An Interactive System to Learn Functional Logic Programs

Francesco Bergadano
University of Catania
via A. Doria 6/A
95100 Catania, Italy
e-mail: bergadan@mathct.cineca.it

Daniele Gunetti
University of Torino
corso Svizzera 185
10149 Torino, Italy
e-mail: gunetti@di.unito.it

Abstract

The problem of learning functional logic programs from positive examples is addressed. We describe a system, called FILP, which asks existential queries to the user, and is able to learn multiple predicates and recursive clauses. We prove that the learned descriptions are correct in the sense that they are consistent with the given examples. Moreover, a correct solution is always found if it exists.

1 Introduction

Inductive Logic Programming (ILP) is the field of Machine Learning concerned with the task of learning logic programs from positive and negative examples in the form of ground literals [Muggleton, 1991; Muggleton, 1992; Rouveirol, 1992; Bergadano *et al.*, 1993]. The obtained results have shown some major problems.

Systems tend to be slow and do not always terminate successfully, even when a solution program exists. A natural approach consists in restricting the hypothesis space by means of strong constraints of various kinds, including rule models [Kietz and Wrobel, 1991], predicate sets [Bergadano, 1991], mode [Kirschenbaum and Sterling, 1991; Muggleton and Feng, 1990; Shapiro, 1983] and type [Morik, 1991; DeRaedt, 1991; Shapiro, 1983] of the variables, and integrity constraints [Raedt and Bruynooghe, 1992].

FILP follows the same idea, and restricts the inductive hypotheses to logic programs that are functional, in the sense that every n-ary predicate P can be associated to a functional relation as follows:

- m of P's arguments are labeled as input,
- the other n-m arguments are labeled as output,
- for every given sequence of input values, there is one and only one sequence of output values that makes the predicate true.

The above properties must be determined by the user and are assumed also for the examples that are not seen. This is necessary because some examples may be missing and will be queried later.
These requirements do not affect the expressive power, as any computable function can be represented by a functional logic program and, on the other hand, make the learning task a lot easier, because many clauses that would otherwise need to be generated and checked against the examples are now disallowed *a priori*, and the order of the literals within a clause antecedent is somehow constrained by the need of computing an output value before it can be used as input in another literal.

Only few systems [Kietz and Wrobel, 1991; Morik, 1991; DeRaedt and Bruynooghe, 1991] are able to learn programs with more than one clause consequent. The reason for this lies in the extensional interpretation of all predicates occurring in clause antecedents, including recursive calls. As a consequence, systems may be unable to learn a program, even when an allowed inductive hypothesis that is consistent with the examples exists. Even worse, it may happen that a program is learned that computes wrong outputs even for the *given* examples. We solve this problem by querying the user for relevant examples that may be missing, depending on the hypothesis space that has been defined. As a result of this, every learned program behaves as required on the given examples and such a program is always found if it exists. The queries that are asked to the user are of the type of the existential queries of CLINT [DeRaedt and Bruynooghe, 1991], because they contain unbound variables. However, these variables are labeled as output, and, as a consequence, there is always one and only one answer to every such query.

Therefore, functionality is the central idea that allows us to face what we have individuated as two major problems in ILP: (1) restricting further and in an explicit way the set of allowed inductive hypotheses and (2) being able to learn multiple predicates, in our case by means of input/output queries that make the extensional informa-

tion sufficient. In the following we will call the predicates that we need to learn the *inductive predicates*, while the others will be either built-in or defined by the user.

FILP does not need a complete extensional specification of the predicates to be learned, up to a certain example complexity, as it happens, e.g., with FOIL [Quinlan, 1990]. Such a specification can be even more time-consuming and error-prone for the user than actually writing the desired program. By contrast, a very limited number of initial examples is required by FILP, and a few additional input values are presented to the user, and the corresponding outputs are requested. However, not all of these input/output examples are always necessary, in the sense that it may happen that, even for a subset of them, there is only one allowed program that is consistent (i.e., the solution is determined). We believe that this problem cannot be remedied unless we give up the extensional interpretation of predicates. But, in that case, the order the clauses are generated with becomes relevant, and backtracking may be required.

