,b,Y\|*V*] |
| 2) | $\theta$={X/1} | $t_1^V$=[a,b,Y\|*V*] | $t_2^V$=[*V*,2] | $t_3^V$=[a,b,Y,*V*,2] |
| 3) | $\theta$={X/[2]} | $t_1^V$=[a,b,Y\|*V*] | $t_2^V$=[1\|*V*] | $t_3^V$=[a,b,Y,1\|*V*] |
| 4) | $\theta$={X/2} | $t_1^V$=[a,b,Y\|*V*] | $t_2^V$=[1,*V*] | $t_3^V$=[a,b,Y,1,*V*] |
| 5) | $\theta$={X/[]} | $t_1^V$=[a,b,Y\|*V*] | $t_2^V$=[1,2\|*V*] | $t_3^V$=[a,b,Y,1,2\|*V*] |

Let us check that $t_1^V$ of sub-unifier 1 satisfies Definition 2, where `s = []`:

$t_1^V.\{V/s\}$ = [a,b,Y|*V*].{*V*/[]} = [a,b,Y] and s.$\theta$ = []

We define the notion of *input sub-unification* of two literals of which we know the mode declaration. Let $p(t_1, t_2, \ldots, t_m, t_{m+1}, \ldots, t_n)$ be a literal with inputs $t_1, t_2, \ldots, t_m$ and outputs $t_{m+1}, t_{m+2}, \ldots, t_n$. An *input sub-unifier* of a literal $p(t_1, t_2, \ldots, t_m, t_{m+1}, \ldots, t_n)$ in a literal $p(s_1, s_2, \ldots, s_m, s_{m+1}, \ldots, s_n)$ is a (m+1)-tuple $<\theta, t_1^V, t_2^V, \ldots, t_m^V>$ where $\theta$ is a substitution such that $<\theta, t_i^V>$ is a sub-unifier of $t_i$ in $s_i$, for i = 1,2,...,m. *Most general input sub-unifier* (i-mgsu) and the *non-trivial* input sub-unifier are defined in a parallel manner.

For example, $<\{X/s(0)\}, V, s(s(V))>$ is a non-trivial i-mgsu of `multiply(X,0,0)` in `multiply(s(0),s(s(0)),s(s(0)))`, where the first and second arguments of `multiply` are inputs, and the third argument is the output.

### 3.1.1 Relation between sub-unification and unification

Sub-unification is quite similar to unification. In fact, it is a more general concept. Let `t1, t2` be terms or literals. If `t1` and `t2` are unifiable, then `t1` is sub-unifiable in `t2`; however, the opposite is not true. Unification is a special case of sub-unification with the embedding term being a variable. Sub-unification is a powerful tool since it allows us to relate two terms in the same way that unification does, but without previously decomposing one of them. In fact, the embedding term of a sub-unification shows how to decompose a term in order to make it unifiable with another term.

## 3.2 Generating term

The obvious question to raise at this point is how to find the successive decompositions which must be applied to a term. The solution is to find *generating terms*—special kinds of recursive generalization—of the embedding term of a sub-unification.

DEFINITION 7:

A *generating term of depth n>0* of an embedding term $t^V \neq V$ is a term $GT^V$ such that there exists a substitution $\theta$ such that:

$$GT_1^V.\{V/GT_2^V\}.\{V/GT_3^V\}.\ldots.\{V/GT_n^V\}.\theta = t^V$$

where $GT_i^V = GT^V.\{X_1/X_{1i}, X_2/X_{2i}, \ldots, X_m/X_{mi}\}$ and $X_1, X_2, \ldots, X_m$ are all variables occurring in $GT^V$, except $V$. $GT_i^V$ is just $GT^V$ with variables renamed.

The intuition behind this definition is as follows: when a recursive clause is applied to a term, its structure is decomposed several times in the same way. The goal is to find the term that will generate this decomposition.

Example 3:

Let $t^V$=f(0, g(0), f(s(0), g(s(0)), f(s(s(0)), g(s(s(0))), V)))

$GT^V$ = f($X_1$, $X_2$, V) is a generating term of depth 3 of$t^V$.

$GT_1^V.\{V/GT_2^V\}.\{V/GT_3^V\}.q$ = f($X_{11}$,$X_{21}$,f($X_{12}$,$X_{22}$,f($X_{13}$,$X_{23}$,V))).$\theta$

To satisfy the equation of Definition 7, $\theta$ = {$X_{11}$/0,$X_{21}$/g(0), $X_{12}$/s(0), $X_{22}$/g(s(0)), $X_{13}$/s(s(0)), $X_{23}$/g(s(s(0)))}

Let us discuss the way to find the generating terms of a given embedding term. Once the sub-unification is done, the position of the variable *V* in the embedding term is known. In the example above, *V* occurred at position [f/3, f/3, f/3], i.e. at the third argument of the first function f, the third argument of the second function f, and the third argument of the third function f. The search for generating term is as follows. The head of the position list is picked up. If the list is composed of this head only, then a generating term is found. In our case, the list is composed of f/3, and thus a generating term is built by replacing *V* for the third argument in f and new variables for the other, yielding f(X, Y, V). Afterwards, the two first elements of the list are picked up. If the list is composed of these sequence of

elements then another generating term is found. For example, if the list is composed with f/1, g/2 where f and g have two arguments, then f(g(X,V),Y) is a generating term. All sequences of elements beginning the list are considered and checked as described above.

DEFINITION 8:

A generating term $GT^V$ of depth n>0 of an embedding term $t^{V \neq V}$ is the *most specific generating term of depth n* (msgt-n) if and only if for each generating term $GT'^V$ of depth n of $t^V$, there exists a substitution $\theta$ such that $GT^V = GT'^V.\theta$

There is only one msgt-n for a given fixed depth n. To obtain it we start with any generating term of depth n. Let $GT^V$ be a generating term of depth n of $t^V$ and $\theta$ be a substitution that satisfies the equation of Definition 7. Finding the msgt-n is just a matter of replacing variables $X_i$ in $GT^V$ as follows: $X_i$ <-- $lgg(T_{i1}, T_{i2}, \ldots, T_{in})$ where $T_{i1}, T_{i2}, \ldots, T_{in}$ are instantiations of variables $X_{i1}, X_{i2}, \ldots, X_{in}$ in the substitution $\theta$. Furthermore variables $X_i$ and $X_j$ are set equal when their respective instantiations of renamed variables $T_{i1}, T_{i2}, \ldots, T_{in}$ and $T_{j1}, T_{j2}, \ldots, T_{jn}$ are identical.

Example 4:

To obtain the msgt-3 of $t^V$ from $GT^V$ in Example 3, we replace $X_1$ by $lgg(0, s(0), s(s(0))) = X_1$, and $X_2$ by $lgg(g(0), g(s(0)), g(s(s(0)))) = g(X_2)$. Furthermore $X_2$ and $X_1$ are set equal because both sequences of renamed variables $X_{11}, X_{12}, X_{13}$ and $X_{21}, X_{22}, X_{23}$ have identical instantiations i.e. $0, s(0), s(s(0))$. We then obtain the following msgt-3: $GT^V = f(X_1, g(X_1), V)$. Another msgt for the embedding term $t^V$ has depth 1: $GT^V = t^V$. In general for any embedding term $t^V$ there is always a *trivial* generating term of depth 1, $GT^V = t^V$.

We now extend the notion of generating terms to a tuple of generating terms. Let $<t_1^V, t_2^V, \ldots, t_m^V>$ be a m-tuple of embedding terms. Then a *tuple of generating terms of depth n* of $<t_1^V, t_2^V, \ldots, t_m^V>$ is a m-tuple $<GT_1^V, GT_2^V, \ldots, GT_m^V>$ such that:

$GT_i^V$ = a generating term of depth n of $t_i^V$, if $t_i^V \neq V$

= $V$, if $t_i^V = V$

for i = 1,2,...,m.

Note that all generating terms of this tuple must have the same depth. This eliminates several generating terms for an argument. If a given argument has a generating term of depth 12, it also has non-trivial generating terms of depth 6, 4, 3 and 2. Suppose that another argument in the tuple has generating terms of depth 3 and 5. For the entire tuple, we would then consider only generating terms of depth 3.

The notion of msgt-n can be extended to a tuple of generating terms. A m-tuple of generating terms is a *m-tuple of msgt's-n* if and only if $GT_i^V$ is a msgt-n of $t_i^V$, if $t_i^V \neq V$, for i = 1,2,...,m. The tuple is said to be non-*trivial* if each generating term $GT_i^V$ is non-trivial.

Example 5:

Example 2 identified five non-trivial mgsu's of L1 in L2.

L1 = append([],X X) and L2 = append([a,b,Y],[1,2],[a,b,Y,1,2])

There is only one non-trivial tuple of msgt's of a tuple of embedding terms resulting from any sub-unification. This is obtained by considering the following mgsu $<\theta, t_1^V, t_2^V, t_3^V>$ of L1 in L2:

$\theta$ = {X/[1,2]} $t_1^V$ = [a,b,Y|V] $t_2^V = V$ $t_3^V$ = [a,b,Y|V]

We have the following non-trivial 3-tuple of msgt's-3 $<GT_1^V, GT_2^V, GT_3^V>$:

$GT_1^V$ = [$X_1$|V] $GT_2^V = V$ $GT_3^V$ = [$X_2$|V]

We can check that $GT_1^V$ is a msgt-3 of $t_1^V$ with $\theta = \{X_{11}/a, X_{12}/b, X_{13}/Y\}$:

$$[X_{11}|V].\{V/[X_{12}|V]\}.\{V/[X_{13}|V]\}.\{X_{11}/a,X_{12}/b,X_{13}/Y\} =$$
$$[X_{11},X_{12},X_{13}|V]\}.\{X_{11}/a,X_{12}/b,X_{13}/Y\} = [a,b,Y|V] = t_1^V$$

$GT_1^V$ is applied to itself three times to obtain $t_1^V$. This clearly shows that $GT_3^V$ is a msgt-3 of $t_3^V$.

For brevity, henceforth in the discussion a tuple of embedding terms and a tuple of msgt's are respectively referred to simply as *embedding terms* and *msgt* . And because we are only interested in non-trivial mgsu and non-trivial msgt, we will routinely omit the word *non-trivial*[9].

From our point of view, <E1, E2> is a pair of "good" learning examples if E1 is sub-unifiable (or input sub-unifiable) in E2 and if embedding terms resulting from the sub-unification have msgt. Furthermore, we prefer msgt-n with maximum n, in order to get the deepest induced clause wrt E1 and E2, according to our criterion for induction.

[9]Trivial msgt yield useless results because they are too specific and do not carry enough information.

# 4 LOPSTER: A learning system based on sub-unification

The LOPSTER learning system is based on the sub-unification mechanism defined in the previous chapter. It uses three modes of induction: *purely recursive* , *left-recursive* and *general*. Each mode is described in its own subsection below. Another subsection explains how LOPSTER can specialize an inconsistent clause using explained negative examples. The algorithm is presented next that LOPSTER uses to determine its inductive mode according to the results of sub-unification. We then discuss the inductive bias of the system. Last, we talk about the implementation and show some experimental results. The discussion throughout focuses on how LOPSTER learns a single recursive clause.

