Inductive Inference of Functional Logic Programs by Inverse Narrowing *

J. Hernández-Orallo § M.J. Ramírez-Quintana§

§ DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain.
Email: {jorallo,mramirez}@dsic.upv.es.

Abstract: A framework for the Induction of Functional Logic Programs (IFLP) from facts is presented. Inspired in the inverse resolution operator of ILP, we study the reversal of narrowing, the most usual operational mechanism for functional logic programming. We also generalize the selection criteria for guiding the search, including coherence criteria in addition to the MDL principle. A non-incremental learning algorithm and a more sophisticated incremental extension of it are presented. We discuss the advantages of IFLP over ILP, most of which are inherited from the power of narrowing w.r.t. resolution and the limitation of conditions, a usual gate for extensional exceptions. At the end of this paper, we comment on the adaptability of our techniques to higher-order induction.

1 Introduction

Since the beginning of this decade, Inductive Logic Programming (ILP) has been a very important area of research as an appropiate framework for the inductive inference of first-order clausal theories from facts. Inductive inference operators are usually obtained by inverting deductive ones. The most interesting approach is based on the inversion of the resolution principle. As we will show, our proposal induces equational clauses in a way which is quite similar.

ILP has provided an outstanding advantage in the inductive machine learning field by increasing the applicability of learning systems to first-order theories (and not only propositional ones); however, ILP has also inherited the main limitations of computational logic: the impossibility of defining functions in a natural way and the absence of higher-order constructs.

During the last decade, it has been theoretically and experimentally shown that *functional logic languages have more expressive power in comparison to functional languages and a better operational behavior in comparison to logic languages* [2]. One relevant approach to the integration is that in which functional logic programs are logic programs which are augmented with Horn equational theories. The main semantic properties of logic programs also hold for functional logic programs. Thus, these programs admit least model and fixpoint semantics. The operational semantics of a functional logic language is defined in terms of semantic unification or $\mathcal{E}$-unification. A sound and complete $\mathcal{E}$-unification method is narrowing [4, 5]. Several strategies have been proposed to improve the efficiency of the narrowing algorithm.

The induction of functional logic programs has recently been addressed. A framework for the induction of Escher programs is presented in [1]. Escher [8] is an integrated logic and functional programming language based on the Church theory of types which incorporates some higher-order concepts. The syntax of programs is functional (as in the Haskell language) and the computational model of Escher is based on the rewriting mechanism. Since functions operate on data types with several data constructors, the proposed algorithm chooses one of the arguments as pattern for the induction of a function and partitions the examples according to the constructor appearing in them in this argument. Then, one statement is learned for each case. On the contrary, our approach does not consider pattern scheme and it is oriented to working with languages which are based on narrowing and are not typed.

In this work, we present a general framework for the induction of functional logic programs (IFLP) from examples, generalizing the scope of ILP. At the moment, we will consider the unconditional case. For simplicity, the (positive and negative) examples are expressed as pairs of ground terms or ground equations where the right term is in normal form. Positive examples represent terms that will have to be proven equal using the induced program, whereas negative examples consist of terms that do not have

*This work has been partially supported by CICYT under grant TIC 95-0433-C03-03.

to be proven equal. Our approach is based on the idea of the inverse resolution of ILP. Starting from the generalization of positive examples to include variables as arguments of functions, we have defined an inverse narrowing mechanism which selects pairs of equations to obtain an equation which is more general than the original ones from the generalized examples. This process is repeated until a program (a set of equations) is valid according to some selection criteria.

In our opinion, one important reason for undertaking the jump to IFLP is that once are established the properties and behaviour of different inverted narrowing techniques, the step to higher order induction may be easier to bridge based on the deductive higher-order counterparts.

2 Preliminaries

We briefly review some basic concepts about ILP, equations, Term Rewriting Systems and $\mathcal{E}$-unification. For any concept which is not explicitly defined the reader may refer to [2, 6, 9].

The problem addressed by ILP can be simply stated as the inference of a theory (a logic program) P from facts (or evidence logic theory) using a background knowledge theory B (another logic program). Evidence can be only positive E^+ or both positive and negative (E^+,E^-). The sets E^+ and E^- are usually given in an extensional manner (i.e., as facts) but the framework does not exclude intensional manner (i.e., theories) as evidence. A program P is a solution to the ILP problem if it covers all positive examples ($B \cup P \models E^+$, *posterior sufficiency or completeness*) and does not cover any negative examples ($B \cup P \not\models E^-$, *posterior satisfiability or consistency*).

