The use of functional and logic languages in machine learning

Peter A. Flach

University of Bristol
Woodland Road, Bristol BS8 1UB, United Kingdom
Peter.Flach@bristol.ac.uk, http://www.cs.bris.ac.uk/~flach/

Abstract. Traditionally, machine learning algorithms such as decision tree learners have employed attribute-value representations. From the early 80's on people have started to explore Prolog as a representation formalism for machine learning, an area which came to be called inductive logic programming (ILP). With hindsight, however, Prolog may not have been the best choice, since it can be argued that types and functions, well known from functional programming, are essential ingredients of the individual-centred representations employed in machine learning. Consequently, a combined functional logic language is a better vehicle for learning with a rich representation. In this talk I will illustrate this by means of the higher-order functional logic programming language *Escher*. The paper concentrates on giving a leisurely introduction to ILP.

1 Introduction

Inductive logic programming has its roots in concept learning from examples, a relatively straightforward form of induction that has been studied extensively by machine learning researchers [23]. The aim of concept learning is to discover, from a given set of pre-classified examples, a set of classification rules with high predictive power. For many concept learning tasks, so-called attribute-value languages have sufficient representational power. An example of an attribute-value classification rule is

IF Outlook = Sunny AND Humidity = High THEN PlayTennis = No

A learned concept definition would consist of several of such rules. Learning a boolean concept can be generalised to multi-class classification problems, where one would learn a set of rules for each class. (In contrast, in concept learning we are usually not interested in learning rules for the complement of the concept.)

When objects are structured and consist of several related parts, we need a richer representation formalism with variables to refer to those parts. In the 1980's machine learning researchers started exploring the use of logic programming representations, which led to the establishment of inductive logic programming (ILP) [30] as a subdiscipline at the intersection of machine learning and computational logic. Recent years have seen a steady increase in ILP research, as well as numerous applications to practical problems like data mining and scientific discovery – see [2, 6] for an overview of such applications. Successful ILP applications include drug design [16], protein secondary

structure prediction [29], mutagenicity prediction [38], carcinogenesis prediction [39], medical diagnosis [24], discovery of qualitative models in medicine [11], finite-element mesh design [5], telecommunications [37], natural language processing [25], recovering software specifications [3], and many others. Detailed surveys of ILP are provided by [18, 30], while [19] offers an extensive overview of on-line available systems and datasets, as well as a bibliography with nearly 600 entries. [28, 4] are collections of research papers. `http://www.cs.bris.ac.uk/~ILPnet2/` is a good starting place for web searches.

2 Inductive Logic Programming

In this section we give a tutorial introduction to the main forms of predictive inductive logic programming.[1] One instance of a predictive ILP problem concerns the inductive construction of an intensional predicate definition (a set of Horn clauses with a single predicate in the head) from a selection of ground instances of the predicate. More generally, there can be several predicates whose definitions are to be learned, also called *foreground* predicates or *observables*. In the general case, this requires suitably defined auxiliary or *background predicates* (simple recursive predicates such as `member/2` and `append/3` notwithstanding). The induced set of rules or *inductive hypothesis* then provides an intensional connection between the foreground predicates and the background predicates; we will sometimes call such rules *foreground* rules. We will also use the terms *facts* to refer to extensional knowledge, and *rules* to refer to intensional knowledge. The terms 'knowledge' or 'theory' may refer to both facts and rules. Thus, predictive induction infers foreground rules from foreground facts and background theory.

Definition 1 (Predictive ILP). *Let P_F and N_F be sets of ground facts over a set of foreground predicates F, called the* positive examples *and the* negative examples*, respectively. Let T_B, the* background theory*, be a set of clauses over a set of background predicates B. Let L be a* language bias *specifying a* hypothesis language H_L *over $F \cup B$ (i.e., a set of clauses). A predictive ILP task consists in finding a* hypothesis $H \subseteq H_L$ *such that $\forall p \in P_F : T_B \cup H \models p$ and $\forall n \in N_F : T_B \cup H \not\models n$.*

The subscripts F and B are often dropped, if the foreground and background predicates are understood. We will sometimes refer to all examples collectively as E.