2 The Learning Procedure

In this section we present the FILP system and prove some formal properties about it. As in most other ILP learning systems clauses are learned one at a time, independently of the ones that were learned previously. Since FILP learns functional relations, it really only needs positive examples. Negative examples are implicitly assumed to be all the ones having the same input values as the positive examples but different outputs.

2.1 A functional mode for variables

For functional logic programs we need to specify a functional mode for every variable of every literal used in the learning task, in order to employ and learn only functional relations. For example *sum*(in,in,out) would be a legal way to use sum, but *sum*(out,out,in) would not, because it does not represent a function. For this reason, FILP asks the user to provide a functional mode for all predicates, and then uses it for constraining the allowed clauses as follows: a literal L can occur in a clause antecedent $\beta \wedge L \wedge \gamma$ only if all of its input variables occur in β, or are input variables in the head. Moreover, as a result must always be produced, the output variables of the head must occur somewhere in the body.

For example, suppose we have to learn a concept c with a mode for variables c(in,in,out), using predicates a, b, d with a mode for variables a(in,out), b(in,out), d(in,in,in,out). Then the clause c(X,Y,Z) :- a(X,W), b(Y,K), d(X,W,K,Z) is a legal one because every input variable is defined before of being used, and the output variable of c is at last instantiated.

2.2 A general extensional top-down method

This subsection contains a basic version of FILP, without queries for adding the missing information, and an example of its use, in order to describe the approach and to prepare the discussion of the problems stemming from it and the extension that is needed. This basic algorithmic scheme is similar to the ones of ML-SMART [Bergadano and Giordana, 1988] and FOIL [Quinlan, 1990], but uses the notion of a predicate set and requires the functionality constraints, and, therefore, does not need any negative example. It also accepts more than one inductive predicate, although the extension presented later is necessary for making this feature work in general. In the following α and γ represent generic conjunctions of literals.

The main loop in the algorithm tries to cover every positive example of every predicate P. This is done by repeatedly generating clauses of the form "P($\vec{X}$) :- α", and then removing the covered examples. Clause generation is performed as follows:

$\alpha \leftarrow$ true
while covered(α) $\neq \emptyset$ do
  if consistent(α) then return(P($\vec{X}$) :- α)
    else choose a predicate Q and its arguments Args
      such that the functionality constraint
      is satisfied
    if no such Q is found then backtrack
    $\alpha \leftarrow \alpha \wedge$ Q(Args)

where covered(α) and consistent(α) are defined below:

Definition 1: We say that the clause P(X,Y) :- α(X,Y) *extensionally covers* P(a,b) iff α(a,Y) *extensionally computes* Y = b, where extensional computation is defined as follows:

- α = Q(a,Y) with functional mode Q(in,out). Then Q(a,Y) *extensionally computes* Y = b iff Q(a,b) is derivable from the definition of Q or is a given example of Q.
- $\alpha = \gamma$(X,T), Q(T,Y) with functional mode γ(in,out) and Q(in,out). Then γ(a,T), Q(T,Y) *extensionally computes* Y = b iff γ(a,T) *extensionally computes* T = e and Q(e,b) is derivable from the definition of Q or is a given example of Q.

In the algorithm, an example P(a,b) belongs to covered(α) iff α(a,Y) extensionally computes Y=b, and consistent(α) is true iff, for no such example, α(a,Y) extensionally computes Y=c and c≠b. ■

Suppose, for instance, that we have to learn the logic program for *reverse*. Let examples(*reverse*) be:

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

Suppose we also know that *reverse* depends on the following set of predicates, with their usual definition supplied (except for *reverse*, of course):

null, *head*, *tail*, *assign*, *append*, *reverse*.

This is an important information, but obviously still very far away from the actual program that we want to learn: we need to associate variables to these predicates, and divide the obtained literals among the unknown number of clauses that will be necessary. The order of the literals is partially constrained by their mode:

null(out), *null*(out), *head*(in,out), *tail*(in,out),
assign(in,out), *append*(in,in,out), *reverse*(in,out).

The algorithm starts to generate the first clause - the antecedent α is initially empty. We need to choose the first literal Q(Args) to be added to α. As we have left the heuristics unspecified, we will choose it so as to make the discussion short. Variables are taken from the clause head, or from a finite set of additional typed variables.