An atom g is a common generalization of atoms a and b if and only if there exist substitutions θ and σ such that $a = g\theta$ and $b = g\sigma$. A clause G is a common generalization of clauses C and D if and only if there exists a substitution θ such that $G\theta \subseteq C$ and $G\theta \subseteq D$. These definitions can be extended in the obvious way to sets of atoms and clauses.

Let Σ be a set of *function symbols* (or functors) together with their arity[1] and let $\mathcal{X}$ be a countably infinite set of *variables*. Then $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of *terms* built from Σ and $\mathcal{X}$. The set of variables occurring in a term t is denoted $Var(t)$. This notation naturally extends to other syntactic objects (like clause, literal, ...). A term t is a *ground term* if $Var(t) = \emptyset$. An *occurrence* u in a term t is represented by a sequence of natural numbers. $O(t)$ and $\bar{O}(t)$ denote the *set of occurrences* and *non-variable occurrences* of t, respectively. $t_{|u}$ denotes the *subterm* of t at the occurrence u and $t[t']_u$ denotes the *replacement* of the subterm of t at the occurrence u by the term t'.

An equation is an expression of the form $l = r$ where l and r are terms. l is called the left hand side (lhs) of the equation and r is the right hand side (rhs). An equational theory $\mathcal{E}$ (which we call *program*) is a finite set of equational clauses of the form $l = r \Leftarrow e_1, \ldots, e_n$. with $n \geq 0$ where e_i is an equation, $1 \leq i \leq n$.

The theory (and the clauses) are called *conditional* if $n > 0$ and *unconditional* if $n = 0$. An equational theory can also be viewed as a (Conditional) Term Rewriting System (CTRS) since the equation in the head is implicitely oriented from left to right and the literals e_i in the body are ordinary non-oriented equations. Given a (C)TRS $\mathcal{R}$, $t \to_{\mathcal{R}} s$ is a rewrite step if there exists an ocurrence u of t, a rule $l = r \in \mathcal{R}$ and a substitution θ with $t_{|u} = \theta(l)$ and $s = t[\theta(r)]_u$. A term t is said to be in *normal form* w.r.t. $\mathcal{R}$ if there is no term t' with $t \to_{\mathcal{R}} t'$. We say that an equation $t = s$ is normalized w.r.t. $\mathcal{R}$ if t and s are in normal form. $\mathcal{R}$ is said to be *canonical* if the binary one-step rewriting relation $\to_{\mathcal{R}}$ is terminating (there is no infinite chain $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} s_3 \to_{\mathcal{R}} \ldots$) and confluent ($\forall\, s_1, s_2, s_3 \in \mathcal{T}(\Sigma, \mathcal{X})$ such that $s_1 \to^*_{\mathcal{R}} s_2$ and $s_1 \to^*_{\mathcal{R}} s_3$, $\exists\, s \in \mathcal{T}(\Sigma, \mathcal{X})$ such that $s_2 \to^*_{\mathcal{R}} s$ and $s_3 \to^*_{\mathcal{R}} s$).

An $\mathcal{E}$-unification algorithm defines a procedure for solving an equation $t = s$ within the theory $\mathcal{E}$. Narrowing is a sound and complete method for solving equations w.r.t. canonical programs. Given a program P, a term t *narrows* into a term t' (in symbols $t \hookrightarrow_P t'$) iff $u \in \bar{O}(t)$, $l = r$ is a new variant of a rule from P, $\theta = mgu(t_{|u}, l)$ and $t' = \theta(t[r]_u)$. We write $t \hookrightarrow^n_P t'$ if t narrows into t' in n narrowing steps.

3 The IFLP framework

IFLP can be defined as the functional (or equational) extension to ILP. The goal is the inference of a theory (a functional logic program P) from evidence (a set of positive and optionally negative equations E) using a background knowledge theory (a functional logic program B). We will consider evidence composed of positive E^+ and negative E^- examples (equations) and their rhs which are normalized wrt the background theory B and the theory P which is intended to be discovered (hypothesis), $B \cup P$

[1] We assume that Σ contains at least one constant.

being canonical. E must always be consistent with B. By possibly modifying the evidence distribution, in any case, we can made some preprocessing to E: (i) any equation of the form $f(x_1, x_2, \ldots, x_n) = f(y_1, y_2, \ldots, y_n)$ is replaced by the n equations $x_1 = y_1$, $x_2 = y_2$, $\ldots$, $x_n = y_n$, and (ii) all redundant equations are eliminated.