Definition 1 is under-specified in a number of ways. First, it doesn't rule out trivial solutions like $H = P$ unless this is excluded by the language bias (which is not often the case since the language bias cannot simply exclude ground facts, because they are required by certain recursive predicate definitions). Furthermore, the definition doesn't capture the requirement that the inductive hypothesis correctly predicts unseen examples. It should therefore be seen as a general framework, which needs to be further instantiated to capture the kinds of ILP tasks addressed in practice. We proceed by briefly discussing a number of possible variations, indicating which of these we can handle with the approach proposed in this chapter.

[1] The alternative form of *descriptive* ILP, which is not primarily aimed at classification, is outside the scope of this paper.

Clauses in T and H are often restricted to definite clauses with only positive literals in the body. Some ILP algorithms are able to deal with normal clauses which allow negative literals in the body. One can go a step further and allow negation over several related literals in the body (so-called *features*).

In a typical predictive ILP task, there is a single foreground predicate to be learned, often referred to as the *target predicate*. In contrast, *multiple predicate learning* occurs when $|F| > 1$. Multiple predicate learning is hard if the foreground predicates are mutually dependent, i.e., if one foreground predicate acts as an auxiliary predicate to another foreground predicate, because in that case the auxiliary predicate is incompletely specified. Approaches to dealing with incomplete background theory, such as abductive concept learning [14], can be helpful here. Alternatively, multiple predicate learning may be more naturally handled by a descriptive ILP approach, which is not intended at learning of classification rules but at learning of properties or constraints that hold for E given T. The problems of learning recursive rules, where a foreground predicate is its own auxiliary predicate, are related to the problems of multiple predicate learning.

Definition 1 only applies to boolean classification problems. The definition could be extended to multi-class problems, by supplying the foreground predicate with an extra argument indicating the class. In such a case, a set of rules has to be learned for each class. It follows that we can also distinguish binary classification problems in which both the positive and negative class have to be learned explicitly (rather than by negation-as-failure, as in the definition).

In *individual-centred* domains there is a notion of individual, e.g. molecules or trains, and learning occurs on the level of individuals only. Usually, individuals are represented by a single variable, and the foreground predicates are either unary predicates concerning boolean properties of individuals, or binary predicates assigning an attribute-value or a class-value to each individual. Local variables referring to parts of individuals are introduced by so-called structural predicates. Individual-centred representations allow for a strong language bias for feature construction. On the other hand, most program synthesis tasks lack a clear notion of individual. Consider, for instance, the definition of `reverse/2`: if lists are seen as individuals – which seems most natural – the clauses are not classification rules; if pairs of lists are seen as individuals, turning the clauses into boolean classification rules, the learning system will have to rediscover the fact that the output list is determined by the input list.

Sometimes a predictive ILP task is unsolvable with the given background theory, but solvable if an additional background predicate is introduced. For instance, in Peano arithmetic multiplication is not finitely axiomatisable unless the definition of addition is available. The process of introducing additional background predicates during learning is called *predicate invention*. Predicate invention can also be seen as an extreme form of multiple predicate learning where some of the foreground predicates have no examples at all.

An initial foreground H_0 may be given to the learner as a starting point for hypothesis construction. Such a situation occurs e.g., in incremental learning, where examples become available one-by-one and are processed sequentially. Equivalently, we can perceive this as a situation where the background theory also partially defines the foreground predicate(s). This is usually referred to as *theory revision*.

After having considered the general form that predictive ILP problems may take, we now turn our attention to predictive ILP algorithms. Broadly speaking, there are two approaches. One can either start from short clauses, progressively adding literals to their bodies as long as they are found to be overly general (*top-down* approaches); or one can start from long clauses, progressively removing literals until they would become overly general (*bottom-up* approaches). Below, we illustrate the main ideas by means of some simplified examples.