Let α=*null*(Y). A given example is covered, but we cannot accept the clause *reverse*(X,Y) :- *null*(Y) as it is, because it computes wrong outputs for some given input values, e.g. *reverse*([a],[]), so more literals need to be added.

Let α=*null*(Y) $\wedge$ *head*(X,H); in this case no positive examples are covered. Clause generation fails and we backtrack to the last literal choice.

Let α=*null*(Y) $\wedge$ *null*(X); the first example is covered and no wrong outputs can be computed. A clause is generated and the covered example *reverse*([],[]) is removed from examples(*reverse*).

We proceed to the generation of another clause; α is empty again. Suppose we have already generated α=*head*(X,H) $\wedge$ *tail*(X,T); all the remaining examples are covered, but again we have to specialize because α could compute wrong outputs.

Let α = *head*(X,H) $\wedge$ *tail*(X,T) $\wedge$ *reverse*(T,W); this clause again extensionally covers all remaining examples. For instance, for the last example we have that *head*([a,b,c],a) and *tail*([a,b,c],[b,c]) are true, and *reverse*([b,c],[c,b]) is a given example. However, the output variable Y is not instantiated and the procedure needs to be continued.

Suppose we add the literal *assign*([H],Y). The so obtained clause covers *reverse*([a],[a]) and *reverse*([c],[c]), but for *reverse*([a,b,c],Y) it computes Y=[a], which is not consistent with the data. Further specialization would fail to correct this problem and we need to backtrack, obtaining, e.g.,

α = *head*(X,H) $\wedge$ *tail*(X,T) $\wedge$
$\wedge$ *reverse*(T,W) $\wedge$ *append*(W,[H],Y)

which covers all remaining examples and does not compute wrong outputs. The final solution turns out to be:

reverse(X,Y) :- *null*(Y), *null*(X).
reverse(X,Y) :- *head*(X,H), *tail*(X,T),
reverse(T,W), *append*(W,[H],Y).

2.3 Partial justification of extensionality

Extensional methods are able to learn clauses individually, without worrying about their behavior in the context of the final program. This is an advantage in terms of efficiency, because once a clause is generated, it will be kept, and there is no need for backtracking of this type. This independence of the clauses is made possible by the extensional interpretation of recursion and of the other sub-predicates: when a predicate Q occurs in a clause antecedent α, it is evaluated as true when the arguments match one of the positive examples. For instance, the clause

reverse(X,Y) :- *head*(X,H), *tail*(X,T),
reverse(T,W), *append*(W,[H],Y).
extensionally covers the example *reverse*([a],[a]) because *head*([a],a) and *tail*([a],[]) are true, and *reverse*([],[]) is also a given example. The previously generated logical definitions of Q are not used. The method leads to a fundamental property of extensional methods, which will be proved below.

Definition 2: A program P is *complete* w.r.t. the examples E iff ($\forall$ Q(i,o) $\in$ E) P $\vdash$ Q(i,o). A program P is *consistent* w.r.t. the examples E iff ($\nexists$ Q(i,o) $\in$ E) P $\vdash$ Q(i,o') and o$\neq$o'.

In other words, a complete program computes all desired outputs, and a consistent program does not compute wrong outputs.

Lemma 1: Suppose the extensional top-down method given above outputs a logic program P, that always terminates for the given examples.
Let Q(X,Y) :- α(X,Y) be any clause of P, then
($\forall$Q(a,b) $\in$ Examples(Q)) α(a,Y) ext. computes Y=b $\rightarrow$ P $\vdash$ Q(a,b).
Proof (by contradiction)
Suppose that (1) ($\forall$Q(a,b) $\in$ Examples(Q)) α(a,Y) ext. computes Y=b but (2) P $\nvdash$ Q(a,b). Let α = β(X,Z) $\wedge$ R(Z,W) $\wedge$ γ(W,Y), where R(Z,W) is the first literal such that:
- R is an inductive predicate (in particular, R could be Q),
- there is an example R(r,s) such that β(a,r) $\wedge$ R(r,s) $\wedge$ γ(s,b) is extensionally true.
- for any such r and s, P $\nvdash$ R(r,s).

