Inverse Narrowing for the Induction of Functional Logic Programs*

J. Hernandez-Orallo [§] M.J. Ramirez-Quintana[§]

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

Abstract

We present a framework for the Induction of Functional Logic Programs (IFLP) from facts. This can be seen as an extension to the now consolidated field of Inductive Logic Programming (ILP). Inspired in the inverse resolution operator of ILP, we study the reversal of narrowing, the more 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. At the end of this paper, we comment on the plausibility of extending the presented techniques to higher-order induction and its appropriateness for function invention, a topic which is difficult to incorporate homogeneously with the basic first-order inductive rules of inference in ILP.

Keywords: Functional Logic Programming, Inductive Logic Programming.

1 Introduction

1.1 Precedents

Since the beginning of this decade, Inductive Logic Programming (ILP) has been a very important area of research as an appropriate framework for the inductive inference of first-order clausal theories from facts. ILP was unified around the works of Muggleton and the new name ILP [23]. However, the ideas are much older and can be dated to the works of Plotkin [25, 26] and Shapiro [31]. Muggleton [23] defined ILP as the intersection of inductive learning and logic programming. As a machine learning paradigm, the general aim of ILP is to develop tools, theories and techniques to induce hypotheses from examples and background knowledge. ILP inherits the representational formalism, the semantical orientation and, the well-established techniques of logic programming,. From a proof theory point of view, ILP can also be considered as the dual paradigm of Logic Programming (LP): whereas ILP describes the process of the induction of logic programs from logic formulae, LP deals with the deduction of logic formulae from logic programs provided by the user, i.e., the induction can be thought of as the inverse process to deduction. Therefore, inductive inference rules can be obtained by inverting deductive ones. Several approaches corresponding to different assumptions about the deductive rule and the format of background theory and examples have been proposed and investigated [7, 25, 36]. The most interesting is based on the inversion of the resolution principle [29, 34, 38]. Although inverse resolution has been proposed in [22] as an inference system which consists of four rules (Absorption, Identification and two rules for the introduction of new predicate symbols), more specific forms of these have been implemented by having a two-stage operation: first, inverse resolution operators are applied to examples, and then clauses are reduced by generalization. As we will show, our proposal induces equational clauses in a way which is quite similar.

Inside the general framework of learning and induction, the importance of ILP may be justified for many reasons. First, one of the advantages of ILP is the ability to use background knowledge and the understandability of theories, differing radically from other novel approaches like fuzzy systems or neural networks. Second, ILP is a more tractable and natural framework for many problems

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

and has all the hypothesis validation efficiency of SLD-resolution. Third, it is easier to state formal considerations about the hypotheses, the evidence and their relationship.

ILP has provided an outstanding advantage in the inductive machine learning field by increasing the applicability of learning systems to theories with more expressive power than propositional frameworks. 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.

There are some previous works on the learning of functions and the induction of functional programs, usually combining very different techniques (evolutionary programming, MDL principle, folding) to synthesize recursive Standard ML programs as in [24], flattening the logic program with function symbols to transform it into an equivalent program without functions [30], using extensions of the least general generalization technique used in ILP (see for instance [1, 16, 20]) or a similar approach to Shapiros MIS [31] for inducing term rewriting systems [35]. Also, there are some early works on the induction of LISP programs in the seventies (see [33] for a survey), although we do not consider them to be very related to our approach because they learn from execution traces.

Over the last decade, it has been theoretically and experimentally demonstrated that *functional logic languages have more expressive power in comparison to functional languages and a better operational behavior in comparison to logic languages* [11]. One relevant approach [13, 17] to integration is the functional logic programming where the 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 [32] (i.e., general unification w.r.t. an equational theory $\mathcal{E}$) requiring a complete $\mathcal{E}$-unification procedure to determine whether two terms, t and s, are equal under an equational theory. A sound and complete $\mathcal{E}$-unification method is narrowing[1] [14, 15, 28]. Several strategies have been proposed in order to improve the efficiency of the narrowing algorithm (basic [14], innermost [10], ...).