2.1 Top-down induction

Basically, top-down induction is a generate-then-test approach. Hypothesis clauses are generated in a pre-determined order, and then tested against the examples. Here is an example run of a fictitious incremental top-down ILP system:

example	*action*	*clause*
`+m(a,[a,b])`	add clause	`m(X,Y)`
`-m(x,[a,b])`	specialise:	try `m(X,[])`
		try `m(X,[V|W])`
		try `m(X,[X|W])`
`+m(b,[b])`	do nothing	
`+m(b,[a,b])`	add clause:	try `m(X,[V|W])`
		try...
		try `m(X,[V|W]):-m(X,W)`

The hypothesis is initialised with the most general definition of the target predicate. After seeing the first negative example, this clause is specialised by constraining the second argument. Several possibilities have to be tried before we stumble upon a clause that covers the positive example but not the negative one. Fortunately, the second positive example is also covered by this clause. A third positive example however shows that the definition is still incomplete, which means that a new clause has to be added. The system may find such a clause by returning to a previously refuted clause and specialise it in a different way, in this case by adding a literal to its body.

The resulting clause being recursive, testing it against the examples means querying the predicate to be learned. Since in our example the base case had been found already this doesn't pose any problem; however, this requires that the recursive clause is learned last, which is not always under control of the teacher. Moreover, if the recursive clause that is being tested is incorrect, such as `m(X,Y):-m(Y,X)`, this may lead to non-termination problems. An alternative approach, known as extensional coverage, is to query the predicate to be learned against the examples. Notice that this approach would succeed here as well because of the second positive example.

The approach illustrated here is basically that of Shapiro's *Model Inference System* [35, 36], an ILP system *avant la lettre* (the term 'inductive logic programming' was coined in 1991 by Muggleton [27]). MIS is an incremental top-down system that performs a complete breadth-first search of the space of possible clauses. Shapiro called his specialisation operator a *refinement operator*, a term that is still in use today (see [17]

for an extensive analysis of refinement operators). A much simplified Prolog implementation of MIS can be found in [7]. Another well-known top-down system is Quinlan's FOIL [34].

As the previous example shows, clauses can be specialised in two ways: by applying a substitution, and by adding a body literal. This is formalised by the relation of θ-subsumption, which establishes a syntactic notion of generality.

Definition 2 (θ-subsumption). *A clause C_1 θ-subsumes a clause C_2 iff there is a substitution* θ *such that all literals in $C_1\theta$ occur in C_2.*[2]

θ-subsumption is reflexive and transitive, but not antisymmetric (e.g., `p(X):-q(X)` and `p(X):-q(X),q(Y)` θ-subsume each other). It thus defines a pre-order on the set of clauses, i.e., a partially ordered set of equivalence classes. If we define a clause to be *reduced* if it does not θ-subsume any of its subclauses, then every equivalence class contains a reduced clause that is unique up to variable renaming. The set of these equivalence classes forms a lattice, i.e., two clauses have a unique least upper bound and greatest lower bound under θ-subsumption. We will refer to the least upper bound of two clauses under θ-subsumption as their θ-LGG (least general generalisation under θ-subsumption). Note that the lattice does contain infinite descending chains.

Clearly, if C_1 θ-subsumes C_2 then C_1 entails C_2, but the reverse is not true. For instance, consider the following clauses:

```
nat(s(X)):-nat(X).
nat(s(s(Y))):-nat(Y).
nat(s(s(Z))):-nat(s(Z)).
```

Every model of the first clause is necessarily a model of the other two, both of which are therefore entailed by the first. However, the first clause θ-subsumes the third (substitute `s(Z)` for `X`) but not the second. Gottlob characterises the distinction between θ-subsumption and entailment [10]: basically, C_1 θ-subsumes C_2 without entailing it if the resolution proof of C_2 from C_1 requires to use C_1 more than once.