There must be a literal R(Z,W) in α with these properties, because of the assumptions (1) and (2). But the example R(r,s) must be extensionally covered, since the algorithm has successfully terminated. Therefore, the same argument can be repeated for R(r,s), with a never ending chain of valid deductions. This contradicts the assumption that the program output by the system terminates on all given examples. ■

Theorem 1: If the given extensional top-down method terminates successfully, then it outputs a complete program P.
Proof
($\forall$e $\in$ examples) P extensionally covers e, since the algorithm terminated successfully. By Lemma 1, then, P $\vdash$ e, i.e. P is complete. ■

The above proof is also valid for systems such as FOIL, and is a partial justification of the extensional evaluation of the generated clauses. However, other desirable properties, similar to the one given by Theorem 1, are not true. First, even if a complete and consistent program P exists in the hypothesis space, we are not guaranteed that it will be found. Second, the algorithm may output a program which is inconsistent with respect to the given examples. This is due to the fact that some examples may be missing, while extensionality somehow assumes them to be present. This is true also for functional programs, because even if the single clauses of P satisfy the functionality constraints, different values may be computed by different clauses. A detailed discussion of these problems with examples may be found in [Bergadano, 1993].

2.4 Completing examples before learning

There is no reason why all examples smaller than some fixed complexity bound should have to be given by the user. After all, the whole motivation of induction is that some information is missing. The important points are that (1) if a program P consistent with the given examples exists, then it must be found and (2) the induced program P must not compute wrong outputs on the inputs of given examples. The basic extensional method described in the previous section (and other systems, such as FOIL) guarantees neither, unless some specially determined examples are given in the inductive relations. To overcome this problem, FILP queries the user for some of the missing examples, as done with the "eager" strategy in MIS. For every legal clause (= permitted by the constraints) of the type
P(X,Y) :- A(X,W), Q(X,W,Z), α.
where Q has mode Q(in,in,out), and for every example P(a,b), we do the following:

- extensionally compute A(a,W), obtaining a value W = c;
- ask the user for the value Z computed by Q(a,c,Z);
- add this example to examples(Q).

Adding one example may cause the request of others. Suppose, for instance, that an example A(a,d) is added for A. Then, the above procedure might add an example for Q, e.g. the one matching Q(a,d,Z). As a consequence, the procedure must be repeated for every clause, again and again, until no more examples are added for the inductive predicates.

Both for making the above procedure terminate and for guaranteeing the termination of learned programs, we require that any recursive call within a generated clause matches the following pattern:

$$P(X_1, \ldots, X_i, \ldots, X_n) \text{ :- } \ldots, Q(X_i,Y), \ldots, \ldots, P(X_1, \ldots, Y, \ldots, X_n), \ldots .$$

where $Q(X,Y)$ is known to define a well ordering between Y and X ($Y<X$). It is possible to show that, if every recursive clause in P satisfies the above constraint, then the procedure terminates.

As an instance, suppose that we want to learn a *sort* program. Consider the following clause:
sort(X,Y) :- *tail*(X,T), *sort*(T,W).
It satisfies the constraint on recursive calls because, when *tail*(X,T) is true, then T is a shorter list than X and this is a well order relation. Consider the example *sort*([3,2,1],[1,2,3]). By using the clause, the user is queried for the value of *sort*([2,1],W), and this is added to examples(*sort*). This new example causes the repetition of the procedure, and the user is queried for *sort*([1],W), and at the next step for *sort*([],W).

Not all possible examples have been added, only the ones that were useful for that clause, given the initial example in examples(*sort*). If this is done for all the clauses that are possible a priori, i.e. that satisfy the given constraints, then the problems mentioned in the previous subsection are solved. When speaking of FILP, we will assume in the following that the above completion procedure has been executed as a first step.