The induction of functional logic programs has been discussed in [2, 3], but as a restricted variant of logic programs such that each n-ary predicate can be associated to a total function as follows: m of its arguments are labeled as input, while the remaining n-m are labeled as output, and for every given sequence of input values, there is one and only one sequence of output values that makes the predicate true. The programs are not in any other way functional. In addition, a framework for the induction of Escher programs is presented in [6]. Escher [21] 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 rewrite mechanism. Since functions operate on data types with several data constructors, the proposed algorithm chooses one of its arguments as a 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. In contrast, our approach does not consider pattern scheme and it is oriented to working with languages which are based on narrowing and are not typed.

1.2 IFLP Motivations

In this work, we present a general framework for the induction of functional logic programs (IFLP) from examples, generalizing the scope of the 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 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. The selection criteria is based on the number of positive examples that can be derived from the equation. Then, if the generated equation covers more positive examples or subsumes other equations, then it is added to the previously selected equations (probably replacing some of them). This process is repeated until a program (or set of equations) is valid according to some selection criteria. One of the main differences between our approach and ILP is the meaning of 'concise' program. ILP looks

[1] The completeness of this algorithm requires some additional requirements on equations, especially on conditional equations like the absence of extra variables in conditions.

for the shortest program [8, 19, 23] whereas we consider programs that are *consilient* [12]: a program is consilient if it has not exceptions, i.e. it does not 'discriminate' a part of its consequences, in the sense that these consequences are outside the scope of the *main* rule.

Since an alternative approach to implementing functional logic languages via SLD-resolution is based on the flattening of the program and the goal[2] [4, 5, 37], we could first apply such a flattening and then use the well-studied techniques of ILP, for the induction of functional logic programs. Indeed, Left-to-right SLD-resolution combined with flattening is equivalent to leftmost innermost basic narrowing [4]. However, the flattening approach is limited as [11] points out: *it is not ensured that functional expressions are reduced in a purely deterministic way if all arguments of a function are ground values. This important property of functional languages is not preserved since the information about functional dependencies is lost by flattening. Moreover, flattening restricts the chance to detect deterministic computations by the dynamic cut which is relevant especially in the presence of conditional equations.* The result of flattening plus resolution in these cases is the appearance of infinite loops or duplicity of solutions, which did not exist in the original functional logic version.

Finally, in our opinion, another important reason for undertaking the jump to IFLP is that once the properties and behaviour of different inverted narrowing techniques are established, the step to higher order induction may be easier to bridge based on the deductive higher-order counterparts [9, 27].

The work is organized as follows. In Section 2, we recall the main concepts of ILP and we formalize the narrowing semantics we focus on. Section 3 presents the general IFLP framework and gives the overall strategy for searching the program space. The search is guided by measures of program quality just as its consilience and the length of the right hand side of the rules. Section 4 presents a non-incremental version of the algorithm which computes a solution program from the examples, including the definition of the inverse narrowing procedure. An example of the application of the algorithm is also included. The extension for including incrementality is tackled in Section 5. In Section 6, we discuss the step for dealing with conditions and the plausibility of higher-order induction. Finally, Section 7 concludes the paper.

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 [11, 18, 23].

2.1 Inductive Logic Programming

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.

2.2 Semantics of Functional Logic Programs

Let Σ be a set of *function symbols* (or functors) together with their arity[3] 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

[2]This flattening eliminates the nesting functional in the head of the rules of a conditional term rewriting system preserving the body of the clauses, along with their function symbols; whereas the flattening procedure in [30] (above mentioned) eliminates all the function symbols, converting each function term of arity n into a new predicate of arity n + 1.