The system is provided with some positive examples and perhaps some negative examples of the target concept. Examples are facts that may contain variables. From our point of view, `<E1,E2>` is a pair of good learning examples if `E1` is sub-unifiable or input sub-unifiable in `E2` and if embedding terms resulting from the sub-unification have msgt. Furthermore we prefer msgt-n with maximum n so as to get the deepest possible induced clause WRT `E1` and `E2`. The system has also access to a set of background definite clauses that are likely to prove useful to the learning task. A clausal definition of a predicate can be provided if the predicate is reversible WRT its definition. This means that if output arguments in the predicate are instantiated, refutation resolution proof yields all possible instantiations for input arguments and then halts. In general, pure Prolog programs are reversible; solving `append(X,Y,[a,b,c])` in this way gives all possible decompositions of the list `[a,b,c]`. Procedures that do not have a reversible mode are represented by a set of facts which are not necessarily ground, i.e. they may include universally quantified variables.

Finally, the mode declaration for each predicate is provided to distinguish inputs from outputs.

## 4.1 Modes of induction

### 4.1.1 Purely recursive mode

Although the purely recursive mode can learn only a limited class of programs, it is very efficient and effective. In this case 'efficiency' means speed and 'effectiveness' mean that any covered program can be learned easily from two weakly related examples, so long as one of them is reducible to the other, that is, one can be successively reduced to the other using the recursive clause. However, overspecialization can occur in this mode and sec. 4.1.1.1 shows cases of it. Clauses which can be learned by this mode are of the form:

$$p(A_1,A_2,...,A_n) \Leftarrow p(V_1,V_2,...,V_n).$$

where $A_1, A_2, \ldots, A_n$ are arguments of p in the head of the clause, $V_1, V_2, \ldots, V_n$ are arguments of p in the body, and either $V_i = A_i$, or $V_i$ is a variable occurring in $A_i$. Note that two arguments in the body of the clause cannot be swapped. Nevertheless, the class of programs solved by this mode is both very well-known and very common in logic programming.

Pure recursion is used when two examples E1 and E2 are found such that there exists a mgsu of E1 in E2 and there exists (non-trivial) msgt of the embedding terms resulting from the sub-unification. As sec. 2.6.2 indicates, we are first of all interested in the deepest msgt possible. We illustrate the functioning of this mode by completing Example 5 of the previous chapter, where:

L1 = append([],X,X) and L2 = append([a,b,Y],[1,2],[a,b,Y,1,2])

and a unique msgt resulted from one of the sub-unifications in Example 2, i.e. msgt-3 $<GT_1^V, GT_2^V, GT_3^V>$:

$$GT_1^V = [X_1|V] \qquad GT_2^V = V \qquad GT_3^V = [X_2|V]$$

The recursive clause is built from this msgt as follows: $V$ is renamed to $V_i$ in each $GT_i^V$. If $GT_i^V \neq V_i$, $GT_i^V$ becomes the $i^{th}$ argument in the literal of the head of the clause and $V_i$ becomes the $i^{th}$ argument in the literal in the body of the clause. If $GT_i^V = V$ [10], then the $i^{th}$ argument in the head and the body becomes the LGG of the $i^{th}$ arguments of `L1` and `L2`. In our example, the 2nd argument in the head and in the body is `lgg(X,[1,2])` = $V_2$. These substitutions produce the following purely recursive clause

$$append([X_1|V_1],V_2,[X_2|V_3]) \Leftarrow append(V_1,V_2,V_3)$$

which can reduce `E2` to `E1` in three steps of recursive deduction. The clause is then specified by setting $X_1$ and $X_2$ equal because they share the same `a,b` sequence of instantiations during the proof. The following definition for `append` is proposed:

append([],X,X).

$append([X_1|V_1],V_2,[X_1|V_3]) \Leftarrow append(V_1,V_2,V_3)$.

LOPSTER accepts the clause because it is consistent. LOPSTER tries to specialize an inconsistent clause using explained negative examples (sec. 4.2). If the attempt at specialization fails, the clause is rejected and a new one built using the next deepest msgt, if any. This process continues until a satisfactory clause is induced or all msgt are exhausted.

Given two appropriate examples, this mode of induction can learn other purely recursive programs just as easily as it learned `append`:

- `member/2` (list membership function);

[10] this means the sub-unifier is also an unifier for this argument.

- `plus/3` (add two numbers represented with the `successor` function);
- `delete/3` (delete the first occurrence of an element in a list);
- `extract`$^{N}$`/3` (extract from a list the $N^{th}$ element, N is represented with the `successor` function);
- `last/2` (extract the last element of a list);

...

Many logic programs can be rewritten in a purely recursive format using functional notation to convert literals into terms. The process can be called "unflattening" or "inverse flattening". It permits LOPSTER to learn more complex programs such as `factorial/2` and `reverse/2` whose examples use functions to represent usual relations. Thus, from the following examples

E1 = factorial($s^1$,$s^1$)

E2 = factorial($s^3$,$s^3$*($s^2$*$s^1$))

LOPSTER induces this clause:

factorial(s(A),s(A)*B) ⇐ factorial(A,B).

If the example `E2` were provided in clausal form:

factorial($s^3$,$s^6$) ⇐ *($s^3$,$s^2$,$s^6$),*($s^2$,$s^1$,$s^2$).

it would have been unflattened to get `E2`. Once the clause had been induced using `E1` and `E2` above, the '*' function would have been brought into the relational form from the induced clause. The unflattening and flattening processes allow us to deal with clauses and in certain sense enlarge the coverage.

#### 4.1.1.1 Requirements for exact learning

We cannot guarantee that LOPSTER will learn the exact recursive clause given two examples, one of which is reducible to the other. The system induces, in some cases, overspecialized clauses. What will ensure that exact learning takes place? An analysis of two cases shows that overspecialization happens when examples are too similar. To ensure that exact learning takes place, additional examples not covered by the overspecialized clause must also be considered.

The first case of overspecialization occurs when the sub-unifier is also a unifier for an argument. In this situation the LGG which LOPSTER computes for the terms of corresponding arguments in the examples may not be general enough. Consider these examples of append:

append([],[d,e],[d,e]).
append([a,b,c],[d,e],[a,b,c,d,e]).

Since `lgg([d,e],[d,e]) = [d,e]`, purely recursive mode (see Appendix A) produces the clause:

append([A|B],[d,e],[A|C]) ⇐ append(B,[d,e],C).

This clause is too specific. The problem can be solved by considering an example of append which the clause does not cover, such as `append([1,2],[],[1,2])`. The LGG of the second arguments of the clause literal and the new example is `lgg([d,e],[]) = D`, where `D` is a new variable. Replacing D for the second argument in the head and the body of the induced clause yields the usual recursive clause of append.

There is another problem, however. The base case is also overspecialized. It cannot prove the new example because they do not use the same constants. One solution is to try to

prove the new example by refutation on the induced clause alone, reducing it as far as possible and keeping the last unsolved goal. The unsolved goal, in our case, would be `append([],[],[])`. The LGG of this goal and the initial base case then becomes the new base case, i.e. `lgg(append([],[d,e],[d,e]),append([],[],[])) = append([],L,L)`. This completes the definition for append.

A second kind of overspecialization is illustrated by the following induction. From the examples:

append([],L,L).
append([a,a,a],[b,c],[a,a,a,b,c]).

LOPSTER induces the recursive clause (see Appendix A):

append([a|A],B,[a|C]) ⇐ append(A,B,C).

which is overspecialized. To overcome this, the program simply needs to consider an uncovered example such as `append([d,d],[e],[d,d,e])`. Repeating the same induction with the base case gives:

append([d|A],B,[d|C]) ⇐ append(A,B,C).

Taking the LGG of the two induced clauses produces the usual recursive clause for append.

We strongly believe that the purely recursive mode of LOPSTER ensures exact learning if neither kind of similarity between the two training examples or inside an example occurs.

### 4.1.2 Left-recursive mode

Left-recursive mode covers a larger class of programs than does purely recursive mode. Let $p(A_1, A_2, \ldots, A_m, A_{m+1}, \ldots, A_n)$ be a literal with $A_1, A_2, \ldots, A_m$ as inputs and $A_{m+1}, A_{m+2}, \ldots, A_n$ as outputs. Clauses of the form:

$$p(A_1,A_2,\ldots,A_m,V_{m+1},\ldots,V_n) \Leftarrow$$
$$p(V_1,V_2,\ldots,V_m,W_{m+1},\ldots,W_n),L_1,L_2,\ldots,L_k.$$

are covered by this mode. $V_{m+1}, V_{m+2}, \ldots, V_n$ and $W_{m+1}, \ldots, W_n$ are variables. $A_1, A_2, \ldots, A_n$ are input arguments of p in the head of the clause. $V_1, V_2, \ldots, V_m$ are input arguments of p in the body, and either $V_i = A_i$, or $V_i$ is a variable occurring in $A_i$. The $L_i$'s are literals.

This inductive mode is used when two examples E1 and E2 are found such that E1 is input sub-unifiable in E2 and there exists msgt of embedding terms resulting from the input sub-unification. Once again we are interested in the deepest msgt, and if it does not produce an accepted clause induction is repeated with the second deepest msgt and so on.

Let us illustrate the operation of left-recursive mode by watching how it learns the procedure `multiply/3`. Assume the usual definition of `plus/3` is in the background knowledge. The definition can be clausal because `plus` is reversible. For clarity, $s^n$ is used as an abbreviation for `s(s(...(0)...))`, which is n times the `successor` function and a representation for the number n.

Let E1 = multiply(X,0,0) and E2 = multiply($s^2$,$s^3$,$s^6$)

Only this i-mgsu of E1 in E2 $\langle\theta, t_1{}^V, t_2{}^V\rangle$:

$\theta = \{X/s^2\}$ $\quad t_1{}^V = V$ $\quad t_2{}^V = s(s(s(V)))$

has msgt $<GT_1^V, GT_2^V>$: $GT_1^V = V$ $GT_2^V = s(V)$

which is depth 3. Left-recursive mode first induces the following overgeneral clause:

multiply($V_1$,s($V_2$),$V_3$) ⇐ multiply($V_1$,$V_2$,$W_3$).

Except for considering only input arguments and introducing a new variable for each output argument, the first part of left-recursive mode is the same as purely recursive mode.

The left-recursive mode process must now bind the unmatched output variable $V_3$ (instantiated to $s^6$ in E2) by adding a literal which has $s^6$ as output to the body of the clause. It finds plus($s^2$, $s^4$, $s^6$) amongst other literals by solving the goal plus(X, Y, $s^6$) with Prolog. This literal seems a good choice for two reasons. $V_1$ is already instantiated to $s^2$ in E2, so it makes sense to use a literal involving $s^2$. Moreover, adding this literal introduces $s^2$ into the clause and makes it unnecessary to add another literal with an $s^2$ output. This is one of our heuristics. At this point all that remains to be done is to produce the term $s^4$ by a literal. Left-recursive mode binds the unmatched variable $W_3$ to it, obtaining the clause:

multiply($V_1$,s($V_2$),$V_3$) ⇐ multiply($V_1$,$V_2$,$W_3$),
plus($V_1$,$W_3$,$V_3$).

To check the validity of this clause, E2 must be proven by refutation resolution with the proposed clause, E1 and the background knowledge clauses. The proof must use the proposed clause three times, the depth established by the generating terms. In this case proof succeeds and the system accepts the clause because it is consistent, not covering any negative example. Were it not consistent, LOPSTER would try to specialize the clause using the method described in sec. 4.2. If this cannot be done or if the training example E2 cannot be proved, LOPSTER unbinds $W_3$, adds a new literal producing the output $s^4$, and recommences using the input arguments of this new literal. Whenever a single input remains unbound the program tries to bind it with $W_3$, and so on.