3.1 Hypothesis Selection

As in ILP, we have to select "the optimal program" from all the many possible *valid* programs ensuring posterior sufficiency and satisfability. The problem is that there is no such thing as "the right hypothesis", so an optimality criterion must be arbitrarily selected depending on the application or purpose of the induction: prediction, scientific discovery, program synthesis, function invention, program transformation, abduction or explanation based learning.

Despite this undeniable fact, the **Minimum Description Length (MDL) principle** is the most popular selection criteria in ILP, which is supported by the classical view of unsupervised learning as compression and by the effectiveness of its use in many applications of machine learning. The MDL principle has been successfully applied mainly where the source has a statistical character and might contain errors. However, in other applications where no errors are expected from the source [7], like program synthesis from examples or, in *incremental* learning, the MDL principle sometimes fails. For our purposes we will compute the length of the equations as $length(e) = 1 + n_v/2 + n_c + n_f$ with n_v, n_c and n_f being the number of variables, constants and functors of the rhs of the rules only, because it is desirable to obtain short equations with diminishing character. Note that we promote variables over constants or functors. Finally, we define the length factor of a set of equations P as $\mathbf{LenF}(P) = -\sum_{e \in P} log_2 length(e)$.

In this paper, we take up the classical concept of **coherence** of scientific theories [3] used as a selection criterion in some applications of machine learning, especially explanatory reasoning or abductive inference. The idea of intrinsical coherence of a description can be adapted to the case of functional logic programs in many slightly different ways. We present just one of these ways. The consilience factor of a functional logic program P w.r.t. some given examples E^+ can be computed effectively as

$$\mathbf{ConF}(P) = \begin{cases} 1 & \text{if } P \text{ has only an equation} \\ 1 - \max(card(e \in E^+ : P_i \subset P \wedge P_i \models e)/card(E^+)) & \text{otherwise} \end{cases}$$

In those cases where the data are approximate or noisy, it is interesting to compute a **covering factor** w.r.t. the positive evidence, defined simply as $\mathbf{CovF}^+(P) = card(e \in E^+ : P \models e)/card(E^+)$ i.e., the proportion of positive cases covered. $CovF^-$ can be defined in the same way.

The idea of "the best hypothesis" only makes sense in the light of all these criteria given the purpose of the application.

3.2 Hypothesis Generation and Heuristics

For the present paper, we will consider the data to be perfect (no transmission errors) and we will be especially interested in program synthesis of only a concept at a time, so, for the moment, the stop criterion consists only of the completeness condition $CovF^+ = 1$ and a threshold for the consilience factor, usually 0.5. However, since consilience is favoured by short programs and a length factor is considered in the search heuristics, the syntactical length criterion is implicitly present.

The search is initially bottom-up, but this is not definitive, because it works with populations of programs and "merges" them using inverse narrowing. A rating is made from this population according to an optimality value, in a way which resembles genetic programming.

Concretely, our optimality measure is constructed simply as: $Opt(P) = LenF(P) + CovF^+(P) + ConF(P)$. These combinated heuristics considerably reduce the size of the sample necessary to induce the *intended hypothesis* over other approaches exclusively based on the MDL principle. Once the hypotheses selection criteria are settled, they are used as search heuristic along with the stop criterion selected. This makes our approach very generic and easily adaptable to quite different applications.

4 Non-incremental Algorithm for IFLP

In this section, we discuss the skeleton of the algorithm for the inference of functional logic programs. Our learning task consists of a search of hypothetical equations and a selection of programs constructed from these equations, until one of the programs is evaluated as a good solution.