Lemma 2: Suppose the examples given to an extensional learning system are completed with the above given procedure. Suppose also that some program P belongs to the hypothesis space and Q(a,b)$\in$examples(Q) after the completion.
If P $\vdash$ Q(a,b) then the first clause in P resolved against Q(a,b) extensionally covers Q(a,b).
Proof (by contradiction)

Suppose that (1) P $\vdash$ Q(a,b), where Q(X,Y) :- $\alpha(X,Y)$ is the first clause used in the proof, but (2) Q(a,b) is not extensionally covered by this clause.
Let $\alpha(X,Y) = \beta$(X,Z) $\wedge$ R(Z,W) $\wedge$ $\gamma(W,Y)$. Suppose that P $\vdash$ β(a,r) $\wedge$ γ(s,b) and P $\vdash$ R(r,s), but R(r,s) $\notin$ examples(R). There must be one literal R(Z,W) having this property, because of assumptions (1) and (2), and let R(Z,W) be the first such literal. The user must have been queried for R(r,W), because β(a,r) is extensionally covered. Since R(r,s) $\notin$ examples(R), the answer must have been W=w$\neq$s. But then no clause could extensionally cover R(r,s), or it would be inconsistent, while P $\vdash$ R(r,s). We could now repeat the same argument for R. This would produce a non-terminating chain of resolution steps, and a finished proof of Q(a,b) would never be obtained, contradicting the initial hypothesis that P $\vdash$ Q(a,b). ■

Theorem 2: If a complete and consistent program P exists, then FILP will terminate successfully.
Proof (by contradiction)
If FILP does not terminate with a solution, there must be an example Q(a,b) that it cannot cover. Since P is complete, P $\vdash$ Q(a,b). Take the first clause Q(X,Y) :- α resolved against Q(a,b). By Lemma 2, α(a,Y) extensionally computes Y=b. But this clause would have been expanded by FILP and found to cover the example. Moreover, consistent(α) is true, because P is consistent and by the contrapositive of Lemma 1, and therefore the clause would have been generated and the example would have been covered. ■

Theorem 3: If FILP terminates successfully, then it outputs a consistent program P.
Proof
If P is not consistent, then there must be some example Q(a,b), such that P $\vdash$ Q(a,c) and b$\neq$c. But, by Lemma 2, some clause Q :- α of P will extensionally cover Q(a,c). But, in that case, consistent(α) would have been false and FILP would not have generated that clause. ■

By virtue of Theorem 1, this program will also be complete.

3 Results

FILP is written in C-prolog (interpreted) and runs on a SUN SPARCstation 1. Table 1 contains results about some standard logic programs learned by FILP. Times required to learn the programs are in seconds.
The second column of the table contains the set of examples initially given to FILP. Observe that, apart from *member* and *subset*, one initial carefully chosen example is sufficient to learn the corresponding program. *Member* and *subset* require a minimum of two examples because they do not represent functional relations, and must be turned into functions by adding a boolean argument which is true if the corresponding relation holds and false otherwise. In general, it is possible to prove that one initial example is sufficient to learn all the clauses necessary to derive it. This means that it is possible to learn a program starting with the *minimum* set of examples such that the program itself and the negation of those examples represent a minimally unsatisfiable set of clauses. Then, it is obviously better to chose examples as simple as possible, in order to limit the number of queries the user will have to answer.
The third column shows the final number of examples used by FILP in the learning process, the initial ones plus those queried to the user. For example, starting from the given example, the set of examples of union is completed by querying the user for *union*([b],[a,c],?) and for *union*([],[a,c],?).
Note that FILP is also able to learn more concepts at the same time. For example it has learned *quicksort* together with its major subprogram *partition*, starting with only one example of *quicksort* and, at the beginning, without examples of *partition*. The time required for the entire task was about the sum of the times required to learn independently *quicksort* and *partition*.
We must remark that FILP does not learn concepts *separately*, as it would be done by other systems such as FOIL and GOLEM [Muggleton and Feng, 1990]. We tell the system that *quicksort* could depend on *partition*, and FILP queries the user for the missing examples of *partition*. Then these examples are used to learn *partition* itself.