It seems that the entailment ordering is the one to use, in particular when learning recursive clauses. Unfortunately, the least upper bound of two Horn clauses under entailment is not defined. The reason is simply that, generally speaking, this least upper bound would be given by the disjunction of the two clauses, but this may not be a Horn clause. Furthermore, generalisations under entailment are not easily calculated, whereas generalisation and specialisation under θ-subsumption are simple syntactic operations. Finally, entailment between clauses is undecidable, whereas θ-subsumption is decidable (but NP-complete). For these reasons, ILP systems usually employ θ-subsumption rather than entailment. Idestam-Almquist defines a stronger form of entailment called T-implication, which remedies some of the shortcomings of entailment [12, 13].

[2] This definition, and the term θ-subsumption, was introduced in the context of induction by Plotkin [32, 33]. In theorem proving the above version is termed subsumption, whereas θ-subsumption indicates a special case in which the number of literals of the subsumant does not exceed the number of literals of the subsumee [21].

2.2 Bottom-up induction

While top-down approaches successively specialise a very general starting clause, bottom-up approaches generalise a very specific bottom clause. Again we illustrate the main ideas by means of a simple example. Consider the following four ground facts:

```
a([1,2],[3,4],[1,2,3,4]).    a([2],[3,4],[2,3,4]).
a([a],[],[a])                a([],[],[]).
```

Upon inspection we may conjecture that these ground facts are pairwise related by one recursion step, i.e., the following two clauses may be ground instances of the recursive clause in the definition of `a/3`:

```
a([1,2],[3,4],[1,2,3,4]):-
     a([2],[3,4],[2,3,4]).
a([a],[],[a]):-
     a([],[],[]).
```

All that remains to be done is to construct the θ-LGG of these two ground clauses, which in this simple case can be constructed by anti-unification. This is the dual of unification, comparing subterms at the same position and turning them into a variable if they differ. To ensure that the resulting inverse substitution is the least general anti-unifier, we only introduce a new variable if the pair of different subterms has not been encountered before. We obtain the following result:

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

which is easily recognised as the recursive clause in the standard definition of `append/3`.

In general things are of course much less simple. One of the main problems is to select the right ground literals from a much larger set. Suppose now that we know which head literals to choose, but not which body literals. One approach is to simply lump all literals together in the bodies of both ground clauses:

```
a([1,2],[3,4],[1,2,3,4]):-
     a([1,2],[3,4],[1,2,3,4]),a([a],[],[a]),
     a([],[],[]),a([2],[3,4],[2,3,4]).

a([a],[],[a]):-
     a([1,2],[3,4],[1,2,3,4]),a([a],[],[a]),
     a([],[],[]),a([2],[3,4],[2,3,4]).
```

Since bodies of clauses are, logically speaking, unordered, the θ-LGG is obtained by anti-unifying all possible pairs of body literals, keeping in mind the variables that were introduced when anti-unifying the heads. Thus, the body of the resulting clause consists of 16 literals:

```
a([A|B],C,[A|D]):-
     a([1,2],[3,4],[1,2,3,4]),a([A|B],C,[A|D]),
```

```
a(W,C,X),a([S|B],[3,4],[S,T,U|V]),
a([R|G],K,[R|L]),a([a],[],[a]),
a(Q,[],Q),a([P],K,[P|K]),a(N,K,O),
a(M,[],M),a([],[],[]),a(G,K,L),
a([F|G],[3,4],[F,H,I|J]),a([E],C,[E|C]),
a(B,C,D),a([2],[3,4],[2,3,4]).
```

After having constructed this bottom clause, our task is now to generalise it by throwing out as many literals as possible. To begin with, we can remove the ground literals, since they are our original examples. It also makes sense to remove the body literal that is identical to the head literal, since it turns the clause into a tautology. More substantially, it is reasonable to require that the clause is connected, i.e., that each body literal shares a variable with either the head or another body literal that is connected to the head. This allows us to remove another 7 literals, so that the clause becomes

```
a([A|B],C,[A|D]):-
     a(W,C,X),a([S|B],[3,4],[S,T,U|V]),a([E],C,[E|C]),a(B,C,D).
```

Until now we have not made use of any negative examples. They may now be used to test whether the clause becomes overly general, if some of its body literals are removed. Another, less crude way to get rid of body literals is to place restrictions upon the existential variables they introduce. For instance, we may require that they are determinate, i.e., have only one possible instantiation given an instantiation of the head variables and preceding determinate literals.