In the case of functional logic programs, we cannot start from the most general program because the examples are equations, and the most general program $X = Y$ would not make the program finite nor confluent. The most specific generalization in this case is the program itself. In contrast, our approach starts from almost all possible generalizations, with a very small and reasonable restriction:

Definition 1 Restricted Generalization (RG)
Given an equation $e \equiv \{t = s\}$*, the equation* $t' = s'$ *is a restricted generalization of* e *if it is a generalization of* e *(i.e.* $\exists \theta : t'\theta = t \wedge s'\theta = s$*) such that* $\forall x (x \in Var(s') \Rightarrow x \in Var(t'))$*.*

In other words, RG does not introduce extra variables on the rhs of the equations. Since we have to ensure posterior satisfiability, we begin generating all possible restricted generalizations from each positive example consistent with the evidence. More formally,

Definition 2 Consistent Restricted Generalization CRG
An equation $e = \{l_1 = r_1\}$ *is a consistent restricted generalization (CRG) w.r.t.* E^+ *and* E^- *and an existing theory* $T = B \cup P$ *if and only if* e *is a RG for some equation of* E^+ *(always oriented from left to right) and there does not exist: (1) a narrowing chain using* e *and* T *that yields some equation of* E^-*, and (2) a narrowing chain using* e *and* T *that yields a different normal form for some lhs different from the rhs which appeared in the equations of* E^+*.*

Despite the fact that we use CRG's, our algorithm is not strictly a generalization algorithm because we work with sets of equations and programs instead of refining a single program.

Straightforwardly, since narrowing is a sound and complete method for $\mathcal{E}$-unification, we will study an inverse method of it that we will call *inverse narrowing*. Let us illustrate the concept with an example.

Example 1 *Given a program* P*, suppose we select the clause* $\{X' + 0 = X'\}$ *and the rhs of another clause* $\{X + s(0) = s(X)\}$*, i.e.,* $s(X)$*. The first rule can be used inversely in the second term in different positions. In this case, there are different possible applications which are variable or non-variable:* $t_1 = s(X + 0)$ *and* $t_2 = s(X) + 0$*. That is to say,* t_1 *and* t_2 *can be narrowed to* $s(X)$ *using a rule of* P*. The resulting equations are* $X + s(0) = s(X + 0)$ *and* $X + s(0) = s(X) + 0$*.*

Definition 3 Inverse Narrowing
Given a functional logic program P*, we say that a term* t *conversely narrows into a term* t'*, and we write* $t \hookleftarrow_P t'$*, iff* $u \in O(t)$*,* $l = r$ *is a new variant of a rule from* P*,* $\theta = mgu(t_{|u}, r)$ *and* $t' = \theta(t[l]_u)$*. The relation* $\hookleftarrow_P$ *is called the inverse narrowing relation.*

As we have already mentioned, we start the inductive process from positive and negative evidence E^+ and E^-. Additionally a background theory B can be used to induce the target program P. In the following, we will denote BF (*B*asic *F*unctions) the subset of functions from B, determined by the user, which can be used in the definition of the learned functions of P. For the sake of efficiency, the IFLP algorithm is also parametrized by three more input parameters: 1) *min* indicates the limit of CRG's that must be generated from one example at each algorithm step; 2) *step* is a measure that indicates the increase of the *min* parameter (this is done when no solution program is found using the current *min* value), and 3) *inarcomb* shows the limit of inverse narrowing steps that can be carried out with a pair of programs.

The basic IFLP algorithm works with a set of equations (we denote EH, *E*quation *H*ypothesis) where the equations are mainly generated by means of CRG, and a set of programs (we denote PH, *P*rogram *H*ypothesis) composed exclusively from equations of EH. At each step of the algorithm, new equations and programs are generated by inverse narrowing. Thus, the kernel of the algorithm is constituted by two auxiliar procedures: *GenerateCRG* and *InverseNarrowing*.

The **Procedure GenerateCRG(input:**E^+, E^-, EH, min**; output:**EH_f**)** returns the set EH_f which is obtained by adding to EH the set of equations which are CRG's wrt E^+ and E^- and which are constructed from each equation in E^+. Also, the optimality of each equation is computed as well as the number of examples which are covered by it. The size of EH_f is limited by the *min* value.

The **Procedure InverseNarrowing(input:**$P_1, P_2, BF, inarcomb$**;output:**EH, PH**)** returns a set of equations (EH) and a set of programs (PH) obtained in the following way: first, inverse narrowing is applied between equations of the two input programs (up to *inarcomb* number of combinations) and, then, the sets are pruned to eliminate redundancy and inconsistency.

