Inductive Metalogic Programming [1]

Andreas Hamfelt
Computing Science Department
Uppsala University

Jørgen Fischer Nilsson
Department of Computer Science
Technical University of Denmark

Abstract: We propose a metalogic programming method for efficient induction of a fairly large class of list-handling logic programs delineated through restrictions on the hypothesis language. These restrictions take the form of predefined program recursion schemes (higher order "clichés") from which the hypotheses programs in the induction process are derived by plugging in either simple, nonrecursive clause programs or invented predicates.

Metalogic programming is applied for handling clauses as first class data objects in connection with program schemes, and moreover for obtaining a flexible control scheme during the induction, avoiding a blindfold generate-and-test against the given program examples. A metalogic program induction testbed has been constructed and has successfully been applied to some induction problems discussed in the literature.

Keywords: Induction of logic programs, metalogic programming, higher order program clichés, predicate invention, mixed bottom-up and top-down induction.

1 Introduction

The primary aim of the present study is to explore application of (quasi-)higher-order and metalogic programming methods to inductive logic programming. This is done by establishing a metalogic programming set up for induction, and by providing a testbed in the form of a metalogic program, written in PROLOG, for conducting experiments with induction of logic programs with the suggested method.

In logic programming a program $\wp$ takes the form of a finite collection of definite clauses, and program execution is conducted as a logical inference process for verifying a goal clause:

[1]The research presented herein was supported by Nordisk Forskerutdanningsakademi (NORFA) under grant no. 93.35.138.

Information about authors:
Andreas Hamfelt: Mail address: Computing Science Department, Uppsala University, Box 311, S-751 05 Uppsala, Sweden. Fax: +46 18 52 12 70. Phone: +46 18 18 10 37. Email: hamfelt@csd.uu.se.
Jørgen Fischer Nilsson: Mail address: Department of Computer Science 344, Technical University of Denmark, DK-2800 Lyngby, Denmark. Fax: + 45 42 88 45 30. Phone: + 45 45 93 33 32 ask for 3730. Email: jfn@id.dtu.dk.

$$\wp \vdash goal$$

The goals here are (conjunctions of) atomic formulae, where variables are existentially quantified. The computation of a proof provides term instances of the variables, which are then construed as a result of the computation.

The logical inference is usually conducted as a resolution proof, say using SLD-resolution, applying a depth-first search of the space of resolvents. In the refutation proof the hypothesis goal is negated (the variables hence becoming universally quantified); a successful proof is reached when obtaining the empty clause resolvent.

2 Metalogic Programming

In metalogic programming [4] a theory in the form of a logic program $\wp$ comprised of clauses is encoded as a term $\lceil \wp \rceil$ and the provability relation $\vdash$ is formalised as a predicate $demo(\ ,\ )$. This predicate is defined by clauses $\wp_{demo}$ constituting a logic program, often being referred to as a metainterpreter.

Accordingly

$$\wp \vdash goal \quad \text{iff} \quad \wp_{demo} \vdash demo(\lceil \wp \rceil, \lceil goal \rceil)$$

which is sometimes regarded as a pair of opposite reflection principles for passing between object-level and metalevel. The ways of encoding a formula ϕ into a corresponding term $\lceil \phi \rceil$ (cf. Gödel encoding) are discussed extensively in the metalogic programming literature, see e.g. [5]. Basically there are two options: (1) Ground term representation in which variables of the object language formula become ground (i.e., variable-free) terms in the encoding and (2) Non-ground encoding in which object language variables become variables of the metalanguage. The latter representation makes it easier to exploit the "built in" object level reasoning mechanism (including unification) at the risk of causing confusion between variables of the different language levels.

The explicit availability of the proof (computation) predicate *demo* through the interpreter clause program $\wp_{demo}$ in metalogic programming renders it possible to "customize", monitor, and control the deduction process.

This reflection between object level and metalevel can be iterated, giving rise to the so-called multilayered metalogic programming methodology [7, 8]. Multilayered metalogic programs have been taken advantage of in connection with partial evaluation, in which in particular the (term encoded) metainterpreter may be provided as argument to itself (self-applicable partial evaluator). We apply multilayered metalogic programming in sect. 6.

3 Inductive Logic Programming

Inductive reasoning aims at deriving principles for establishing general rules or new concepts on the basis of a number of observations in the context of some background knowledge.

Stated in logical terms induction is the rational formulation of a hypothesis in the form of a logical sentence H, so that the factual logical sentences constituting the available observations logically follows from H together with the background knowledge B through logical inference:

$$H + B \vdash observations$$

In addition to the positive observations to be confirmed by the hypothesis, there may also be negative observations available, constraining the admissible hypotheses. The theoretical basis for a constructive computational logical approach to induction was laid down in [14] and followed up by contemporary theoretical research, e.g. [15].

Inductive logic programming in particular aims at computationally synthesizing a logic program which can derive the given input-output data. These data usually take the form of a finite test suite of (ground) atomic goals.

Such a proposed program hypothesis has to conform with certain constraints so as to avoid trivial solutions, e.g. ruling out hypotheses which simply take the form of the given test suite (assumed finite). Inductive logic programming thus seeks to turn a partial explicit definition of a predicate into an implicit definition.