Because the system backtracks on literals added in the head of the clause, their number must be bound to keep processing tractable. Sec. 4.4 presents a way to keep this number small.

### 4.1.3 General mode

Certain recursive programs have at least one literal containing output arguments to the left of the recursive call in the body of the clause. This means an output argument of the literal is used as input in the recursive call. An example is the recursive clause `p(X,f(Z))` ⇐ `q(X,Y),p(Y,Z)` where in both p and q the first argument is the input and the second argument is the output. The two previous modes do not handle this case because sub-unification cannot be applied to the arguments of examples which are modified by a literal. Purely recursive and left-recursive modes are also incapable of handling a recursive clause which has multiple recursive calls in its body.

If neither mode applies, the target program may incorporate the sort of recursion discussed above or it not be recursive at all. The general mode of induction is meant do deal with such cases, but it only performs θ-subsumption generalizations and cannot deal with the notion of generalization based on implication, as the other two modes can. To solve this class of programs, we do not have a better method than existing systems. This is why we propose to fall back on the general methods described in Muggleton & Feng [1990] and Quinlan [1990] in the general mode.

## 4.2 Specializing an inconsistent clause

This subsection describes a general way to specialize an inconsistent clause using explained negative examples. It is used in the purely recursive and left-recursive modes when

an inconsistent clause is built, enlarging the class of programs covered by these modes. Certain recursive programs involving a single recursive call, all literals to the left of the recursive call in the body of the clause contain input arguments only. Such programs are representative of programming schemata in which certain properties of input are verified prior to the recursive call. LOPSTER's objective is to add such literals to the clause it has built in order to eliminate the inconsistency.

The method relies on explained negative examples which are assumed to be expressed in terms of known concepts whose definitions are in the background knowledge. Given the following negative example of the SET UNION concept, `union([a,b,c],[c,d,b],[a,b,c,c,d,b])`, an explanation might be `member(b,[c,d,b])` or `member(c,[c,d,b])`. An explanation of a negative example is considered useful if the negation[11] of this explanation causes the failure of the proof of this negative example. This removes the inconsistency associated with that negative example. Note that any explanation can only be expressed in terms of inputs.

The negation of an explanation is called a condition. The method used to add a condition to a clause restricts the arguments of the condition to variables occurring in input arguments in the head of the clause. It creates a condition by first identifying a provable negative example that has an explanation. This explanation is then negated and generalized by replacing its terms with variables, keeping track of terms which were arguments of the explanation. Essentially LOPSTER re-proves the negative example and tries to add the condition somewhere in the proof in such a way that its variables can be bound to variables already used in input arguments.

[11] under the closed-world assumption.

The following simple example illustrates the method well. The target concept checks whether a list of elements is a set, i.e. that it contains no duplicates. Suppose the purely recursive mode builds this program:

no_duplicate([]).

no_duplicate([H|T]) ⇐ no_duplicate(T).

This clause is inconsistent. Now assume we have the following negative example with its explanation:

no_duplicate([a,b,c,d,c,e])
explanation: member(c,[d,c,e])

LOPSTER tries to add the condition ¬`member(X,Y)` to the clause, with the variable `X` and Y is instantiated to `c` and `[d,c,e]` respectively. It then re-proves this negative example step by step until the variables `X` and `Y` can be bound to existing input variables.

STEP 1:

no_duplicate([a|[b,c,d,c,e]]) ⇐ no_duplicate([b,c,d,c,e]).

`H` is instantiated to `a` and `T` is instantiated to `[b,c,d,c,e]`. The variables `X` and `Y` in the ¬`member(X,Y)` condition cannot be bound to H and T because they have a different instantiation. LOPSTER continues to step 2 of the proof.

STEP 2:

no_duplicate([b|[c,d,c,e]]) ⇐ no_duplicate([c,d,c,e]).

`H` is now instantiated to `b` and `T` is instantiated to `[c,d,c,e]`. A binding to `X` and `Y` is rejected for the same reason as before and proving continues:

STEP 3:

no_duplicate([c|[d,c,e]]) ⇐ no_duplicate([d,c,e]).

H is instantiated to c and T to [d, c, e]. X and H now have the same instantiation and so do Y and T. Since both variables can be bound, we propose the following specialization for the initial recursive clause:

no_duplicate([H|T]) ⇐ ¬member(H,T), no_duplicate(T).

Negative examples can have more than one explanation. The user may provide unhelpful explanations which are of no use in the process of justifying the negative example with the existing clauses. In this case LOPSTER tries each explanation until it finds one whose condition can be added. If no condition can be the system considers the explanations of the next negative example, if any. Conditions are successively added to a clause until it becomes consistent; the ultimate recursive clause may have several.

The UNION concept provides a more complex example. The task is to specialize its two clauses:

1) union([H|T],L,U) ⇐ union(T,L,U).

2) union([H|T],L,[H|U]) ⇐ union(T,L,U).

Note that these clauses can be obtained by the purely recursive mode from three appropriately-chosen examples:

union([],L,L)
union([a,b],[d,b,c,a],[d,b,c,a])
union([a,b],[c,d],[a,b,c,d])

The definition as it stands is overgeneral and requires conditions be added to make it consistent. Consider the following provable negative example and its two disjunct explanations:

union([a,b,c],[d,c,a,e],[c,d,c,a,e])
explanation 1: ¬member(b,[d,c,a,e])
explanation 2: member(c,[d,c,a,e])

LOPSTER first tries to use explanation 1 to add the condition `member(X,Y)` to one of the two clauses above with `X` instantiated to `b` and `Y` to `[d,c,a,e]`. This negative example is proved by two successive applications of clause 1, clause 2, and the base case, i.e. `union([],L,L)`. The second step of this proof produces the following instance of the clause 1:

union([b|[c]],[d,c,a,e],[c,d,c,a,e]) ⇐ union([c],[d,c,a,e],[c,d,c,a,e]).

`H` is instantiated to `b`, `T` to `[c]`, and `L` to `[d,c,a,e]`. This leaves `H` and `L` with the same respective instantiations as `X` and `Y` in the condition `member(X,Y)`. LOPSTER keeps these bindings and proposes the following specialization for clause 1:

3) union([H|T],L,U) ⇐ member(H,L), union(T,L,U).

The negative example is no longer covered by the induced logic program (clauses 2 and 3, and the base case). However the following negative example is still covered by an application of clause 2 and the base case in the induced program:

union([a],[b,a],[a,b,a])

What can we do to overcome this inconsistency ? We can use the unused explanation 2 (`member(c,[d,c,a,e])`) of the first negative example. Thus, we try to add the condition ¬`member(X,Y)` in the clause 2 (since the second negative example only uses the clause 2),

where X is instantiated to c, and Y is instantiated to [d,c,a,e]. At the third step of the proof of the first negative example (with the initial program, i.e. using clause 1 instead of clause 3) the following instance of the clause 2 is produced:

union([c|[]],[d,c,a,e],[c|[d,c,a,e]]) ⇐ union([],[d,c,a,e],[d,c,a,e]).

H is instantiated to c, T to [] and L to [d,c,a,e]. H and T once again have the same respective instantiations as X and Y in the ¬member(X,Y) condition. Clause 2 can now be specialized as:

4) union([H|T],L,[H|U]) ⇐ ¬member(H,L), union(T,L,U).

The second negative example is no longer covered and the UNION concept is completely and consistently defined:

union([],L,L).

union([H|T],L,U) ⇐ member(H,L), union(T,L,U).

union([H|T],L,[H|U]) ⇐ ¬member(H,L), union(T,L,U).

## 4.3 Algorithm

We are now ready to describe the algorithm that selects modes to be executed and their order of execution. Let N be the number of positive examples; there are then $O(N^2)$ ordered pairs of distinct examples. This algorithm below learns a single clause consistent with the examples:

1) Find a pair of distinct examples <E1,E2> such that E1 is input sub-unifiable in E2 and there is msgt of embedding terms results from the input sub-unification:

If the input sub-unification can be extended to a sub-unification and there is msgt of embedding terms results from the sub-unification, then induce a clause with *purely recursive mode*. Otherwise, induce a clause with *left-recursive mode*.

2) If no pair of examples has succeeded in 1, then induce with *general mode*.

The algorithm backtracks to try left-recursive mode if purely recursive mode fails, and if a pair of examples fails it tries another pair.

## 4.4 Bias

Left-recursive mode (described in sec. 4.1.2) relies on some preference criteria ("biases") to prune the search space of literals added to the induced clause or heuristically speed up the search process:

1) *depth of an output argument*: the depth of an output argument in the body of the clause that also appears in the head is 0; the depth of an output argument in the body is one plus the maximum of depths of the output arguments in the literal in which it occurs as input. Otherwise, the depth of an output argument is 0. This is similar to the notion of depth used in ij-determination [Muggleton & Feng, 1990a]. Here is an example.

Given the mode declaration `p(+,-),q1(+,+,-),q2(+,-),q3(+,-)`[12] for the following clause:

$p(s(V_1),V_2) \Leftarrow p(V_1,W), q3(W,I_2), q2(s(V_1),I_1), q1(I_1,I_2,V_2).$

$V_2$ has depth 0; $I_1$ and $I_2$ have depth 1; and W has depth 2.

[12] "+" means an input argument and "-" means an output argument.

Imposing a maximum depth on output arguments will prune the search space. We suggest 2 or 3 as reasonable numbers.

2) Background clauses can be ordered so as to speed the search up. Those more likely to be useful should be tried first when searching for a literal to be added in a clause. For instance the clauses for `append` could be tried first, since they are used so frequently in logic programming.

## 4.5 Implementation and experimental results

### 4.5.1 Implementation

The purely recursive and left-recursive modes of LOPSTER have been implemented in Quintus Prolog (see appendices for demonstrations and more detail). The implementations induce all possible clauses of any depth. The general mode and the specialization process based on explained negative examples have not been implemented.

### 4.5.2 Comparison of purely recursive mode with two existing systems

We have compared the purely recursive mode of LOPSTER with two ILP systems available to us, GOLEM [Muggleton & Feng, 1990] and a version of FOIL using determinate literals [Quinlan, 1991a]. The experiments summarized in Table 4.1 below were performed on a SPARCstation1. The reported times are in seconds and the number of positive examples is indicated. The examples used by LOPSTER and the induced clauses for three problems follow:

LIST MEMBERSHIP:

Examples are:

member(X,[X|Y])
member(4,[1,2,3,4,5])

There is one induced clause of depth 3:

member(A,[B|C]) ⇐ member(A,C).

APPEND TWO LISTS:

Examples are:

append([],X,X)
append([a,b,Y],[1,2],[a,b,Y,1,2])

There is one induced clause of depth 3:

append([A|B],C,[A|D]) ⇐ append(A,C,D).

FACTORIAL:

Examples are:

fact($s^1$,$s^1$)
fact($s^3$,$s^3$*($s^2$*$s^1$))

A first induced clause of depth 2 is:

fact(s(A),s(A)*B) ⇐ fact(A,B).

An alternative clause is also proposed:

fact(s(A),s(s(B))*C) ⇐ fact(A,C).

This clause makes no sense because the output argument `s(s(B))*C` contains the unbound variable `B`. It is rejected and only the purely recursive definition of `fact` is kept.