The first step of the learning algorithm generates the initial EH set, with all the CRG from E^+. Next, PH is initialized to a set of programs containing only one equation from EH. Then, at each iteration RH and PH are recalculated until a program P is found which covers E^+ and whose $ConF$ factor is better than a certain desired consilience value (that we call dc). At every step, the theory B is only used if there is no program in PH which covers some example with an acceptable optimality Op.

Finally, we would like to note that the parameters $dc, min, step, Op$ and $inarcomb$ are heuristical. Therefore, they must be estimated depending on several factors (like the complexity of the theory B, the expected complexity of P, the number of examples, etc.). Our experiments demonstrate good performances of the algorithm when the following values are used: $dc = 0.5$, $min = 2-3$, $step = 2-3$, $Op = 0$ and $inarcomb = 3$. Some of them can be modified if no solution is found (for instance, the $inarcomb$ parameter can be increased).

Next, we outline the IFLP algorithm.

Input: $E^+, E^-, B, BF, dc, min, step, inarcomb$. **Output:** a program $P = BestSolution$
begin

Let $EH = \emptyset$ and let $PH = \emptyset$
GenerateCRG(input:$E^+, E^-, \emptyset$; output: EH)
Let $BestSolution = Select\_best(PH)$
while not $stop\_criterion(BestSolution)$ **do**

if using B {using background knowledge} and
$\exists E' \subseteq E^+$ and $\nexists P \in PH \mid Opt(P) \geq Op$
then begin

for each $e \in E'$ **do**

Let $P = \{e\}$
$InverseNarrowing$(input:P, B, BF;output:EH', PH')
$Update\_all(BestSolution, EH, PH, EH', PH')$

endfor

endbegin
endif { using background knowledge}
{General case. Select the most weighted pair of programs P_1, P_2 from PH}
Let $n = card(E^+)$
while $n > 0$ **do**

$PP = \{(P_1, P_2) \mid P_1, P_2 \in PH, P_1 \neq P_2$ s.t. $card(\{e \in E^+ \mid P_1 \models e \vee P_2 \models e\}) \geq n\}$
if $PP \neq \emptyset$
then let $(P_1, P_2) = argmin_{PP}(Opt(P_1) + Opt(P_2)$ and **break while**
else let $n = n - 1$
endif

endwhile
if $n = 0$ **then begin**

let $min = min + step$
$GenerateCRG$(input:E^+, E^-, EH, min; output: EH')
if $EH' = EH$ **then halt** {No more programs to essay. No solution.}

endbegin
else begin

$InverseNarrowing$(input:$P_1, P_2, \emptyset$;output:EH', PH')
$Update\_all(BestSolution, EH, PH, EH', PH')$

endbegin
endif

endwhile

endalgorithm

where: $Select\_best(PH)$ selects the program with the best covering, the greatest consilience and, finally, the best optimality and $Update\_all(S, E, P, E', P')$ makes the following actions: let $E = E \cup E'$, let $P = P \cup P'$ and $S = Select\_best(P)$

The following example illustrates the use of the algorithm for a typical problem: the induction of the function *append*.

Example 2 *To shorten the trace, the following parameters are selected:* $min = 2$, $step = 2$, $inarcomb = 3$. *The stop-criterion is settled at consilience* $> dc = 0.5$. *Using Prolog notation for lists, the evidence is as follows:*

(E_1^+)	$append([1,2],[3]) = [1,2,3]$	(E_1^-)	$append([3],[4]) = [4,3]$
(E_2^+)	$append([c],[a]) = [c,a]$	(E_2^-)	$append([1,2],[]) = [1]$
(E_3^+)	$append([],[4]) = [4]$	(E_3^-)	$append([1,2,3],[4]) = [1,2,3,4,5]$
(E_4^+)	$append([a,b],[]) = [a,b]$	(E_4^-)	$append([],[a,b]) = [b,a]$
(E_5^+)	$append([a,b,c],[d,e]) = [a,b,c,d,e]$		