The approach illustrated here is essentially the one taken by Muggleton and Feng's Golem system [26] (again, a much simplified Prolog implementation can be found in [7]). Although Golem has been successfully applied to a range of practical problems, it has a few shortcomings. One serious restriction is that it requires ground background knowledge. Furthermore, all ground facts are lumped together, whereas it is generally possible to partition them according to the examples (e.g., the fact `a([a],[],[a])` has clearly nothing to do with the fact `a([2],[3,4],[2,3,4])`). Both restrictions are lifted in Muggleton's current ILP system Progol [31]. Essentially, Progol constructs a bottom clause for a selected example by adding its negation to the (non-ground) background theory and deriving all entailed negated body literals. By means of mode declarations this clause is generalised as much as possible; the resulting body literals are then used in a top-down refinement search, guided by a heuristic which measures the amount of compression the clause achieves relative to the examples (see the next section on heuristics). Progol is thus a hybrid bottom-up/top-down system. It has been successfully applied to a number of scientific discovery problems.

The examples we used above to illustrate top-down and bottom-up ILP algorithms concerned inductive synthesis of simple recursive programs. While illustrative, these examples are non-typical of many ILP approaches which perform classification rather than program synthesis, use an individual-centred representation, and employ background knowledge rather than recursion. An example of this kind of ILP problem will be given in the next section, where we will also introduce the Escher representation.

3 Learning in Escher

Consider the following learning problem from [22]. The learning task is to discover low size-complexity Prolog programs for classifying trains as Eastbound or Westbound. The problem is illustrated in Figure 1.

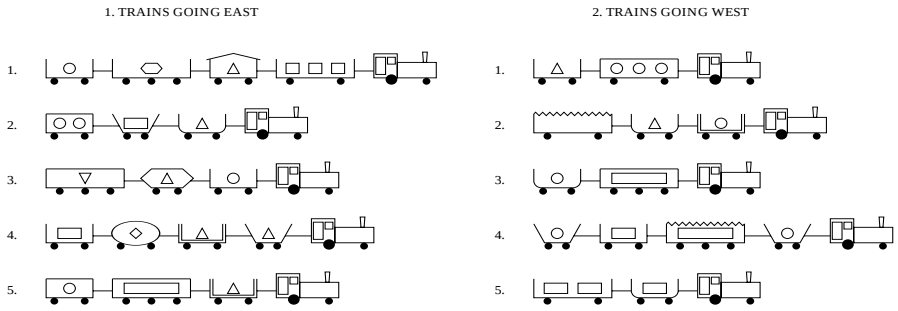

Fig. 1. The ten train East-West challenge.

Each train consists of 2-4 cars; the cars have attributes like shape (rectangular, oval, u-shaped, ...), length (long, short), number of wheels (2, 3), type of roof (none, peaked, jagged, ...), shape of load (circle, triangle, rectangle, ...), and number of loads (1-3). A possible rule distinguishing between eastbound and westbound trains is 'a train is eastbound if it contains a short closed car, and westbound otherwise'.

3.1 Prolog representations

Datalog is a subset of Prolog in which the only functors are of arity 0 (i.e., constants). This simplifies inference as unification only needs to be performed between two variables, or between a variable and a constant. Similarly, it simplifies the specialisation and generalisation operators in ILP. The drawback is a loss of structure, as aggregation mechanisms such as lists are not available. Structured objects need to be represented indirectly, by introducing names for their parts.