Table 4.1: Comparison of ILP systems.

| | | member | append | factorial |
|---|---|---|---|---|
| GOLEM | time | 2.4 | 16 | 0.78 |
| | #pos. ex. | 15 | 49 | 4 |
| FOIL | time | 0.30 | 225 | n/a[13] |
| | #pos. ex. | 7 | 161 | |
| LOPSTER | time | 0.016 | 0.067 | 0.034 |
| | #pos. ex. | 2 | 2 | 2 |

The experiment provided each system with the minimal set of examples adequate for the learning algorithm it implements. In the experiment described here GOLEM has been given a subset of the h-easy model as its training set. GOLEM should learn the concept of `append` with a smaller training set constituting of the union of two basic representative sets. However, it is not clear how to obtain basic representative set without knowing the clause being induced. LOPSTER users do not need specialized ILP concept knowledge to design a minimal set of examples. It suffices simply for them to know two examples that belong to the same chain of recursive calls or to provide the base case.

[13]There were insufficient examples to learn `factorial` using FOIL.

LOPSTER searches on terms while the other systems search on literals. Its times would increase slightly, but not significantly, if searching occurred before identifying a pair of examples for which a clause can be induced. The performance gap between the systems is partially due to the design goals of different systems. GOLEM and FOIL are meant to be general ILP systems for a wide range of applications which are capable of inducing such complex programs as `quicksort`. LOPSTER targets a narrower, but useful, class of programs.

### 4.5.3 Results with left-recursive mode

Experiments have also been performed with the left-recursive mode of LOPSTER. Here is an explanation for each problem we have considered:

LIST REVERSAL:

Positive examples are:

reverse([],[])

reverse([a,b,c],[c,b,a]).

The background knowledge contains the clausal definition of the concept ADD AN ELEMENT TO THE END OF A LIST. There is a single induced clause of depth 3:

reverse([A|B],C) ⇐ reverse(B,D),add_to_end(D,A,C).

FACTORIAL:

Positive examples are:

fact($s^1$,$s^1$)

fact($s^3$,$s^6$).

The background knowledge contains ground facts of the MULTIPLY concept. There are two equivalent induced clause of depth 2:

fact(s(A),B) ⇐ fact(A,C),multiply(C,s(A),B).

fact(s(A),B) ⇐ fact(A,C),multiply(s(A),C,B).

INSERTION SORT:

Positive examples are:

isort([],[])
isort([5,2,4,1,3,6], [1,2,3,4,5,6]).

The background knowledge contains the clausal definition of the concept INSERT AN ELEMENT IN A SORTED LIST. Three clauses are induced when the parameter *maximal depth of output argument* is set to 2. The first induced clause of depth 6 is:

isort([A|B],C) ⇐ isort(B,D),insert(A,D,C).

Two other clauses of depth 3 are equivalent to:

isort([A,B|C],D) ⇐ isort(C,E),insert(A,E,F),insert(B,F,D).

This clause inserts two numbers rather than one at a time into a sorted list. Setting the parameter to 3 results in LOPSTER proposing six other clauses. All are equivalent to:

isort([A,B,C|D],E) ⇐ isort(D,F),insert(A,F,G),insert(B,G,H),insert(C,H,E).

This clause inserts three numbers at a time in a sorted list. `isort` illustrates clearly that for each generating term of depth `n`, there is a generating term of depth `m`, for every divisor `m` of `n`.

MULTIPLY:

Positive examples are

time(0,X,X)

time($s^3$,$s^2$,$s^6$).

The background knowledge contains the clausal definition of the concept ADD TWO NUMBERS. Two equivalent clauses of depth 3 are induced. One is:

time(s(A),B,C) ⇐ time(A,B,D),plus(B,D,C).

Three alternative clauses[14] are proposed that can prove the example. All are very incomplete and inconsistent and can be eliminated easily.

Table 4.2 shows the results. Times are in seconds. Experiments were performed with the system parameter *maximal depth of output arguments* set to 2 and 3. As expected, times increase with increases in this parameter because it expands the search space of literals that can be added to the clause. Nevertheless, performance is quite reasonable for both values.

Increasing the value of this parameter allows induction of more complex clauses. When set to 2 the clause in the INSERTION SORT problem that inserts three numbers at a time in a sorted list cannot be obtained because the output variable F has depth 3. A reasonable course of action therefore is to run LOPSTER with the parameter set to 2, and if a satisfactory result is not obtained, to increase it to 3 and so on until a satisfactory clause is induced or running time becomes excessive.

[14]In fact seven clauses are proposed, but several of them are equivalent.

Table 4.2: Experimental results of left-recursive mode.

| *Maximal depth* | *2* | *3* |
|---|---|---|
| list inversion | 0.03 | 0.05 |
| factorial | 0.25 | 0.40 |
| insertion sort | 0.5 | 0.73 |
| multiply | 2.4 | 7.5 |

# 5 Extensions of the method

The learning method used in LOPSTER can be extended in two ways. The first defines a more general notion of sub-unification than the one already presented. The second can induce several clauses simultaneously by considering several generating terms for the same argument. The first extension accepts examples that are less correlated while the second extension enlarges the class of programs that LOPSTER can induce. The extensions can be combined.

The presentation relies on examples to show the main ideas employed by these extensions rather than giving an explicit presentation of the underlying theory. In part this is because the extensions have not been developed extensively yet. One remaining problem has to do with the number of induced clauses in the extended approach; we have not yet investigated good heuristics to prune the search space.

## 5.1 Generalizing sub-unification

There exists a natural extension to the notion of sub-unification presented earlier. Generalized sub-unification can be expressed as follows: terms `t1` and `t2` are said to be sub-unifiable if there exists a subterm `s1` of `t1` and a subterm `s2` of `t2` such that `s1` and `s2` are unifiable. This is a generalization of the kind of sub-unification used in our system which restricts `s1` to be in `t1`. An example will illustrate:

Let t1 = f(a,f(c,s(a,Y))
and t2 = f(b,s(X,b))

`t1` and `t2` are sub-unifiable because the subterm `s(a,Y)` of `t1` is unifiable with the subterm `s(X,b)` of `t2`. The embedding term in `t1` is `f(a,f(c,V)`, and in `t2` it is `f(b,V)`.

The benefit of extended sub-unification is that it no longer requires one of the examples to be reducible to the other. For the extended form to succeed, two examples must simply use a same instance or two unifiable instances somewhere in the proof. Consider the following example of the APPEND TWO LISTS concept:

Let E1 = append([a,b],L,[a,b|L])
and E2 = append([c,d,e],[f,g],[c,d,e,f,g])

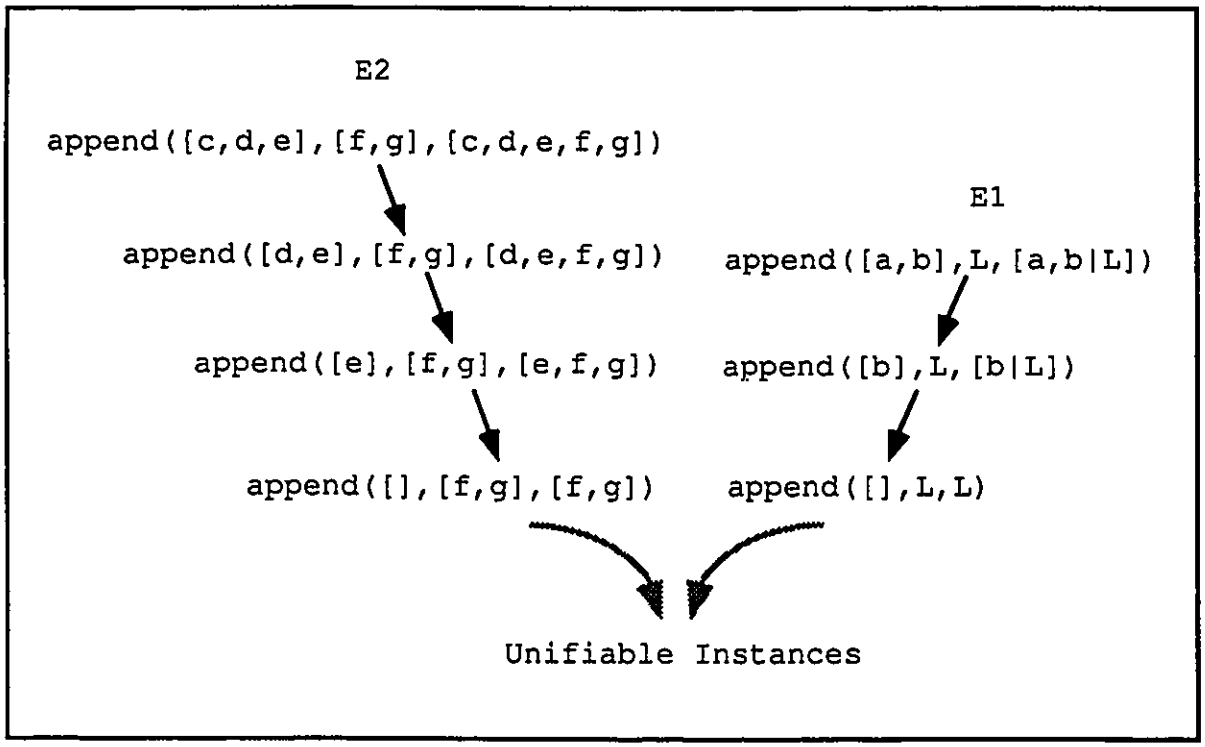


Figure 5.1: Two examples with unifiable instances.

Fig. 5.1 shows how these two examples are reduced to two unifiable instances; an arrow indicates a reduction by the recursive clause. The recursive clause is induced from these two examples in the same way as in the purely recursive mode, with two exceptions. One change has to do with sub-unification. It now involves embedding terms in both examples rather than just one. The other difference concerns generating terms, which must now be the same for both embedding terms for a given argument. To see the impact of these changes consider fig. 5.1 again. `[]` (the empty list) is the unifiable subterm for the first arguments of `E1` and `E2`. For the second arguments `L` is unifiable with `[f,g]`. For the third arguments, `L`, now instantiated to `[f,g]`, is unifiable with the subterm `[f,g]` of `[c,d,e,f,g]` in `E1`. The embedding terms are:

E1: [a,b|*V*], *V*, [a,b|*V*]
E2: [c,d,e|*V*], *V*, [c,d,e|*V*]

Generating terms can then be found for each argument except the second argument for which a unification has occured. `[X, V]` is the generating term for the first argument and `[Y, V]` for the third. The induced clause is built in exactly the same way as it is done in the purely recursive mode:

$$\text{append}([X|V_1],V_2,[X|V_3]) \Leftarrow \text{append}(V_1,V_2,V_3).$$

where `X` and `Y` have been set equal because both are instantiated to `a,b` in `E1` and to `c,d,e` in `E2`.

One task remains. Examples cannot be proven using the induced clause alone; a base case is needed as well. To produce a base case from the materials at hand, the LGG is computed for each two subterms that have been unified for a given argument. This value replaces the argument in the base case. In the preceding example, `lgg([],[]) = []`, `lgg(L,[f,g]) = L`, and `lgg(L,[f,g]) = L`. Thus the base case is:

append([],L,L)

A shortcoming of extended sub-unification is that it tends to induce overspecific programs in the absence of a general base case. This is less a problem than a consequence of the sort of examples supplied to the method; we do not want to proceed if two examples do not suggest a more general program. To overcome this we must consider additional examples (as has been done in sec. 4.1.1.1). The following proof of MEMBER illustrates this deficiency. Consider these two examples:

E1 = member(a,[1,2,a,b])

E2 = member(a,[f,e,d,c,a,b])

LOPSTER proposes the following logic program:

member(a,[a,b]).

member(a,[X|V]) ⇐ member(a,V).