Since min = 2 we generate only the two CRG with the best optimality from each example.
The first EH and PH are composed of 10 equations and the corresponding 10 programs. The first BestSolution covering all the examples can be constructed from 4 equations with consilience = 0.2 and optimality = - 5.7. Next we begin the inverse narrowing combinations. Since there is no pair of programs covering 5 or 4 examples, with $n = 3$ *we find* $P_1 = \{append(.(X,.(Y,[])), Z) = .(X,.(Y,Z))\}$, *covering* $\{E_1^+, E_4^+\}$ *and optimality = - 0.76 and* $P_2 = \{append([], X) = X\}$, *covering* E_3^+ *and optimality = + 0.62. We have 3 possible inverse narrowing combinations (which is just equal to inarcomb), all using* $e_1 = \{append(.(X,.(Y,[])), Z) = .(X,.(Y,Z))\}$ *and* $e_2 = \{append([], X) = X\}$, *giving three consistent programs, which are added to PH:*

$$P_a = \{append(.(X,.(Y,W)), Z) = .(append(W,X),.(Y,Z)), append([], X) = X\}$$
$$P_b = \{append(.(X,.(Y,W)), Z) = .(X,.(append(W,Y),Z)), append([], X) = X\}$$
$$P_c = \{append(.(X,.(Y,W)), Z) = .(X,.(Y, append(W,Z))), append([], X) = X\}$$

In the same way, the second EH and PH are computed with 3 more equations and programs, respectively. Now, there is no pair of programs covering 5 examples. With $n = 4$ *we find two programs* $P_1 = \{append(.(X,.(Y,W)), Z) = .(append(W,X),.(Y,Z)), append([], X) = X\}$ *covering* $\{E_1^+, E_3^+, E_4^+\}$ *and* $P_2 = \{append(.(X,[]), Y) = .(X,Y)\}$ *covering* $\{E_2^+\}$. *We select the two rules with higher optimaly, i.e.,* $\{append([], X) = X\}$ *and* $\{append(.(X,[]), Y) = .(X,Y)\}$ *which generate some new programs by inverse narrowing. Most of them are inconsistent, others are not confluent and then splitted into inconsistent programs. Finally, only one of them results in a consistent and confluent program:*

$$P_d = \{append(.(X,Z), Y) = .(X, append(Z,Y)), append([], X) = X\}$$

which covers all E^+ *and has optimality = -2.7. A fourth combination could be made but the value of inarcomb* $= 3$ *forces the exit from the procedure InverseNarrowing. Since* P_d *covers all the examples, it is consistent and has consilience* > 0.5, *the algorithm stops and outputs* P_d.

Finally, it is straightforward to prove the following correctness theorem for the learning algorithm.

Theorem 1 *Given an evidence* E^+, E^- *and a background theory* B, *if a program* P *is a solution of the IFLP algorithm then it is canonical and* $B \cup P \models E^+$ *and* $B \cup P \not\models E^-$.

5 Incremental Version

In general, incremental learning is necessary when the number of examples is large and presented one by one. Here are two new phenomena: the hypothesis cannot be absolutely validated since any new example can make it inconsistent, and, the goal is to obtain "the intended hypothesis" *the sooner the better* and not in the limit[2]. In this case, the algorithm can always present a selection of the k best programs and interacts with the user.

With all this in mind, an 'operative' adaptation of the preceding algorithm to the incremental case is straightforward, using a memory M to store the presented evidence so far. With the first positive example, the algorithm behaves exactly as in the non-incremental case, although it may be preferable to wait for some examples to start the algorithm. For each new example presented, we work as follows:

- If it is a positive example: E_n^+, we check for every program $P_i \in PH$:
  1. HIT: if it is correctly covered by P_i we recompute its new consilience and covering factor (we can use the old values for it). Eventually, the consilience factor may decrease.
  2. UNCOVERED: if it is not covered by P_i but it remains consistent (still confluent because there is no narrowing chain for the lhs of E_n^+), we recompute the optimality factor as in the HIT case.
  3. ANOMALY: if it is covered *erroneously* by P_i we remove P_i from PH.

  and we generate all the CRG's for it into RH and the corresponding unary programs in PH.
- If it is a negative example: E_n^-, we check the consistency for every program $P_i \in PH$ and we act in the same way as in either the UNCOVERED case or as in the ANOMALY case.

In any case, if the best program does not comply with the stop-criterion, the algorithm of the previous section is 'reactivated' until a new program makes the stop-criterion true again.

[2]Here the consilience factor is more appropriate because it is less conservative than MDL for perfect data.