A Datalog representation of the first train in Figure 1 is as follows.

```
eastbound(t1).

hasCar(t1,c11).          hasCar(t1,c12).
cshape(c11,rect).        cshape(c12,rect).
clength(c11,short).      clength(c12,long).
croof(c11,none).         croof(c12,none).
cwheels(c11,2).          cwheels(c12,3).
hasLoad(c11,l11).        hasLoad(c12,l12).
```

```
lshape(l11,circ).        lshape(l12,hexa).
lnumber(l11,1).          lnumber(l12,1).

hasCar(t1,c13).          hasCar(t1,c14).
cshape(c13,rect).        cshape(c14,rect).
clength(c13,short).      clength(c14,long).
croof(c13,peak).         croof(c14,none).
cwheels(c13,2).          cwheels(c14,2).
hasLoad(c13,l13).        hasLoad(c14,l14).
lshape(l13,tria).        lshape(l14,rect).
lnumber(l13,1).          lnumber(l14,3).
```

Using this representation, the above hypothesis would be written as

```
eastbound(T):-hasCar(T,C),clength(C,short),not croof(C,none).
```

Testing whether this hypothesis correctly classifies the example amounts to proving the query `?-eastbound(t1)` from the hypothesis and the description of the example (i.e., all ground facts minus its classification).

In full Prolog we can use terms to represent individuals. The following representations uses functors to represent cars and loads as tuples, and lists to represent a train as a sequence of cars.

```
eastbound([car(rect,short,none,2,load(circ,1)),
           car(rect,long, none,3,load(hexa,1)),
           car(rect,short,peak,2,load(tria,1)),
           car(rect,long, none,2,load(rect,3))]).
```

In this representation, the hypothesis given before is expressed as follows:

```
eastbound(T):-member(C,T),arg(2,C,short),not arg(3,C,none).
```

Strictly speaking, this representation is not equivalent to the previous ones because we now encode the order of cars in a train. We could encode the order of cars in the Datalog representation by using the predicates `hasFirstCar(T,C)` and `nextCar(C1,C2)` instead of `hasCar(T,C)`. Alternatively, we can ignore the order of cars in the term representation by only using the `member/2` predicate, effectively turning the list into a set. From the point of view of hypotheses, the two hypothesis representations are isomorphic: `hasCar(T,C)` corresponds to `member(C,T)`, `clength(C,short)` corresponds to `arg(2,C,short)`, and `croof(C,none)` corresponds to `arg(3,C,none)`. Thus, Datalog and term representations look very different concerning examples, and very similar concerning hypotheses.

3.2 Escher representation

The term representation has the advantage that all information pertaining to an individual is kept together. Moreover, the structure of the terms can be used to guide hypothesis construction, as there is an immediate connection between the type of an individual

and the predicate(s) used to refer to parts of the individuals. This connection between term structure and hypothesis construction is obviated by using a strongly typed language [8]. The following representation uses a Haskell-like language called Escher, which is a higher order logic and functional programming language [20].

```
eastbound :: Train->Bool;
type Train = [Car];
type Car = (CShape,CLength,CRoof,CWheels,Load);
data CShape = Rect | Hexa | ...;
data CLength = Long | Short;
data CRoof = None | Peak | ...;
type CWheels = Int;
type Load = (LShape,LNumber);
data LShape = Circ | Hexa | ...;
type LNumber = Int;

eastbound([(Rect,Short,None,2,(Circ,1)),
           (Rect,Long, None,3,(Hexa,1)),
           (Rect,Short,Peak,2,(Tria,1)),
           (Rect,Long, None,2,(Rect,3))]) = True;
```

The important part here is the type signature. The first line defines `eastbound` as a function mapping trains to booleans. The lines starting with `type` define type synonyms (i.e., the type signature could be rewritten without them). The lines starting with `data` define algebraic datatypes; here, they are simply enumerated types. The actual representation of an example is very similar to the Prolog term representation, except that it is an equation rather than a fact. Notice that functions are more natural to express classification rules than predicates.