This program is very specific. To generalize it more examples must be considered.

## 5.2 Inducing several recursive clauses simultaneously

A second extension to the learning system involves considering more than one generating term for the same argument if possible, i.e. inducing several recursive clauses. If `f(a,g(g(f(b,0))))` is an embedding term, then `{f(X,V),g(V)}` is a set of generating terms of it. The example below shows how three recursive clauses are induced simultaneously[15]:

Let E1 = p(s(X))

and E2 = p(f(f(g(f(h(h(g(s(0)))))))))

[15]The first definition of sub-unification is used in this example.

`s(X)` in `E1` is unified with the subterm `s(0)` of `E2`. The embedding term is `f(f(g(f(h(h(g(V)))))))`. `f(V)`, `g(V)` and `h(h(V))` are generating terms of this embedding term. The following logic program is then proposed:

p(s(X)).

p(f(V)) ⇐ p(V).

p(g(V)) ⇐ p(V).

p(h(h(V))) ⇐ p(V).

Another solution can be obtained by considering the following set of generating terms: `f(V)`, `g(V)`, `h(V)`.

Many times the set of generating terms can contain duplicates. For instance, the `cons` list function is a generating term of the two clauses for the first argument of the SET UNION concept, i.e. the two clauses of `union` have the same generating term for their first argument. This situation requires attention because it significantly increases the number of sets of generating terms that have to be considered.

## 5.3 An example of combined extensions

The two extensions we have defined—generalized sub-unification and induction of multiple clauses—can be combined harmoniously. The following fictitious case of induction illustrates how.

Let E1 = p(g(f(f(g(h(a,Y))))))
and E2 = p(f(g(g(f(h(X,b))))))

`h(a,Y)` is a subterm of `E1` unifiable with the subterm `h(X,b)` in `E2`. The embedding term in `E1` is `g(f(f(g(V))))` and in `E2`, `f(g(g(f(V))))`. There is a unique choice for the

set of generating terms:. `f(V), g(V). lgg(h(a,Y), (h(X,b)) = h(X,Y)` is also needed. The following logic program is then induced:

p(h(X,Y)).

p(f(V)) ⇐ p(V).

p(g(V)) ⇐ p(V).

## 5.4 The search space

These extensions increase the size of the search space considerably. Extended sub-unification involves many more sub-unifiers than the original notion of sub-unification, and for a given embedding term the number of sets of generating terms can be quite large. To use these extensions efficiently we need good heuristics for pruning. Identifying these remains to be done.

# 6 Application in necessary construction

This chapter discusses a single issue in necessary constructive induction, the manner in which the definition of a new or invented predicate is induced. A distinction must first be drawn between *necessary* and *useful* constructions. The invention of a predicate is *necessary* if the concept is not definable[16] without such a new predicate. The invention of a predicate is *useful* if it does not affect the learnability of the concept, i.e. if the concept is definable without this new predicate but the invention eases the learning task. Our interest is in necessary construction which is a real challenge. Ling [1991b] gives an extended and interesting study on this.

Almost all current systems that perform construction invent useful predicates only [Silverstein & Pazzani, 1991]. The invented predicate is often defined as a conjunction of literals without recursion. When a predicate is defined in this way it can be eliminated simply by substituting the conjunction of literals for each of its occurrences.

Once a new necessary predicate has been invented its definition must still be induced. This definition must be recursive if the predicate is necessary. Otherwise the conjunction of literals that define the new predicate could be substituted for it without affecting the learnability of the given concept, i.e. the predicate is not necessary. This consequence has significance for our learning system, which specifically deals with recursive concepts. Furthermore, a chronic difficulty in defining new predicates is the lack of sufficient data (sec. 6.1). LOPSTER tends to be more adequate than existing systems in generalizing instances of

[16]By definable, we mean representable by a finite numbers of definite clauses.

a new predicate because, in many cases, it does not suffer from lack of data. Let us see how well LOPSTER does in this regard.

## 6.1 Collected instances of the new predicate

Once a predicate has been invented its instances are collected.[17] Because the target concept for which the predicate has been invented may not use all of the instances of the new predicate, the learning system may not collect many or most of the instances (see the example in the next subsection). There are two problems associated to this situation: *partial utilization* and *zero utilization*, which can be characterized in terms of basic representative sets. Recall that a basic representative set is obtained from each instance of a clause. A new predicate is *partially utilized* if there is at least one (not all) basic representative set of the clause defining the new predicate among the collected instances, and it is *zero utilized* if there is no basic representative set among the collected instances. The second problem is more serious since existing systems will not induce the definition of the new predicate without the appropriate basic representative set. Both problems are studied in following subsections. We demonstrate cases in which LOPSTER can induce the definition of a new predicate when existing systems cannot.

## 6.2 Example of partial utilization

Learning the `multiply` concept without background knowledge is a good example of partial utilization. A new predicate, such as `plus`, must be invented. `plus` is a necessary predicate, because without it `multiply` cannot be represented by a finite number of clauses.

[17] We do not discuss the way the instances are collected.

Note that an infinite number of clauses could do the job; each clause would represent a particular instance of multiplication. Multiplication by 3 would be represented by :

multiply(s(s(s(0))),s(X),s(s(s(Y)))) ⇐ multiply(s(s(s(0))),X,Y).

It is clear that in practice this is unsatisfactory and that learning the definition of `multiply` requires invention of a new predicate. At the successful conclusion of learning, the definition of `multiply` using `plus` is:

multiply(X,s(Y),Z) ⇐ multiply(X,Y,W), plus(W,X,Z).

This clause does not use all possible instances of `plus`. In fact, the learning system will only collect instances of the form `plus(ab,a,ab+a)` for all integers a and b. It will never see `plus(3,2,5)`, `plus(1,2,3)` or `plus(2,4,6)`. One question arises—is there any basic representative set of the recursive clause of `plus` among the examples used to learn `multiply` ? Yes, but only when a in `plus(ab,a,ab+a)` is 1. Thus, `plus(3,1,4)` and `plus(2,1,3)` form a basic representative set of the clause:

plus(s(X),Y,s(Z)) ⇐ plus(X,Y,Z).

So, the only examples that existing systems can use are too specific—the second argument of `plus` is always 1—for them to induce the general definition of `plus`. Thus the examples of `plus` collected from `multiply`'s basic representative sets are not sufficient for systems such as GOLEM to learn its definition. The situation is better with LOPSTER. In its purely recursive mode (see Appendix A) the examples `plus(6,2,8)` and `plus(2,2,4)` yield:

plus(s(X),s(s(0)),s(Y)) ⇐ plus(X,s(s(0)),Y).

If we consider another example, `plus(s(s(s(0))),s(0),s(s(s(s(0)))))`, then the second argument can be replaced by `lgg(s(s(0)),s(0)) = s(Z)`, yielding the generalized clause:

plus(s(X),s(Z),s(Y)) ⇐ plus(X,s(Z),Y).

This clause is almost complete WRT to the collected instances, leaving only `plus(0,0,0)` uncovered. This instance would be included in the base case, whose method of construction is shown in sec. 4.1.1.1.

## 6.3 Example of zero utilization

Here is a function[18] that zero utilizes the `double` function which multiplies a number by 2. Let us consider the following function:

$f(X) = 2^X$ for every integer $X \geq 1$

This function can be defined recursively in mathematical form as follows:

$f(1) = 2$

$f(X) = f(X-1)*2$

It can be defined in a relational form using `double`:

$f(s^1,s^2)$.

f(s(X),Y) ⇐ f(X,Z),double(Z,Y).

`f` uses instances of `double` of the form `double(`$s^{2^{n-1}}$`,` $s^{2^n}$`)` for every integer $n \geq 2$. More explicitly these are:

double($s^2$,$s^4$)

double($s^4$,$s^8$)

double($s^8$,$s^{16}$)

[18]employing one variable whose range and domain is the positive integers.

double($s^{16}$,$s^{32}$)

...

The clausal definition of `double` to be considered is[19]:

double($s^2$,$s^4$).

double(s(X),s(s(Y))) ⇐ double(X,Y).

It is easy to see that no basic representative set of the recursive clause of `double` can be collected by an observation of input/output behavior of `f`. In fact, the difference between the first arguments of any two instances is always greater than or equal to 2. A basic representative set of `double` must contain two instances representing consecutive integer numbers in the first argument. For this reason LOPSTER succeeds where existing systems fail to induce the recursive clause of `double`. Any two instances provided to it will lead to the induction of the recursive clause of `double` because all instances belong to the same recursive chain. Operating in its purely recursive mode, LOPSTER induces the recursive clause from the following examples:

double(4,8)
double(16,32)

The depth of the clause induced WRT to these examples is 12.

Now consider the following family of functions for every n≥2:

$f_n(X) = n^X$ for every integer X≥1

These functions can be represented in the relational form:

[19]Note that if we consider the following clause `double(s(s(X)),s(s(s(s(Y)))))` ⇐ `double(X,Y)`, which doubles only even integers if the base case is `double(2,4)`, then the instances `double(2,4)` and `double(4,8)` form a basic representative set of this clause.

$f_n(1,n)$.

$f_n(s(X),Y) \Leftarrow f_n(X,Z),mult\_by\_n(Z,Y)$.

where `mult_by_n` multiplies the first argument by n. It is easy to see that for any n, $f_n$ zero utilizes `mult_by_n`. Therefore, the same conclusion apply for any $f_n$ that for $f_2$ (also called `f`), as discussed above.

# 7 Conclusion

Inductive logic programming involves inverting a notion of generalization. Two notions of generalization are well-known in logic programming—θ-subsumption and logical implication. The θ-subsumption notion is not as powerful as implication, because it cannot represent the generality relation for a clause and the resolvent of this clause with itself. In fact, recursion is not handled by this notion of generalization. All previous ILP systems only invert the θ-subsumption relation of generalization and then inherit the inadequacy of this relation for generalizing recursive concepts. Our system is the first that inverts the implication relation of generalization. Inverting implication is a very important problem and yet no attempt had led to some results before our work. Furthermore, inverting implication frees us from the need of basic representative sets for induction of recursive concepts.

## 7.1 Summary of results

This thesis presents a system called LOPSTER that learns a class of recursive logic programs quickly from as few as two examples. The system takes advantage of regularities in the structure of example terms, an idea consistent with that recently proposed by Aha [in press]. Term structure is analyzed in terms of a newly-developed operation called sub-unification. Sub-unification identifies examples with some similarities and then induces the arguments of a recursive clause that generalizes them. Generalization is accomplished by logical implication, a form stronger than the usual θ-subsumption. LOPSTER appears to be the first system to use this form. Unlike θ-subsumption, implication generalization does not require that the training examples include a basic representative set.

Sub-unification is the concept at the center of our research. This logical technique unifies a term with subterms of another term. LOPSTER relies on it to discover from repetitive term structure the arguments and depth of the clause being induced.

Early experiments indicate that LOPSTER performs an order of magnitude better than other systems on the class of purely recursive clauses. Despite their apparent simplicity such clauses are often used in many benchmarks of ILP systems. A left-recursive mode has also been implemented and shows the expected and quite satisfactory results .