FILP can also work with the extensional definition of a concept. So, it is possible to learn *quicksort* without learning *partition*, and only relying on its extensional definition (the given or queried examples). On the other hand it can learn more than two concepts together. We could also give no intensional definition for *append*, but only some examples, and have FILP learn also *append* together with *quicksort* and *partition*. Finally, the last program in the table is a version of *quicksort* not employing the *append* predicate.

program	*initial examples*	*c. e.*	*time*
exponential	exponential(2,3,8)	4	4.52
factorial	factorial(3,6)	4	5.32
member	member(a,[c,a],yes)	4	12.24
	member(a,[b],no)		
reverse	reverse([a,b,c],[c,b,a])	4	4.38
union	union([a,b],[a,c],[b,a,c])	3	13.67
intersection	int([b,a],[c,a],[a])	3	13.90
subset	subset([b],[c,b,a],yes)	3	16.07
	subset([a,d],[c,b,a],no)		
partition	part([1,3,0],2,[1,0],[3])	7	81.50
quicksort	quick([2,1,3,0],[0,1,2,3])	6	12.86
qsort. app.	qsapp([2,1,3],[],[1,2,3])	7	6.80

Table 1.

As an example, below are the learned programs for *partition* and *quicksort* without *append*; it should be noted that FILP works with "flattened" clauses [Rouveirol, 1993], where functions are transformed into predicates.

partition(L,El,L1,L2) :- *null*(L), *null*(L1), *null*(L2).
partition(L,El,L1,L2) :- *head*(L,X1), *tail*(L,X2),
partition(X2,El,Ls,Bs),
cons(X1,Bs,L2),
assign(Ls,L1), X1>El.
partition(L,El,L1,L2) :- *head*(L,X1),*tail*(L,X2),
partition(X2,El,Ls,Bs),
cons(X1,Ls,L1),
assign(Bs,L2), X1≤El.

qsort_app(X,Acc,Y) :- *null*(X),*assign*(Acc,Y).
qsort_app(X,Acc,Y) :- *head*(X,H),*tail*(X,T),
partition(T,H,Ys,Yl),
qsort_app(Yl,Acc,Yls),
cons(H,Yls,Ylsacc),
qsort_app(Ys,Ylsacc,Y).

4 Conclusion

We have described four major characteristics of FILP.

First of all, since FILP *knows* it is learning a function, it needs only positive examples of the program to be learned. This makes the task of the user easier, because he or she has to think only in terms of "what the program must compute" and not in terms of "what the program must not compute". Nonetheless, for every given positive example Q(a,b), all the corresponding negative examples {Q(a,k)|k≠b} are implicitly known and can be used by FILP. Moreover, strong functionality constraints limit the number of legal clauses and consequently increase the efficiency of the system.

Second, FILP requires a very limited number of examples and, partially, this is still due to the knowledge of FILP about functions. For example, *quicksort* is learned with only six positive examples (plus seven examples for *partition*, if we want to learn that, too), while GOLEM employs fifteen positive examples (plus an unspecified number of negative examples) to learn *quicksort* alone. Simpler programs require a smaller number of examples to be learned. The only system, up to our knowledge, able to learn logic programs with a smaller number of examples than FILP is LOPSTER [Lapointe and Matwin, 1992]. Usually LOPSTER works with only two examples, but one of the two is in fact required to correspond to the non-recursive clause of the desired program. Moreover LOPSTER is a restricted learning system and it is, for example, unable to learn *quicksort*.

Third, FILP queries the user for the missing examples. This means that the user must not provide all the needed examples to learn a program at the beginning. He or she can forget some examples, and FILP will ask for them. Observe that FILP queries the user only for the examples it really needs, so it will not waste time trying to cover useless examples. A similar technique is used in Shapiro's MIS. However MIS is an incremental system, and newly added examples may require some previously generated clause to be retracted. This happens both with the contradiction backtracing algorithm and when refining clauses with the eager strategy [Shapiro, 1983]. In other systems [Muggleton and Feng, 1990; Quinlan, 1990] the user must provide all the examples at one time, and usually a superset of the examples needed is given, resulting in a lot of time wasted in covering useless examples. Experience with FILP has shown that the best way to use it is to start with just few significant examples (most of the time one is enough), and the system will query for the missing ones needed to perform the learning task.

Finally, FILP learns complete and consistent programs, and this means first of all that programs learned by FILP have not unexpected behavior (that is, they do not cover negative examples) as it is the case for other extensional methods (such as FOIL) which do not complete the given examples.