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

objects (like clause, literal, ...). A term t is a *ground term* if $Var(t) = \emptyset$. A *substitution* is defined as an almost identical mapping from the set of variables $\mathcal{X}$ into the set of terms $\mathcal{T}(\Sigma, \mathcal{X})$. 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 implicitly 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 $t \overset{u,l=r,\theta}{\hookrightarrow}_P t'$[4]) 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 of ILP[5]. 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).

3.1 Sample Presentation and Other Assumptions

We will consider evidence composed of positive E^+ and negative E^- equations[6] and their rhs which are normalized wrt the background theory B and the theory P which is meant to be discovered (hypothesis), with $B \cup P$ being canonical. E must always be consistent with B. We can do some preprocessing to E, which includes well known equation simplifications: (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 $x_1 = y_1$, $x_2 = y_2$, ..., $x_n = y_n$, and (ii) the elimination of all redundant equations. This is illustrated in the following example:

Example 1 *: Consider the following background theory* $B = \{s(X) < s(Y) = X < Y,\quad 0 < s(Y) = true,\quad X < 0 = false\}$ *along with the incomplete positive and negative evidence* E^+ *and* E^-*:*

		(E^-_1)	$s(0) + 0 = 0$
(E^+_1)	$0 + 0 = 0$	(E^-_2)	$0 + 0 = s(0)$
(E^+_2)	$s(0) + s(0) = s(s(0))$	(E^-_3)	$s(0) + s(0) = s(0)$
(E^+_3)	$0 + s(0) = s(0)$	(E^-_4)	$s(0) + 0 = s(s(0))$
(E^+_4)	$s(s(0) + s(0))) = s(s(0))$	(E^-_5)	$s(0 + 0) = s(s(0))$
(E^+_5)	$s(s(s(0)) + s(0))) = s(s(s(s(0))))$	(E^-_6)	$s(s(0) + s(0)) = 0$
		(E^-_7)	$s(0) = 0$

[4]Or simply $t \overset{l=r,\theta}{\hookrightarrow}_P t'$ or $t \overset{\theta}{\hookrightarrow}_P t'$ if the occurrence or the rule is clear from the context. Also, the subscript P will usually be dropped when clear from the context.

[5]It is obvious that any problem expressed in the ILP framework can also be expressed in the IFLP framework, because all the positive facts e^+_i of an ILP problem can be converted into equations of the form $e^+_i = true$ and all the negative facts e^-_j can be expressed as $e^-_j = false$.

[6]If only boolean functions were to be induced, no negative sample would be required for IFLP. For general functions, however, it may be very useful to also have a negative sample.

As we have just said, example E_4^+ can be simplified into $s(0)+s(0)=s(0)$, because (by (i)) E_4^+ is equal to E_2^+ and hence one of them (e.g. E_4^+) can be eliminated (by (ii)). In the same way, the last positive example is simplified into $s(s(0))+s(0))=s(s(s(0)))$ and the negative example E_5^- is removed because it is redundant with E_2^-.

3.2 Hypothesis Selection

As in ILP, we have to select "the optimal program" from all the many possible *valid* programs ensuring posterior sufficiency and satisfiability. 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 (EBL). Moreover, some of them drastically diverge for different kinds of samples: (perfect / imperfect), (complete / incomplete) and (positive evidence only / positive and negative evidence).

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, by the effectiveness of its use in many applications of machine learning, and by its recent formal justification [19] in relation to Bayesian learning. The MDL principle has been successfully applied mainly where the source has a statistical character and it might contain errors. However, in other applications where no errors are expected from the source [19], 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, respectively, the number of variables, constants and functors of the rhs of the rules only, because it is desirable to obtain short equations with decreasing character[7]. Note that we promote variables over constants or functors. Finally, we define the length factor of a set of equations P as $\mathbf{LenF}(P)=-\sum_{e\in P} log_2 length(e)$.

However, there are other selection criteria. The so-called **subset-principle**, with Plotkin's least general generalization (lgg) [25] being its concretization for logic programs, means that the hypothesis must cover the smallest superset of the sample data.