The hypothesis is now expressed as follows:

```
eastbound(t) = (exists \c -> member(c,t) &&
                  proj2(c)==Short && proj3(c)!=None)
```

Here, the phrase `exists \c ->` stands for explicit existential quantification of variable `c`, and `proj2` and `proj3` project on the second and third component of a 5-tuple representing a car, respectively. Again, the hypothesis is structurally similar to the Prolog one. However, the main point about strongly typed representations is that the type signature is available to the learning algorithm to guide hypothesis construction.

The term perspective gives us a clear view on the relation between attribute-value learning (AVL) and first- and higher-order learning. In AVL, examples are represented by tuples of constants. Hypotheses are built by referring to one of the components of the tuple by means of projection, followed by a boolean condition on that component (e.g., being equal to a constant).[3] First-order representations such as Prolog generalise this

[3] In practice this projection is not explicitly used, as any condition on a component of the tuple can be equivalently written as a condition on the tuple. The resulting rule will then have the same variable in all literals. Such rules could be called *semi-propositional*, as the only role of the variable is to distinguish hypotheses from examples. This explains why attribute-value learning is often loosely called propositional learning.

by allowing lists and other recursive types, as well as an arbitrary nesting of subtypes (e.g., an individual could be a tuple, one component of which could be a list of tuples). Higher-order representations generalise this further by allowing sets and multisets.[4]

Further examples illustrating the Escher representation can be found in [8]. A higher-order decision tree learner is described in [1], and a higher-order evolutionary progamming system in [15].

4 Concluding remarks

This paper has provided an introduction to ILP for non-machine learners. We have also given an example of a learning problem represented in Escher. In the talk I will outline the specific advantages of using a language like Escher for learning.

Acknowledgements

This work would not have been possible without the fruitful collaboration with (former) members of the Bristol Machine Learning group. I am particularly indebted to Christophe Giraud-Carrier, Nicolas Lachiche, and John Lloyd. Part of this material appeared in [9]; another part has been written in collaboration with Nada Lavrač.

References

1. A.F. Bowers, C. Giraud-Carrier, and J.W. Lloyd. Classification of individuals with complex structure. In P. Langley, editor, *Proceedings of the 17th International Conference on Machine Learning*, Morgan Kaufmann, pp.81–88, 2000.
2. I. Bratko and S. Muggleton. Applications of Inductive Logic Programming. *Communications of the ACM* 38(11):65–70, November 1995.
3. W. Cohen. Recovering software specifications with inductive logic programming. *Proc. Twelfth National Confeernce on Artificial Intelligence*, The MIT Press, 1994.
4. L. De Raedt, editor. *Advances in Inductive Logic Programming*. IOS Press, 1996.
5. B. Dolšak, S. Muggleton. The application of inductive logic programming to finite-element mesh design. In S. Muggleton, ed., *Inductive Logic Programming*, pp. 453–472. Academic Press, 1992.
6. S. Džeroski and I. Bratko. Applications of Inductive Logic Programming. In [4], pp.65–81.
7. P.A. Flach. *Simply Logical – intelligent reasoning by example*. John Wiley, 1994.
8. P.A. Flach, C. Giraud-Carrier, and J.W. Lloyd. Strongly typed inductive concept learning. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 185–194. Springer-Verlag, 1998.
9. P.A. Flach. From extensional to intensional knowledge: Inductive Logic Programming techniques and their application to deductive databases. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *Lecture Notes in Computer Science*, pages 356–387. Springer-Verlag, 1998.
10. G. Gottlob. Subsumption and implication. *Inf. Proc. Letters* 24:109–111, 1987.

[4] A set is equivalent to a predicate; passing around sets as terms requires a higher-order logic.