Example 3 *Now, we present an example to see the adaptation of the algorithm to incremental learning and also to illustrate the use of background knowledge.*

Let us consider the inference of the power function from the product function, which consists of $B = \{0 \times X = 0, sX \times Y = X \times Y + Y, X + 0 = X, X + s(Y) = s(X + Y)\}$ *and from the following evidence:*

(E_1^+)	$ss0 \uparrow ss0 = ssss0$	(E_1^-)	$ss0 \uparrow sss0 = sssssssss0$
(E_2^+)	$sss0 \uparrow ss0 = sssssssss0$	(E_2^-)	$sss0 \uparrow sss0 = sssssssss0$
(E_3^+)	$sss0 \uparrow s0 = sss0$	(E_3^-)	$ss0 \uparrow sss0 = ssss0$
(E_4^+)	$0 \uparrow sss0 = 0$	(E_4^-)	$sss0 \uparrow ss0 = ssss0$
(E_5^+)	$ss0 \uparrow 0 = s0$	(E_5^-)	$0 \uparrow s0 = s0$
(E_6^+)	$ssss0 \uparrow 0 = s0$	(E_6^-)	$s0 \uparrow 0 = 0$

The examples are given one by one in this order: (E_1^+),(E_1^-),(E_2^+), (E_2^-), (E_3^+), (E_3^-), (E_4^+), (E_4^-), (E_5^+), (E_5^-), (E_6^+), (E_6^-). *The additional inputs of the algorithm are* $BF = \{\times\}$ *and* $Op = 0$*, suggesting the use of the functor* $\times$ *from* B*, but not* $+$. *After an interactive learning session with just 5 positive and 4 negative examples, the following program for exponentiation is induced by the algorithm:* $\{X \uparrow sY = (X \uparrow Y) \times X,\ X \uparrow 0 = s0\}$

6 Future Work

In incremental learning, conditions are a powerful tool for making inconsilient programs (modifying the previous hypothesis by adding the new anomaly as a negated condition) if syntactic length is the prevailing criterion. Hence, if functional logic programs have advantages over functional ones, we have to introduce conditions only when necessary provided the program is shortened and consilience is preserved (or increased). Also there are other restrictions, depending on the kind of conditional narrowing (e.g. simple conditional narrowing does not allow extra variables in conditions).

The power of higher-order languages for induction of theories from facts has not been fully exploited so far. The issue here is that if higher-order unification is difficult and deduction very problematic, what can be expected from a much harder problem like induction? However, there are reasons to think that new possibilities are open. In this way, the first steps towards Higher-Order Induction are being taken in [1]. An intended higher-order inverse narrowing first requires the choice of a proper "higher-order narrowing" from some higher-order unification methods which have been presented to date.

7 Conclusions

We have presented a general framework for the Induction of Functional Logic Programs as an extension of ILP, including a discussion of selection criteria for equational theories and an algorithm that is guided by an adaptable optimality factor based on these criteria. The kernel of the algorithm is an inverse narrowing procedure which is used for the induction of equational clauses. Our approach is quite generic and powerful enough to be adapted to different tasks: program synthesis, abduction, explanation-based learning (EBL) and prediction.

References

[1] A.F. Bowers, C. Giraud-Carrier, C. Kennedy, J.W. Lloyd, and R. Mackinney-Romero. A framework for Higher-Order Inductive Machine Learning. In *Representation issues in reasoning and learning.* Area meeting of CompulogNet Area 'Computational Logic and Machine Learning, 1997.

[2] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19-20:583–628, 1994.

[3] J.H. Holland, K.J. Holyoak, R.E. Nisbett, and P.R. Thagard. *Induction. Processes of Inference, Learning, and Discovery.* The MIT Press, 1989.

[4] J.M. Hullot. Canonical Forms and Unification. In *5th Int'l Conf. on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag, Berlin, 1980.

[5] H. Hussmann. Unification in conditional-equational theories. Technical report, Fakultät für Mathematik und Informatik, Universität Passau, 1986.

[6] J.W. Klop. Term Rewriting Systems. *Handbook of Logic in Computer Science*, I:1–112, 1992.

[7] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* 2nd Ed. Springer-Verlag, 1997.

[8] J.W. Lloyd. Programming in an integrated functional and logic language. *to appear in The Journal of Functional and logic Programming.*

[9] S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.