In this paper, we take up the classical concept of **coherence** of scientific theories [12] 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 two of these ways. The first one deals with the concept of separation, i.e, the facts can be independently covered by parts of the programs. More formally, a program P is n-separable in a partition $\{P_1,P_2,\ldots,P_n\}$ from P such that $P_i \not\subseteq P_j \quad \forall i\neq j$ iff for every equation e such that $P\models e$ there exists a P_i such that $P_i\models e$. Otherwise, P is said to be robust iff it has no 2-separation. In order to give a more gradual value (a factor) of coherence, we introduce a related but different concept: the consilience factor of a functional logic program P wrt some given examples E^+, which can be computed effectively as

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

Also, in some cases, like abductive or explanatory learning, the consilience factor should be computed jointly with the background theory, i.e. $\mathrm{ConF}(P\cup B)$.

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.

Finally, the **efficiency** of a program is a very interesting criterion for program synthesis. Computing the number of narrowing steps is a good approximation for it. Thus, we define the efficiency of a program P as $\mathbf{EffF}(P)=\sum_{l=r\in E^+} n, l\hookrightarrow_P^n r$.

To illustrate the divergence of these criteria, let us consider some consistent ($CovF^-=0$) and complete ($CovF^+=1$) programs with Example 1:

$P_1=E^+$
$P_2=\{X+0=X, 0+X=X, s(X)+s(0)=s(s(X))\}$
$P_3=\{X+0=X, X+s(Y)=s(X+Y)\}$

[7] Theoretically, the length should be computed using a generating grammar for all the terms in the Herbrand Universe as in [8]. In our case, we give an approximation that works well in practice and has the advantage of being independent of the total number of constants, variables and functors in the program or the Herbrand Universe.

$P_4 = \{X + 0 = X, X + s(0) = s(X)\}$
$P_5 = \{X + Y = X : -Y = 0, X + s(Y) = s(X) : -Y = 0\}$
$P_6 = \{X + Y = Y + X : -X < Y, X + 0 = X, s(X) + s(Y) = s(s(X + Y))\}$
$P_7 = \{X + 0 = X, 0 + X = X, s(X) + s(Y) = s(s(X + Y))\}$
$P_8 = \{X + 0 = X, 0 + X = X, s(X) + s(Y) = s(X + s(Y)))\}$

According to the criteria we have just presented, P_4 is the shortest one, followed by P_3. According to the subset principle, P_4 is also better than P_3, but P_1 is the best hypothesis. However, P_1 is clearly separable, along with P_2, P_4, P_5, P_7, P_8 whereas P_3 and P_6 are robust. Both P_3 and P_4 have the greatest consilience factor 0.75. Finally, in most cases, P_7 will be more efficient than P_3 and always more efficient than P_8.

In the light of this example, it even seems arbitrary to select P_3 as the "right hypothesis". In fact, the idea of "the best hypothesis" only makes sense in the context of the purpose of the application.

3.3 Hypothesis Generation and Heuristics

For the present paper, we will consider the data to be perfect (no transmission errors) and we are especially interested in program synthesis of only one concept at a time, so, for the moment, the stop criterion consists only of the completeness condition $CovF^+ = 1$ and a threshold for the consilient 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. Also, efficiency is implicitly taken into account due to the character of the search.

As we will see in the next section, 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[8]: $Opt(P) = \alpha \times LenF(P) + \beta \times CovF^+(P) + \gamma \times ConF(P)$.

These combined heuristics considerably reduce the size of the sample which is necessary to induce the *intended hypothesis* over other approaches which are exclusively based on the MDL principle. Once the hypotheses selection criteria are settled, the algorithm drives their generation in a proper way, using the optimality criterion as a 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 form of the algorithm for the inductive inference of functional logic programs. As already pointed out, our learning task consists of an inductive search of hypothetical equations and a selection of programs constructed from these equations, until one of the programs is evaluated as a good solution.