11. D.T. Hau and E.W. Coiera. Learning qualitative models of dynamic systems. *Machine Learning*, 26(2/3): 177–212, 1997.
12. P. Idestam-Almquist. *Generalization of clauses*. PhD thesis, Stockholm University, October 1993.
13. P. Idestam-Almquist. Generalization of Clauses under Implication. *J. AI Research*, 3:467–489, 1995.
14. A.C. Kakas and F. Riguzzi. Learning with abduction. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 181–188. Springer-Verlag, 1997.
15. C.J. Kennedy. *Strongly typed evolutionary programming*. PhD Thesis, University of Bristol, 2000.
16. R.D. King, S. Muggleton, R. Lewis, and M.J.E. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proc. of the National Academy of Sciences of the USA* 89(23): 11322–11326, 1992.
17. P. van der Laag. *An analysis of refinement operators in Inductive Logic Programming*. PhD Thesis, Erasmus University Rotterdam, December 1995.
18. N. Lavrač and S. Džeroski. *Inductive Logic Programming: techniques and applications*. Ellis Horwood, 1994.
19. N. Lavrač, I. Weber, D. Zupanič, D. Kazakov, O. Štěpánková, and S. Džeroski. ILPNET repositories on WWW: Inductive Logic Programming systems, datasets and bibliography. *AI Communications* 9(4):157–206, 1996.
20. J.W. Lloyd. Programming in an integrated functional and logic programming language. Journal of Functional and Logic Programming, 1999(3).
21. D.W. Loveland and G. Nadathur. Proof procedures for logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 5, D.M. Gabbay, C.J. Hogger, and J.A. Robinson (editors), Oxford University Press, pp.163–234, 1998.
22. D. Michie, S. Muggleton, D. Page, and A. Srinivasan. To the international computing community: A new East-West challenge. Technical report, Oxford University Computing laboratory, Oxford,UK, 1994.
23. T.M. Mitchell. Machine Learning. McGraw-Hill, 1997.
24. F. Mizoguchi, H. Ohwada, M. Daidoji, S. Shirato. Using inductive logic programming to learn classification rules that identify glaucomatous eyes. In N. Lavrač, E. Keravnou, B. Zupan, eds., *Intelligent Data Analysis in Medicine and Pharmacology*, Kluwer, pp. 227–242, 1997.
25. R.J. Mooney, M.E. Califf. Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research* 3: 1–24, 1995.
26. S. Muggleton and C. Feng. Efficient induction of logic programs. *Proc. First Conf. on Algorithmic Learning Theory*, Ohmsha, Tokyo, pp. 368–381, 1990. Also in [28], pp.281–298.
27. S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–317, 1991. Also in [28], pp.3–27.
28. S. Muggleton, editor. *Inductive Logic Programming*. Academic Press, 1992.
29. S. Muggleton, R.D. King, and M.J.E. Sternberg. Protein secondary structure prediction using logic. *Protein Engineering* 7: 647–657, 1992.
30. S. Muggleton and L. De Raedt. Inductive Logic Programming: theory and methods. *J. Logic Programming*, 19/20:629–679, 1994.
31. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
32. G. Plotkin. A note on inductive generalisation. *Machine Intelligence 5*, B. Meltzer and D. Michie (editors), North-Holland, pp.153–163, 1970.

33. G. Plotkin. A further note on inductive generalisation. *Machine Intelligence 6*, B. Meltzer and D. Michie (editors), North-Holland, pp.101–124, 1971.
34. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.
35. E.Y. Shapiro. *Inductive inference of theories from facts*. Techn. rep. 192, Comp. Sc. Dep., Yale University, 1981.
36. E.Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.
37. E. Sommer. Rulebase stratifications: An approach to theory restructuring. *Proc. Fourth Int. Workshop on Inductive Logic Programming*, GMD-Studien 237, pp. 377-390, 1994.
38. A. Srinivasan, S. Muggleton, R.D. King, and M.J.E. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. *Proc. Fourth Int. Workshop on Inductive Logic Programming*, GMD-Studien 237, pp. 217–232, 1994.
39. A. Srinivasan, R.D. King, S. Muggleton, and M.J.E. Sternberg. Carcinogenesis prediction using inductive logic programming. In N. Lavrač, E. Keravnou, and B. Zupan, eds., *Intelligent Data Analysis in Medicine and Pharmacology*, Kluwer, pp. 243–260, 1997.