Experiments and results obtained with FILP have shown how, limiting our attention to functional logic programs, it is possible to acquire efficiency (with almost no loss in generality) by exploiting constrains on functionality and implicit negative examples. We believe that this possibility is not limited to extensional methods, but could be useful for every ILP learning system.

Acknowledgments: This work was in part supported by BRA ESPRIT III Project 6020 on Inductive Logic Programming.

References

[Bergadano and Giordana, 1988] F. Bergadano and A. Giordana. A Knowledge Intensive Approach to Concept Induction. In *Proc. of the Fifth Int. Conf. on Machine Learning*, pages 305–317, Ann Arbor, MI, 1988. Morgan Kaufmann.

[Bergadano *et al.*, 1993] F. Bergadano, L. DeRaedt, S. Matwin, and S. Muggleton (Eds.). *Proc. of the IJCAI-93 Workshop on Inductive Logic Programming.* IJCAII, Chambéry, France, 1993.

[Bergadano, 1991] F. Bergadano. The Problem of Induction and Machine Learning. In *Proc. Int. Joint Conf. on Artificial Intelligence*, pages 1073–1079, Sydney, Australia, 1991. IJCAII.

[Bergadano, 1993] F. Bergadano. Inductive Data Base Relations. *To appear in IEEE Trans. on Data and Knowledge Engineering*, 1993.

[DeRaedt and Bruynooghe, 1991]
L. DeRaedt and M. Bruynooghe. CLINT: A Multistrategy Interactive Concept-Learner and Theory Revision System. In R. S. Michalski and G. Tecuci, editors, *Proc. Workshop on Multistrategy Learning*, pages 175–190, Harpers Ferry, VA, 1991.

[DeRaedt, 1991] L. DeRaedt. *Interactive Concept Learning*. Ph.D. thesis, Katholieke Univ. Leuven, 1991.

[Kietz and Wrobel, 1991] J. U. Kietz and S. Wrobel. Controlling the Complexity of Learning in Logic Through Syntactic and Task-Oriented Models. In S. Muggleton, editor, *Inductive Logic Programming*, London, 1991. Academic Press.

[Kirschenbaum and Sterling, 1991] M. Kirschenbaum and L. Sterling. Refinement Strategies for Inductive Learning of Simple Prolog Programs. In j. Mylopoulos and R. Reiter, editors, *Proc. Int. Joint Conf. on Artificial Intelligence*, pages 757–761, Sydney, Australia, 1991. IJCAII.

[Lapointe and Matwin, 1992] S. Lapointe and S. Matwin. Sub-unification: A Tool for Efficient Induction of Recursive Programs. In *Proc. of the Int. Machine Learning Conference*, pages 273–281. Morgan Kaumann, 1992.

[Morik, 1991] K. Morik. Balanced Cooperative Modeling. In R. S. Michalski and G. Tecuci, editors, *Proc. Workshop on Multistrategy Learning*, pages 65–80, Harpers Ferry, VA, 1991.

[Muggleton and Feng, 1990] S. Muggleton and C. Feng. Efficient Induction of Logic Programs. In *Proc. of the First Conf. on Algorithmic Learning Theory*, Tokyo, 1990. OHMSHA.

[Muggleton, 1991] S. Muggleton, editor. *Inductive Logic Programming*. Academic Press, 1991.

[Muggleton, 1992] S. Muggleton, editor. *Proc. of the Workshop on Inductive Logic Programming*. Held as a post-workshop at the FGCS conference, Tokyo, Japan, 1992.

[Quinlan, 1990] J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266, 1990.

[Raedt and Bruynooghe, 1992]
L. De Raedt and M. Bruynooghe. Belief Updating from Integrity Constraints and Queries. *Artificial Intelligence*, 53:291–307, 1992.

[Rouveirol, 1992] C. Rouveirol, editor. *Proc. of the ECAI Workshop on Logical Approaches to Learning*. ECCAI, Vienna, Austria, 1992.

[Rouveirol, 1993] C. Rouveirol. Flattening: a Representation Change for Generalization. *Machine Learning*, 1993. Special issue on Evaluating and Changing Representation, K. Morik, F. Bergadano and W. Buntine (Eds.).

[Shapiro, 1983] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.