For instance, in a logic program, given only the positive data $\{p(a, b, a, a), p(b, c, b, c), q(a, f(a), c), q(b, f(b), c)\}$ we can compute the most general program $\{p(X, Y, Z, W), q(X, Y, Z, W)\}$ and refine it by specialization. Alternatively, we can begin from the positive data as a program and proceed by generalization.

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 of the sample equations, 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. RG prevents meaningless generalizations from being taken into consideration when we search the intended program for the given examples.

[8]For the examples we will consider just $\alpha = 1$, $\beta = 1$, $\gamma = 1$, but in the future these values could be parametrised for different kind of problems.

Since we have to ensure posterior satisfiability, we begin generating all possible restricted generalizations from each positive example which is consistent with both the positive and negative examples. More formally,

Definition 2 Consistent Restricted Generalization CRG
An equation $e = \{l_1 = r_1\}$ is a consistent restricted generalization (CRG) wrt 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 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^+.

Also, by this definition, trivial CRG's like $X = X$ are not allowed.

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 2 *Suppose we are inducing a program P from the positive examples in Example 1. At the n-th step, suppose we select the clause $\{X' + 0 = X'\}$ as good for P and we arbitrarily select 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) $s(X + 0)$
(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 $t \overset{u,l=r,\theta}{\hookleftarrow}_P t'$, 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.

The IFLP algorithm

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 (Basic Functions) 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 maximal number 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 (as we will see, *min* value must be increased when no program solution is found using the current *min* value), and 3) *inarcomb* shows the maximal number of inverse narrowing steps that can be carried out with a pair of programs. These parameters are provided in order to improve the efficiency and performances of the algorithm.

The basic IFLP algorithm learns programs by generating two sets of hypotheses: 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) which are 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 auxiliary 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 the generated set 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.

Procedure InverseNarrowing(input:$P_1, P_2, BF, inarcomb$**;output:**EH, PH**)**
Calculate the set $\{e_i\}_{i\in I}$ by applying inverse narrowing steps between a deterministically[9] selected equation e from P_1 (or P_2) and all possible equations e' from P_2 (or P_1). When $BF \neq \emptyset$ then $e \in P_1$ and $e' \in P_2$. Then we compute all the CRG's $e_{i_j}, j \in J$, from e_i
Let EH be $\cup_{j\in J, i\in I}\{e_{i_j}\}$
Let PH be $\{P_1 \cup P_2 \cup \{e_{i_j}\}/\{e\}\}_{j\in J, i\in I}$
Calculate coverings and optimalities of each set in PH
while some program p in PH is inconsistent or not confluent

Remove p from PH if p is not consistent.
Replace p by each of its confluent subsets, if p is not confluent.

endwhile
Clean the sets in PH removing redundant rules
endprocedure

The first step of the learning algorithm generates the initial EH set with all the CRG's from E^+. Next, PH is initialized to the set of all possible 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 must be 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, as well as the coefficients for $Opt(P)$. 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, ...). 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 for generating more programs).

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.}

[9]Beginning with the pair of equations with best optimality until $inarcomb = card(PH)$

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.
- $Update\_all(S, E, P, E', P')$ performs the following actions:
  - $E = E \cup E'$ and $P = P \cup P'$
  - $S = Select\_best(P)$

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

Example 3 *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_2^+) $append([c],[a]) = [c,a]$
(E_3^+) $append([],[4]) = [4]$
(E_4^+) $append([a,b],[]) = [a,b]$
(E_5^+) $append([a,b,c],[d,e]) = [a,b,c,d,e]$

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

Since min = 2, we generate only the two CRG's with best optimality from each example:

$CRG(E_1^+) = \{append(.(X,.(Y,[])),Z) = .(X,.(Y,Z)),$
$append(.(X,.(Y,Z)),.(W,Z)) = .(X,.(Y,.(W,Z)))\}$
$CRG(E_2^+) = \{append(.(X,[]),Y) = .(X,Y), append(.(X,Y),.(Z,Y)) = .(X,.(Z,Y))\}$
$CRG(E_3^+) = \{append([],X) = X, append(X,.(Y,X)) = .(Y,X)\}$
$CRG(E_4^+) = \{append(X,[]) = X, append(.(X,.(Y,Z)),Z) = .(X,.(Y,Z))\}$
$CRG(E_5^+) = \{append(.(Y,.(Z,.(W,V))),X) = .(Y,.(Z,.(W,X))),$
$append(.(Y,.(Z,.(W,[]))),X) = .(Y,.(Z,.(W,X)))\}$

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 split 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 between* $\{append(.(X,.(Y,W)),Z) = .(append(W,X),.(Y,Z))\}$ *and* $\{append(.(X,[]),Y) = .(X,Y)\}$ *giving some other new programs, 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

Since the complexity of induction depends mostly (and not polynomially) on the number of examples, any non-incremental algorithm would be useless for real problems. In general, incremental learning is necessary when the number of examples is infinite 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[10]. In this case, the algorithm can always present a selection of the k best programs or make guesses to the user when the optimality of the best one is high and much greater than the second one.

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 evidence presented up to this point. 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 (this is very efficient because we can use the old values for it). Eventually, the consilience factor may diminish.
  2. UNCOVERED: if it is not covered by P_i but 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 constructing 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 preceding section is 'reactivated' and works in the same way until a new program makes the stop-criterion true again.

When the number of examples increases, the memory gets too large to compute consistency of new generated programs using combinations (inverse narrowing). If the target program has many rules, both PH and EH may be huge. In both situations, the performance of the algorithm may decrease considerably. There is no easy solution for this, although some 'cautious' pruning could be done in M (forgetting), PH and EH using extra programs (along with the programs in PH) to describe intensionally (but with strict covering) the past evidence.

Example 4 *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 example of inducing 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 the following evidence:

(E_1^+) $ss0 \uparrow ss0 = ssss0$
(E_2^+) $sss0 \uparrow ss0 = sssssssss0$
(E_3^+) $sss0 \uparrow s0 = sss0$
(E_4^+) $0 \uparrow sss0 = 0$
(E_5^+) $ss0 \uparrow 0 = s0$
(E_6^+) $ssss0 \uparrow 0 = s0$
(E_7^+) $0 \uparrow 0 = s0$

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

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

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^-), (E_7^+).

The additional inputs of the algorithm are $BF = \{\times\}$ *and* $Op = 0$*, suggesting the use of the functor* $\times$ *from* B*, but not* $+$.

An interactive session of the algorithm would go like this:

1. *After the first example* (E_1^+)*, the first* EH *and the corresponding* PH *are enormous because there is no evidence to restrict the generalizations. Since there are many programs with great consilience and total covering, we do not 'activate' the algorithm. However, as none of them is especially better than the other, the algorithm waits for more examples.*
2. *After the second example* E_1^-*, which is negative, no program is inconsistent, so there are still many good programs; we continue.*
3. *The third example* E_2^+ *is not covered by any program. New equations are generated from it like* $X \uparrow Y = ssssssX$ *or* $sX \uparrow X = sssssssX$ *but all of them have poor optimality (*$< OpB$*) so we perform inverse narrowing between* E_2^+ *and* B*, generating new equations: one of them is* $X \uparrow ss0 = X \times X$ *covering the two examples with good optimality, much better than the other programs, so it is offered to the user. The user considers that it is too soon to make any guesses and continues with the algorithm.*
4. *Example* E_2^- *prunes some uninteresting programs.*
5. *Example* E_3^+ *is not covered by any program. New equations are generated from it like* $X \uparrow s0 = X$ *and* $ssX \uparrow X = ssX$*. After some iterations, the algorithm stops because it does not find a consilient program. The best one is, for the moment,* $P = \{X \uparrow s0 = X, X \uparrow ss0 = X \times X\}$ *which is inconsilient but covers all examples.*
6. *Example* E_3^- *prunes some uninteresting programs.*
7. *Example* E_4^+ *is not covered by any program. New equations are generated from it like* $0 \uparrow X = 0$ *generating many combinations but no optimal program is found.*
8. *Example* E_4^- *prunes some uninteresting programs.*
9. *Example* E_5^- *is not covered by any program. New equations are generated from it like* $X \uparrow 0 = s0$ *or* $ss0 \uparrow s0 = s0$*. The former one is combined with a program which contained* $X \uparrow ss0 = X \times X$ *results in new equations like* $X \uparrow sY = (X \uparrow Y) \times X$*,* $X \uparrow sY = X \times (X \uparrow Y)$ *and* $X \uparrow sY = (X \uparrow X) \times Y$*. Using* $X \uparrow sY = (X \uparrow Y) \times X$*, a consistent program is constructed, with very good optimality and consilience. The algorithm stops, offering it to the user.*