We envision LOPSTER's purely recursive and left-recursive modes serving as front-ends to a general ILP system. LOPSTER can be run on a small number of examples. If an answer is obtained it will be found quickly and a training set adequate to induce it will have been acquired with relatively little expenditure of resources. Should LOPSTER fail, it will fail quickly and the user has some justification in moving on to processes which trade efficiency for generality.

## 7.2 Future work

We have focused on learning a single clause in this thesis. It would be interesting to investigate in detail how to generalize a set of clauses which have been induced by LOPSTER; the similar form of all these clauses might suggest a direction in which to proceed. Our examples have all assumed the general base case of the recursive program because we believe that usually this is not difficult to provide. However, sometimes only ground facts are available, suggesting that the base case to be used in the induction is not general enough. In such cases we need to be able to generate a more general base case; a method to do so was briefly introduced in sec. 4.1.1.1 and we believe further work on it

would be very fruitful. In summary, any research to broaden the applicability of a system based on sub-unification would likely be interesting.

We have not formally defined the heuristics used to choose a literal to be added to the body of the clause being induced when LOPSTER is operating in left-recursive mode. Further research on these heuristics is needed. Furthermore, we can immediately envision some straightforward improvements to the selection process—instead of using only one sub-unified pair of literals, several sub-unified pairs of literals having the same generating terms might be used. A literal could then be added if the bindings of the new inputs with known variables correspond in all pairs. Doing so would allow non-variable arguments to be introduced to the added literal. Taking the LGG of all instances of the literal being added (one instance per sub-unified pair) is a plausible approach.

It would be interesting to implement and test the method described in sec. 4.2 to specialize a clause based on explained negative examples. We believe this is a promising line of research; it is not difficult to provide an explanation for something that does not fit our target concept. People seldom fail to understand negative events and it is probably easier to explain negativeness than positiveness (positive events being in many cases trivial yet unexplainable).

Good heuristics to avoid an explosion of the search space would need attention, should the extensions to our method proposed in Chapter 5 be adopted. We believe that extending the notion of sub-unification to unify subterms with subterms would be both realistic and useful. It would not cause the search space to explode of itself. In fact, many sub-unifiers would be eliminated from the search for generating terms with corresponding depth.

Constructive induction opens interesting avenues of investigation. It has already been noted that often it is easy to induce the definition of a new (necessary) predicate with LOPSTER. One might consider doing so by adding a new predicate to the right of the recursive call when LOPSTER is running in left-recursive mode and background knowledge cannot be used successfully. The form of clauses using a new predicate would be `p` $\Leftarrow$ `p, newp`, where `newp` is the new predicate. However there is a problem; because our method does not need basic representative sets it may not be possible to collect instances of the new predicate. In fact because of the way our method is designed, given the instance of `p` in the head of the above clause, the instance of the `p` in the body is not necessarily known. The only way to overcome this is to rely on an oracle to answer questions about the target predicate p (it does not make sense to ask questions about the new predicate). For instance in learning `multiply` and inventing `plus`, one might ask the oracle "What is multiply(3,2) ?" to collect an instance of `plus`.

# References


Aha, D.W. (in press). Relating Relational Learning Algorithms. *Advances in Inductive Logic Programming*. Academic Press. London. Edited by S. Muggleton.

Genesereth, M.R. & Nilson, N.J. (1987). *Logical foundation of artificial intelligence.* Morgan Kaufmann Publishers inc.

Lapointe, S. & Matwin, S. (1992a). Sub-unification: A tool for efficient induction of recursive programs. In *Proceedings of Ninth International Machine Learning Conference* (ML-92). Aberdeen, Scotland. Morgan Kaufmann.

Lapointe, S. & Matwin, S. (1992b). Induction de programmes logiques récursifs fondée sur la sous-unification. In *Journées francophones d'apprentissage et d'explication des connaissances* (JFAEC-92). Dourdan, France.

Ling, C.X. (1991a). Inductive Learning from Good Examples. In *IJCAI-91* (pages 751-756).

Ling, C.X. (1991b). Inventing Necessary Theoretical Terms. Technical Report No. 302, Department of Computer Science, University of Western Ontario, 1991.

Ling, C.X. & Narayan, M.A. (1991). A critical comparison of various methods based on inverse resolution. In *MLW-91*.

Lloyd, J.W. (1987). *Foundations of logic programming* (2nd ed.). Springer-Verlag. Berlin.

Muggleton, S.H. (1990). Inductive logic programming. In *Proceedings of the First Conference on Algorithmic Learning Theory*. Tokyo. Ohmsha.

Muggleton, S.H. (1992). Inverting implication. To appear in *Proceedings of ECAI-92, Workshop on Logical Approaches to Machine Learning*. Vienna.

Muggleton, S.H. & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Fifth International Conference on Machine Learning* (pages 339-352). Ann Arbor, Michigan. Morgan Kaufmann.

Muggleton, S.H. & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory* (pages 368-381). Tokyo. Ohmsha.

Niblett, T. (1988). A study of generalization in logic programs. In *EWSL-88* (pages 131-138). London. Pitman.

Plotkin, G.D. (1970). A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5* (pages 153-163). Elsevier North Holland, New York.

Plotkin, G.D. (1971). A further note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6* (pages 101-124). Elsevier North Holland, New York.

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

Quinlan, J.R. (1991a). Determinate literals in inductive logic programming. In *IJCAI-91* (pages 746-750). Sydney.

Quinlan, J.R. (1991b). Knowledge acquisition from structured data. In *IEEE EXPERT*, December 1991 (pages 32-37).

Robinson, J.A. (1965). A machine-oriented logic based on the resolution principle. *JACM*, 12(1) (pages 23-41).

Rouveirol, C. (1991). ITOU: Induction of first order theories. In *Proceedings of International Workshop on Inductive Logic Programming* (pages 127-157). Viana de Castelo, Portugal.

Rouveirol, C. & Puget, J.F. (1990). Beyond inversion of resolution. In *ML-90*.

Schank, R.C. (1991). Where's the AI ? *AI magazine*, winter 1991, vol. 12, no. 4 (pages 38-49).

Silverstein, G. & Pazzani, M.J. (1991). Relational clichés: Constraining constructive induction during relational learning. In *Proceedings of Eight International Machine Learning Workshop (ML-91)*.

Wirth, R. (1989). Completing logic programs by inverse resolution. In *EWSL-89* (pages 239-250). London. Pitman.

Wirth, R. & O'Rorke, P. (1991). Constraints on predicate invention. In *ML-91* (pages 457-461).

# Appendix A: A demonstration of purely recursive mode

The purely recursive mode of LOPSTER has been implemented in Quintus Prolog in a program of approximately one hundred lines. The following pages show the results of the execution of several examples, including those used for the comparison in sec. 4.5.2. This implementation displays all possible clauses of any depth. Runtime displayed is for the whole search. Every embedding term of a sub-unifier (mgsu) is displayed even though some of them do not lead to the induction of a clause. Note that for all these problems, the number of sub-unifiers is very small and the runtime is very low.

```
{kaml2}-Stephane-(30) prolog
Quintus Prolog Release 3.1.1 (Sun-4, SunOS 4.1)
Copyright (C) 1990, Quintus Corporation.  All rights reserved.
2100 Geng Road, Palo Alto, California U.S.A. (415) 813-3800
% compiling file /home/kaml1/usr3/sr/lapointe/prolog.ini
% prolog.ini compiled in module user, 0.050 sec 332 bytes

| ?- [mpr].
% compiling file /tmp_mnt/home/kaml1/usr3/sr/lapointe/these/mode.purely-recursive/mpr.pl

                    LOPSTER
                    =======
          PURELY RECURSIVE MODE
          ---------------------------------

If you wish to see demos, type "showdemos."

You can also enter your problem of the following form:
"induce(Base_example,Complex_example)."

% mpr.pl compiled in module user, 2.867 sec 11,608 bytes

yes
| ?- showdemos.


        LIST OF DEMOS:
        ==============

demo(1)
----------
member(A,[A|B]).
member(4,[1,2,3,4,5]).

demo(2)
----------
plus(0,A,A).
plus(s(s(s(0))),s(s(0)),s(s(s(s(s(0))))))).

demo(3)
----------
extractNth(s(0),[c|A],c).
extractNth(s(s(s(0))),[a,b,c,d,c,d],c).

demo(4)
----------
last_of(A,[A]).
last_of(i,[f,g,h,i]).
```

```
demo(5)
----------
noneIsZero([]).
noneIsZero([s(0),s(s(s(s(0))))]).

demo(6)
----------
delete(A,[A|B],B).
delete(c,[a,b,c,d,e],[a,b,d,e]).

demo(7)
----------
append([],A,A).
append([a,b,A],[1,2],[a,b,A,1,2]).

demo(8)
----------
reverse([],[]).
reverse([a,b],append(append([],[b]),[a])).

demo(9)
----------
fact(s(0),s(0)).
fact(s(s(s(0))),s(s(s(0)))* (s(s(0))*s(0))).

demo(10)
----------
split([],[],[]).
split([a,b,c,d,e,f],[a,c,e],[b,d,f]).


For running a demo, type "demo(No)."

You can also enter your problem of the following form:
"induce(Base_example,Complex_example)."


yes
| ?- demo(1).


BASE EXAMPLE:
---------------------
member(A,[A|B]).

COMPLEX EXAMPLE:
--------------------------
member(4,[1,2,3,4,5]).
```

```
INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
['V',[1,2,3|'V']].

member(A,[B|C]) :-
      member(A,C).

Depth of the induced clause: 3


--------------------
Runtime is 16ms

yes
| ?- demo(2).


BASE EXAMPLE:
---------------------
plus(0,A,A).

COMPLEX EXAMPLE:
----------------------------
plus(s(s(s(0))),s(s(0)),s(s(s(s(s(0))))))).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),'V',s(s(s('V')))].

plus(s(A),B,s(C)) :-
      plus(A,B,C).

Depth of the induced clause: 3

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s(s('V'))),s('V'),s(s(s(s('V'))))].

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s(s('V'))),s(s('V')),s(s(s(s(s('V'))))))].
```

```
---------------------
Runtime is 17ms

yes
| ?- demo(3).


BASE EXAMPLE:
----------------------
extractNth(s(0),[c|A],c).

COMPLEX EXAMPLE:
---------------------------
extractNth(s(s(s(0))),[a,b,c,d,c,d],c).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[s(s('V')),[a,b|'V'],'V'].

extractNth(s(A),[B|C],c) :-
      extractNth(A,C,c).

Depth of the induced clause: 2

----- Current Sub-Unification -----

Embedding Terms:
[s(s('V')),[a,b,c,d|'V'],'V'].

extractNth(s(A),[B,C|D],c) :-
      extractNth(A,D,c).

Depth of the induced clause: 2


---------------------
Runtime is 33ms

yes
| ?- demo(4).


BASE EXAMPLE:
---------------------
last_of(A,[A]).
```

```
COMPLEX EXAMPLE:
----------------------------
last_of(i,[f,g,h,i]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
['V',[f,g,h|'V']].

last_of(A,[B|C]) :-
      last_of(A,C).

Depth of the induced clause: 3


---------------------
Runtime is 17ms

yes
| ?- demo(5).


BASE EXAMPLE:
---------------------
noneIsZero([]).

COMPLEX EXAMPLE:
----------------------------
noneIsZero([s(0),s(s(s(s(0))))]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[[s(0),s(s(s(s(0))))|'V']].

noneIsZero([s(A)|B]) :-
      noneIsZero(B).

Depth of the induced clause: 2


---------------------
Runtime is 33ms

yes
| ?- demo(6).
```

```
BASE EXAMPLE:
---------------------
delete(A,[A|B],B).

COMPLEX EXAMPLE:
----------------------------
delete(c,[a,b,c,d,e],[a,b,d,e]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
['V',[a,b|'V'],[a,b|'V']].

delete(A,[B|C],[B|D]) :-
      delete(A,C,D).

Depth of the induced clause: 2


---------------------
Runtime is 17ms

yes
| ?- demo(7).


BASE EXAMPLE:
---------------------
append([],A,A).

COMPLEX EXAMPLE:
----------------------------
append([a,b,A],[1,2],[a,b,A,1,2]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[[a,b,A|'V'],'V',[a,b,A|'V']].

append([A|B],C,[A|D]) :-
      append(B,C,D).

Depth of the induced clause: 3
```

----- Current Sub-Unification -----

Embedding Terms:
[[a,b,A|'V'],['V',2],[a,b,A,'V',2]].