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

6.1 Conditional Version

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. Therefore, 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 conserved or increased. Also there are other restrictions, depending of the kind of conditional narrowing (e.g. simple conditional narrowing does not allow extra variables in conditions).

Besides the difficulty of extending the techniques, we have introduced for unconditional theories, we have to deal with the question of selecting when it is convenient to introduce conditions to make a program better according to some criteria.

For instance, with B defining the greater function $>$ we have two short ways of defining max.

$$P_1 = \left\{ \begin{array}{lll} max(X,Y) = X & :- & X > Y = true \\ max(X,Y) = Y & :- & X > Y = false \end{array} \right\}$$
$$P_2 = \left\{ \begin{array}{l} max(s(X), s(Y)) = max(X,Y) \\ max(X,0) = X \\ max(0,Y) = Y \end{array} \right\}$$

The first one is more explanatory because it is consilient jointly with B, but the second one is more efficient. Again, the user must have some parametrised selection criteria to request what kind of hypotheses she expects.

We are currently working on an extension of the algorithm for conditional theories. We have added an extra set CH of conditions, written as sets of equations. We now allow inconsistent generalizations such that if a generalization covers most cases but is inconsistent with a few examples, a new condition can be generated which is 'inspired' in some program from PH (or its negation) that covers these examples.

6.2 Higher-order and Function Invention

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 feasible. In this way, the first steps towards Higher-Order Induction are being taken by Bowers et al. [6]. 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 [9, 27].

Although the greater expressible power of higher-order logic can make hypotheses shorter and more consilient, function invention is the problem that highlights the necessity of higher-order representation languages for induction. Of course, we lose the conveniences of first-order languages, mainly their complete deduction methods, but we acquire benefits for inductive tasks. Fortunately, a good premise to start from is that, whatever the selected higher-order deductive mechanism, any higher-order inductive algorithm should be required to construct terminating programs *for the evidence*.

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. In this paper, we have not studied how selection strategies of narrowing can affect the inference process.

In the future, classical problems of ILP could be addressed under the higher-order extension, like function invention or the induction of schemata (not an ad-hoc use) for complex problems.

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] H. Arimura, H. Ishizaka, T. Shinohara, and S. Otsuki. A generalization of the least generalization. *Machine Intelligence*, 13:59–85, 1992.

[2] F. Bergadano and D. Gunetti. Functional Inductive Logic Programming with Queries to the User. In *Proceedings of the European Conference on Machine Learning*, 1993.

[3] F. Bergadano and D. Gunetti. An Interactive System to Learn Functional Logic Programs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1993.

[4] P. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[5] P.G. Bosco, E. Giovannetti, G. Levi, and C. Palamidessi. A complete semantic characterization of K-leaf, a logic language with partial functions. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 318–327, 1987.

[6] 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.