----- Current Sub-Unification -----

Embedding Terms:
[[a,b,A|'V'],[1|'V'],[a,b,A,1|'V']].

----- Current Sub-Unification -----

Embedding Terms:
[[a,b,A|'V'],[1,'V'],[a,b,A,1,'V']].

----- Current Sub-Unification -----

Embedding Terms:
[[a,b,A|'V'],[1,2|'V'],[a,b,A,1,2|'V']].

---
Runtime is 67ms

yes
| ?- demo(8).

BASE EXAMPLE:
---
reverse([],[]).

COMPLEX EXAMPLE:
---
reverse([a,b],append(append([],[b]),[a])).

INDUCED CLAUSES:
==========

----- Current Sub-Unification -----

Embedding Terms:
[[a,b|'V'],append(append('V',[b]),[a])].

```
reverse([A|B],append(C,[A])) :-
      reverse(B,C).
```

Depth of the induced clause: 2

----- Current Sub-Unification -----

```
Embedding Terms:
[[a,b|'V'],append(append([],[b|'V']),[a])].

----- Current Sub-Unification -----

Embedding Terms:
[[a,b|'V'],append(append([],[b]),[a|'V'])].


--------------------
Runtime is 17ms

yes
| ?- demo(9).


BASE EXAMPLE:
---------------------
fact(s(0),s(0)).

COMPLEX EXAMPLE:
----------------------------
fact(s(s(s(0))),s(s(s(0)))* (s(s(0))*s(0))).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[s(s('V')),s(s('V'))* (s(s(0))*s(0))].

----- Current Sub-Unification -----

Embedding Terms:
[s(s('V')),s(s(s(0)))* (s('V')*s(0))].

----- Current Sub-Unification -----

Embedding Terms:
[s(s('V')),s(s(s(0)))* (s(s(0))*'V')].

fact(s(A),s(A)*B) :-
      fact(A,B).

Depth of the induced clause: 2

fact(s(A),s(s(B))*C) :-
      fact(A,C).

Depth of the induced clause: 2
```

```
---------------------
Runtime is 34ms

yes
| ?- demo(10).


BASE EXAMPLE:
----------------------
split([],[],[]).

COMPLEX EXAMPLE:
-----------------------------
split([a,b,c,d,e,f],[a,c,e],[b,d,f]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[[a,b,c,d,e,f|'V'],[a,c,e|'V'],[b,d,f|'V']].

split([A,B|C],[A|D],[B|E]) :-
      split(C,D,E).

Depth of the induced clause: 3


---------------------
Runtime is 17ms

yes
| ?- induce( append([],[d,e],[d,e]) , append([a,b,c],[d,e],[a,b,c,d,e]) ).


BASE EXAMPLE:
----------------------
append([],[d,e],[d,e]).

COMPLEX EXAMPLE:
-----------------------------
append([a,b,c],[d,e],[a,b,c,d,e]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[[a,b,c|'V'],'V',[a,b,c|'V']].
```

```
append([A|B],[d,e],[A|C]) :-
      append(B,[d,e],C).

Depth of the induced clause: 3


------------------
Runtime is 0ms

yes
| ?- induce( append([],L,L) , append([a,a,a],[b,c],[a,a,a,b,c]) ).


BASE EXAMPLE:
---------------------
append([],A,A).

COMPLEX EXAMPLE:
----------------------------
append([a,a,a],[b,c],[a,a,a,b,c]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[[a,a,a|'V'],'V',[a,a,a|'V']].

append([a|A],B,[a|C]) :-
      append(A,B,C).

Depth of the induced clause: 3

----- Current Sub-Unification -----

Embedding Terms:
[[a,a,a|'V'],['V',c],[a,a,a,'V',c]].

----- Current Sub-Unification -----

Embedding Terms:
[[a,a,a|'V'],[b|'V'],[a,a,a,b|'V']].

----- Current Sub-Unification -----

Embedding Terms:
[[a,a,a|'V'],[b,'V'],[a,a,a,b,'V']].

----- Current Sub-Unification -----
```

```
Embedding Terms:
[[a,a,a|'V'],[b,c|'V'],[a,a,a,b,c|'V']].


--------------------
Runtime is 50ms

L = _6918

| ?- induce( plus(s(s(0)), s(s(0)), s(s(s(s(0)))) ) ,
             plus(s(s(s(s(s(s(0))))))), s(s(0)), s(s(s(s(s(s(s(s(0)))))))) ) ).


BASE EXAMPLE:
-------------- ----------
plus(s(s(0)),s(s(0)),s(s(s(s(0))))).

COMPLEX EXAMPLE:
----------------------------
plus(s(s(s(s(s(s(0)))))),s(s(0)),s(s(s(s(s(s(s(s(0)))))))).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s(s('V')))),'V',s(s(s(s('V'))))].

plus(s(A),s(s(0)),s(B)) :-
      plus(A,s(s(0)),B).

Depth of the induced clause: 4

plus(s(s(A)),s(s(0)),s(s(B))) :-
      plus(A,s(s(0)),B).

Depth of the induced clause: 2


----------------------
Runtime is 33ms

yes
```

# Appendix B: A demonstration of left-recursive mode

The left-recursive mode of LOPSTER has been implemented in Quintus Prolog in a program of approximately two hundred lines. The following pages show the results of the execution of every example used in sec. 4.5.3. The program displays all possible clauses of any depth that satisfy the settings of the parameters. As with purely recursive mode, runtime displayed is for the whole search and all embedding terms are shown.

Some explanations are necessary to understand this demo. Induction is performed by sub-unifying the base example in the complex example. All possible sub-unifiers and all possible generating terms for a given sub-unifier are considered. The program accepts four parameters: *dl* (maximal depth of output arguments), *cp* (completeness WRT to the other positive examples not used in the induction), *cs* (consistency WRT to the given negative examples), *pn* (pruning). *pn* is not yet in use and is not discussed. *dl* is used to restrict the clauses that can be induced as described in sec. 4.4. If *cp* is on, only induced clauses that can prove all positive examples not used in the induction are displayed. If *cs* is on, only induced clauses that cannot prove any of the negative examples are displayed.

## LIST INVERSION

```
{kaml2}-Stephane-(10) prolog
Quintus Prolog Release 3.1.1 (Sun-4, SunOS 4.1)
Copyright (C) 1990, Quintus Corporation.  All rights reserved.
2100 Geng Road, Palo Alto, California U.S.A. (415) 813-3800
% compiling file /home/kaml1/usr3/sr/lapointe/prolog.ini
% prolog.ini compiled in module user, 0.050 sec 332 bytes

| ?- [mlr].
% compiling file /tmp_mnt/home/kaml1/usr3/sr/lapointe/these/mode.left-recursive/mlr.pl

                LOPSTER
                ========
        LEFT-RECURSIVE MODE
        -------------------

% mlr.pl compiled in module user, 4.500 sec 20,304 bytes

yes
| ?- load(te,reverse).

Base example:
-------------
reverse([],[]).

Complex example:
----------------
reverse([a,b,c],[c,b,a]).

yes
| ?- load(bk,reverse).

Background knowledge:
---------------------
reverse([],[]).
add_to_end([],A,[A]).
add_to_end([A|B],C,[A|D]) :-
      add_to_end(B,C,D).

Mode declaration:
-----------------
reverse(+,-).
add_to_end(+,+,-).

yes
| ?- induce.
```

```
Parameter Name        | Value
----------------------------------
dl (Depth of Literals) =  2
cp (Completeness)      =  no
cs (Consistency)       =  no
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.


BASE EXAMPLE:
-----------------------
reverse([],[]).

COMPLEX EXAMPLE:
---------------------------
reverse([a,b,c],[c,b,a]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[[a,b,c|'V']].

reverse([A|B],C) :-
      reverse(B,D),
      add_to_end(D,A,C).

Depth of the induced clause: 3


---------------------
Runtime is 33ms

yes
| ?- set(dl,3).

Parameter Name        | Value
----------------------------------
dl (Depth of Literals) =  3
cp (Completeness)      =  no
cs (Consistency)       =  no
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.

yes
| ?- induce.
```

```
Parameter Name       | Value
---------------------------------
dl (Depth of Literals) =  3
cp (Completeness)      =  no
cs (Consistency)       =  no
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.


BASE EXAMPLE:
---------------------
reverse([],[]).

COMPLEX EXAMPLE:
----------------------------
reverse([a,b,c],[c,b,a]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[[a,b,c|'V']].

reverse([A|B],C) :-
      reverse(B,D),
      add_to_end(D,A,C).

Depth of the induced clause: 3


---------------------
Runtime is 50ms

yes
```

## FACTORIAL

```
| ?- load(te,fact).

Base example:
-------------
fact(s(0),s(0)).

Complex example:
----------------
fact(s(s(s(0))),s(s(s(s(s(s(0)))))).

yes
| ?- load(bk,fact).

Background knowledge:
---------------------
fact(s(0),s(0)).
time(0,0,0).
time(0,s(0),0).
time(0,s(s(0)),0).
time(0,s(s(s(0))),0).
time(0,s(s(s(s(0)))),0).
time(s(0),0,0).
time(s(0),s(0),s(0)).
time(s(0),s(s(0)),s(s(0))).
time(s(0),s(s(s(0))),s(s(s(0)))).
time(s(0),s(s(s(s(0)))),s(s(s(s(0))))).
time(s(s(0)),0,0).
time(s(s(0)),s(0),s(s(0))).
time(s(s(0)),s(s(0)),s(s(s(s(0))))).
time(s(s(0)),s(s(s(0))),s(s(s(s(s(s(0)))))).
time(s(s(0)),s(s(s(s(0)))),s(s(s(s(s(s(s(s(s(0)))))))))).
time(s(s(s(0))),0,0).
time(s(s(s(0))),s(0),s(s(s(0)))).
time(s(s(s(0))),s(s(0)),s(s(s(s(s(s(s(0)))))))).
time(s(s(s(0))),s(s(s(0))),s(s(s(s(s(s(s(s(s(0))))))))).
time(s(s(s(0))),s(s(s(s(0)))),s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))).
time(s(s(s(s(0)))),0,0).
time(s(s(s(s(0)))),s(0),s(s(s(s(0))))).
time(s(s(s(s(0)))),s(s(0)),s(s(s(s(s(s(s(s(s(0))))))))).
time(s(s(s(s(0)))),s(s(s(0))),s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))).
time(s(s(s(s(0)))),s(s(s(s(0)))),s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))))).
time(s(s(s(s(0)))),s(s(s(s(s(0))))),
        s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))))))))))).

Mode declaration:
-----------------
fact(+,-).
time(+,+,-).
```

```
yes
| ?- induce.

Parameter Name        | Value
----------------------------------
dl (Depth of Literals) =  2
cp (Completeness)      =  no
cs (Consistency)       =  no
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.


BASE EXAMPLE:
---------------------
fact(s(0),s(0)).

COMPLEX EXAMPLE:
----------------------------
fact(s(s(s(0))),s(s(s(s(s(s(s(0))))))).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[s(s('V'))].

fact(s(A),B) :-
      fact(A,C),
      time(C,s(A),B).

Depth of the induced clause: 2

fact(s(A),B) :-
      fact(A,C),
      time(s(A),C,B).

Depth of the induced clause: 2


----------------------
Runtime is 250ms

yes
| ?- set(dl,3).
```

```
Parameter Name        | Value
------------------------------
dl (Depth of Literals) =  3
cp (Completeness)      =  no
cs (Consistency)       =  no
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.

yes
| ?- induce.

Parameter Name        | Value
------------------------------------
dl (Depth of Literals) =  3
cp (Completeness)      =  no
cs (Consistency)       =  no
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.


BASE EXAMPLE:
---------------------
fact(s(0),s(0)).

COMPLEX EXAMPLE:
---------------------------
fact(s(s(s(0))),s(s(s(s(s(s(0)))))).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[s(s('V'))].

fact(s(A),B) :-
      fact(A,C),
      time(C,s(A),B).

Depth of the induced clause: 2

fact(s(A),B) :-
      fact(A,C),
      time(s(A),C,B).

Depth of the induced clause: 2
```

```
----------
Runtime is 400ms

yes
```

INSERTION SORT
==========

```
| ?- load(te,isort).

Base example:
-------------
isort([],[]).

Complex example:
----------------
isort([5,2,4,1,3,6],[1,2,3,4,5,6]).

yes
| ?- load(bk,isort).

Background knowledge:
---------------------
isort([],[]).
insert(A,[],[A]) :-
      call(integer(A)).
insert(A,[B|C],[A,B|C]) :-
      call(A=<B).
insert(A,[B|C],[B|D]) :-
      insert(A,C,D),
      call(A>B).

Mode declaration:
-----------------
isort(+,-).
insert(+,+,-).

yes
| ?- induce.

Parameter Name       | Value
---------------------------
dl (Depth of Literals) =  2
cp (Completeness)      =  no
cs (Consistency)       =  no
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.


BASE EXAMPLE:
-------------
isort([],[]).
```

COMPLEX EXAMPLE:
---
isort([5,2,4,1,3,6],[1,2,3,4,5,6]).

INDUCED CLAUSES:
==========

----- Current Sub-Unification -----

Embedding Terms:
[[5,2,4,1,3,6|'V']].

```
isort([A|B],C) :-
      isort(B,D),
      insert(A,D,C).
```

Depth of the induced clause: 6

```
isort([A,B|C],D) :-
      isort(C,E),
      insert(A,E,F),
      insert(B,F,D).
```

Depth of the induced clause: 3

```
isort([A,B|C],D) :-
      isort(C,E),
      insert(B,E,F),
      insert(A,F,D).
```

Depth of the induced clause: 3

---
Runtime is 500ms

yes
| ?- set(dl,3).

| Parameter Name | | Value |
|---|---|---|
| dl (Depth of Literals) | = | 3 |
| cp (Completeness) | = | no |
| cs (Consistency) | = | no |
| pn (Pruning) | = | no |

Use "set(P,V)" to change parameter P to value V.

yes
| ?- induce.

```
Parameter Name        | Value
-----------------------------------
dl (Depth of Literals) =  3
cp (Completeness)      =  no
cs (Consistency)       =  no
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.


BASE EXAMPLE:
---------------------
isort([],[]).

COMPLEX EXAMPLE:
---------------------------
isort([5,2,4,1,3,6],[1,2,3,4,5,6]).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[[5,2,4,1,3,6|'V']].

isort([A|B],C) :-
      isort(B,D),
      insert(A,D,C).

Depth of the induced clause: 6

isort([A,B|C],D) :-
      isort(C,E),
      insert(A,E,F),
      insert(B,F,D).

Depth of the induced clause: 3

isort([A,B|C],D) :-
      isort(C,E),
      insert(B,E,F),
      insert(A,F,D).

Depth of the induced clause: 3
```

```
isort([A,B,C|D],E) :-
        isort(D,F),
        insert(A,F,G),
        insert(C,G,H),
        insert(B,H,E).

Depth of the induced clause: 2

isort([A,B,C|D],E) :-
        isort(D,F),
        insert(C,F,G),
        insert(A,G,H),
        insert(B,H,E).

Depth of the induced clause: 2

isort([A,B,C|D],E) :-
        isort(D,F),
        insert(A,F,G),
        insert(B,G,H),
        insert(C,H,E).

Depth of the induced clause: 2

isort([A,B,C|D],E) :-
        isort(D,F),
        insert(B,F,G),
        insert(A,G,H),
        insert(C,H,E).

Depth of the induced clause: 2

isort([A,B,C|D],E) :-
        isort(D,F),
        insert(C,F,G),
        insert(B,G,H),
        insert(A,H,E).

Depth of the induced clause: 2

isort([A,B,C|D],E) :-
        isort(D,F),
        insert(B,F,G),
        insert(C,G,H),
        insert(A,H,E).

Depth of the induced clause: 2

----------------------
Runtime is 734ms

yes
```

## MULTIPLY

```
| ?- load(all,time).

Base example:
-------------
time(0,A,0).

Complex example:
----------------
time(s(s(s(0))),s(s(0)),s(s(s(s(s(s(0)))))).

Positive examples:
------------------
time(s(s(s(0))),s(s(s(0))),s(s(s(s(s(s(s(s(s(0)))))))))).

Negative examples:
------------------
time(s(s(0)),s(s(0)),s(s(0))).

Background knowledge:
---------------------
time(0,A,0).
plus(0,A,A).
plus(s(A),B,s(C)) :-
      plus(A,B,C).

Mode declaration:
-----------------
time(+,+,-).
plus(+,+,-).

yes
| ?- induce.

Parameter Name       | Value
---------------------------
dl (Depth of Literals) =  2
cp (Completeness)      =  no
cs (Consistency)       =  no
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.


BASE EXAMPLE:
-------------
time(0,A,0).
```

COMPLEX EXAMPLE:
---
time(s(s(s(0))),s(s(0)),s(s(s(s(s(s(s(0))))))).

INDUCED CLAUSES:
==========

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),'V'].

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(A,D,E),
      plus(A,E,C).
```

Depth of the induced clause: 3

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(D,A,E),
      plus(A,E,C).
```

Depth of the induced clause: 3

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(B,D,C).
```

Depth of the induced clause: 3

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(s(A),D,C).
```

Depth of the induced clause: 3

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(D,s(A),C).
```

Depth of the induced clause: 3

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(A,D,E),
      plus(E,A,C).
```

Depth of the induced clause: 3

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(D,A,E),
      plus(E,A,C).

Depth of the induced clause: 3

time(s(A),B,C) :-
      time(A,B,D),
      plus(D,B,C).

Depth of the induced clause: 3

time(s(A),B,C) :-
      time(A,B,D),
      plus(A,A,E),
      plus(E,D,C).

Depth of the induced clause: 3

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),s('V')].

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),s(s('V'))].


----------------------
Runtime is 2366ms

yes
| ?- set(cs,yes).

Parameter Name        | Value
-----------------------------------
dl (Depth of Literals) =  2
cp (Completeness)      =  no
cs (Consistency)       =  yes
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.

yes
| ?- induce.
```

```
Parameter Name        | Value
-----------------------------------
dl (Depth of Literals) =  2
cp (Completeness)      =  no
cs (Consistency)       =  yes
pn (Pruning)           =  no

Use "set(P,V)" to change parameter P to value V.


BASE EXAMPLE:
---------------------
time(0,A,0).

COMPLEX EXAMPLE:
-----------------------------
time(s(s(s(0))),s(s(0)),s(s(s(s(s(s(0)))))).

INDUCED CLAUSES:
================

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),'V'].

time(s(A),B,C) :-
      time(A,B,D),
      plus(B,D,C).

Depth of the induced clause: 3

time(s(A),B,C) :-
      time(A,B,D),
      plus(s(A),D,C).

Depth of the induced clause: 3

time(s(A),B,C) :-
      time(A,B,D),
      plus(D,s(A),C).

Depth of the induced clause: 3

time(s(A),B,C) :-
      time(A,B,D),
      plus(D,B,C).

Depth of the induced clause: 3

----- Current Sub-Unification -----
```

```
Embedding Terms:
[s(s(s('V'))),s('V')].

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),s(s('V'))].


------------------------
Runtime is 2316ms

yes
| ?- set(cs,no).

Parameter Name       | Value
---------------------------------
dl (Depth of Literals) =  2
cp (Completeness)    =  no
cs (Consistency)     =  no
pn (Pruning)         =  no

Use "set(P,V)" to change parameter P to value V.

yes
| ?- set(cp,yes).

Parameter Name       | Value
---------------------------------
dl (Depth of Literals) =  2
cp (Completeness)    =  yes
cs (Consistency)     =  no
pn (Pruning)         =  no

Use "set(P,V)" to change parameter P to value V.

yes
| ?- induce.

Parameter Name       | Value
---------------------------------
dl (Depth of Literals) =  2
cp (Completeness)    =  yes
cs (Consistency)     =  no
pn (Pruning)         =  no

Use "set(P,V)" to change parameter P to value V.
```

BASE EXAMPLE:
---
time(0,A,0).

COMPLEX EXAMPLE:
---
time(s(s(s(0))),s(s(0)),s(s(s(s(s(s(0)))))).

INDUCED CLAUSES:
==========

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),'V'].

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(B,D,C).
```

Depth of the induced clause: 3

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(D,B,C).
```

Depth of the induced clause: 3

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),s('V')].

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),s(s('V'))].

---
Runtime is 2350ms

yes
| ?- set(dl,3).

| Parameter Name | | Value |
|---|---|---|
| dl (Depth of Literals) | = | 3 |
| cp (Completeness) | = | yes |
| cs (Consistency) | = | no |
| pn (Pruning) | = | no |

Use "set(P,V)" to change parameter P to value V.

yes
| ?- induce.

| Parameter Name | | Value |
|---|---|---|
| dl (Depth of Literals) | = | 3 |
| cp (Completeness) | = | yes |
| cs (Consistency) | = | no |
| pn (Pruning) | = | no |

Use "set(P,V)" to change parameter P to value V.

BASE EXAMPLE:
---
time(0,A,0).

COMPLEX EXAMPLE:
---
time(s(s(s(0))),s(s(0)),s(s(s(s(s(s(0)))))).

INDUCED CLAUSES:
==========

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),'V'].

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(B,D,C).
```

Depth of the induced clause: 3

```
time(s(A),B,C) :-
      time(A,B,D),
      plus(D,B,C).
```

Depth of the induced clause: 3

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),s('V')].

----- Current Sub-Unification -----

Embedding Terms:
[s(s(s('V'))),s(s('V'))].

---
Runtime is 7517ms

yes