[7] W. Buntine. Generalised Subsumption and Its Applications to Induction and Redundacy. *Artificial Intelligence*, 36(2):149–176, 1988.

[8] D. Conklin and I.H. Witten. Complexity-Based Induction. *Machine Learning*, 16:203–225, 1994.

[9] Daniel J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114(2):273–298, 21 1993.

[10] L. Fribourg. Slog: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. Second IEEE Int'l Symp. on Logic Programming*, pages 172–185. IEEE, 1985.

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

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

[13] S. Hölldobler. Equational Logic Programming. In *Proc. Second IEEE Symp. on Logic In Computer Science*, pages 335–346. IEEE Computer Society Press, 1987.

[14] 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.

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

[16] A. Ishno and A. Yamamoto. Generalizations in Typed Equational Programming and Their Application to Learning Functions. *New Generation Computing*, 15:85–103, 1997.

[17] J. Jaffar, J.-L. Lassez, and M.J. Maher. A logic programming language scheme. In D. de Groot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 441–468. Prentice Hall, Englewood Cliffs, NJ, 1986.

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

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

[20] C.X. Ling and L. Ungar. Inventing theoretical terms in inductive learning of functions. In Zbigniew W. Ras, editor, *Methodologies for Intelligent Sytems*, volume 4, pages 323–331. Elsevier Science Publishing, North-Holland, 1989.

[21] J.W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995.

[22] S. Muggleton. Duce, An Oracle Based Approach to Constructive Induction. In *Proc. 10th Int'l Joint Conference on Artificial Intelligence*, pages 287–292. Morgan Kaufmann, 1987.

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

[24] R. Olson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, 1995.

[25] G. Plotkin. A note on inductive generalization. *Machine Intelligence*, 6, 1970.

[26] G. Plotkin. A further note on inductive generalization. *Machine Intelligence*, 6, 1971.

[27] Z. Qian. Higher-order equational logic programming. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 254–267, 1994.

[28] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, 1985.

[29] C. Rouveirol. Extensions of Inversion of Resolution Applied to Theory Completion. In S. Muggleton, editor, *Inductive Logic Programming.* Academis Press, London, 1992.

[30] C. Rouveirol. Flattening and Saturation: Two Representation Changes for Generalization. *Machine Learning*, 14:219–232, 1994.

[31] E.Y. Shapiro. Inductive inference of theories from facts. *Computational Logic: Essays in Honor of Alan Robinson*, 1991.

[32] J.H. Siekmann. Universal unification. In *7th Int'l Conf. on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, pages 1–42. Springer-Verlag, Berlin, 1984.

[33] D.R. Smith. The synthesis of LISPprograms from examples: A survey. *Automatic Program Construction Techniques*, pages 307–324, 1984.

[34] K. Taylor. Inverse Resolution of Normal Clauses. In S. Muggleton, editor, *Proc. of 3rd International Workshop on Inductive Logic Programming*, pages 165–178, 1993.

[35] A. Togashi and S. Noguchi. Inductive Inference of Term Rewriting Systems Realizing Algebras. In S. Arikawa, S. Goto, S. Ohsuga, and T. Yokmori, editors, *Proceedings of the First International Workshop on Algorithmic Learning Theory*, pages 411–424. Japanese Society for Artificial Intelligence, 1990.

[36] P.R. van der Laag and Nienhuys-Cheng. Subsumption and Refinement in Model Inference. In P. Brazdil, editor, *Proc. 6th European Conference on Machine Learning*, volume 667 of *Lecture Notes in Artificial Intelligence*, pages 95–114. Springer-Verlag, Berlin, 1993.

[37] M.H. van Emden and K. Yukawa. Logic Programming with Equations. *Journal of Logic Programming*, 4:265–288, 1987.

[38] R. Wirth. Completing Logic Programs by Inverse Resolution. In *Proc. of 4th European Workshop Session on Learning*, pages 239–250, 1989.