3.1 Bottom-up versus Top-down Induction

There are two contemporary computational approaches to inductive logic programming [6, 11, 12]: Bottom-up induction based on generalization of observations (i.e. input/output examples), and conversely top-down induction based on specialization of an overly general hypothesis program.

The *bottom-up approach* tries to come up with definite clauses from which the observations follow or are plainly instances. These clauses are constructed by a generalization technique using the so-called *least general generalization LGG* of two atoms (unit clauses).

The LGG of atoms A_1 and A_2 is an atom A such that
(1) there are substitutions θ_1 and θ_2 so $A\theta_1 = A_1$ and $A\theta_2 = A_2$, and
(2)if atom A' has also property (1), then there exists a substitution σ such that $A'\sigma = A$.

This notion generalises to clauses through the so-called θ-subsumption notion. However in the present context the LGG of atoms suffices, since we combine with a top-down method for forming non-unit clauses.

In the *top-down approach* to induction the starting point in principle is the k-place universal relation expressed by the hypothesis clause:

$$p(X_1, ..., X_k)$$

In successive steps this relation is then constrained by introduction of definite clauses with their bodies forming conditions.

In any case the crucial (and the most intricate) part of the logic programming induction process is to come up with proper recursion structures and to invent appropriate auxiliary predicates. Before explaining our computational approach to this creative aspect of induction to which we direct our attention, let us motivate the choice of a metalogic

programming framework.

3.2 Metalogic Programming for Inducing Programs

Appealing to the above-mentioned formalization of the logic programming provability, computational induction in principle lends itself to metalogic programming through the goal clause

$$\leftarrow \ demo(T, \lceil observation \rceil)$$

with the understanding that the range of T is now term encodings of theories. The variable T is to be instantiated with the term representation of an induced logic program.

Less naïvely the candidate programs may be constrained through a predicate *good_theory* delineating the admissible clausal programs. This gives rise to two variant programs, either

$$\leftarrow \ demo(T, observation), \ good\_theory(T)$$

where programs emerging as instantiations of the variable T are filtered, or conversely

$$\leftarrow \ good\_theory(T), \ demo(T, observation)$$

where *good_theory* is used for generating candidate hypotheses in the induction.

This pair of complementary proposals suggest a versatile merger, say, in the form of a constraint logic programming solution in which the hypothesis theory is computed by collaboration in an interleaved computation. As alternative here we pursue a metalogic programming methodology, which combines the above two control schemes with a mixed top-down and bottom-up induction approach.

However before elaborating on the control framework, let us introduce the program schemes used for the top-down part of the induction.

4 Higher-order Programming Clichés

In logic programming languages based on first order predicate logic such as PROLOG, quantified variables range over individuals, and there is hence no variable ranging over relations.

Nevertheless it is tempting from a programming methodology point of view to admit predicate variables in order to provide program schemes or "clichés" [2] from which proper programs result by instantiation of predicate variables with predicate constants. Strictly logically this is a daring move from predicate logic to logical type theory, which is pursued in higher logic programming languages such as λ-PROLOG, calling for the ensuing complex notion of higher order unification.

Here we take a more pragmatical view using the method outlined in [16] for emulating higher order mechanisms in usual clausal logic, akin to the rule models in the induction system MOBAL/RDT [9] (see also [11]).

Consider for instance the higher order predicate *map* which expresses that the relationship R holds for each pair obtained by consecutively coupling the members of lists L_1 and L_2.[2]

$map([], R, []).$
$map([X|L_1], R, [Y|L_2]) :- R(X,Y), map(L_1, R, L_2).$

One way of coping with this logical specification in ordinary first order clauses is to introduce a predicate, say *apply* ([16], see also [1, 13]), which expresses predication, that is the application of a predicate to its arguments. For the above this gives the logic program proper

```
map([],R,[]).
map([X|U],R,[Y|V]) :- apply(R,X,Y), map(U,R,V).
```

The *apply* predicate is then defined "pointwise" for the relevant predicates by

`apply`$(p, X_1, ..., X_n)$ `:-` $p(X_1, ..., X_n)$.

for each proper predicate constant p in the program.[3]

Metalogic programming features the alternative more direct clause representation (as a term)

```
[[map,[],R,[]],
[map,[X|U],R,[Y|V]] <= ([R,X,Y] & [map,U,R,V])].
```

Another commonly occurring cliché, a classical from functional programming, cf. e.g. [3], is the `filter`

```
filter([],P,[]).
filter([X|L],P,[X|LL]) :- apply(P,X), filter(L,P,LL).
filter([X|L],P,LL) :- not apply(P,X), filter(L,P,LL).
```

As the most general cliché, however, we focus on `fold` (called `iterate` in [16])

```
fold([],R,Z,Z).
fold([X|L],R,Z,Y) :- fold(L,R,Z,U), apply(R,X,U,Y).
```

With this powerful cliché the well-known `append` predicate simply becomes

```
append(U,V,W) :- fold(U,cons,V,W).
```

appealing to the auxiliary general purpose list constructor predicate defined by the unit clause

```
cons(U,V,[U|V]).
```

The usual form of `append` then can be obtained by unfolding and folding. [4]

[2]This predicate conforms with the function(al) *map*(*list*) in functional programming.

[3]This clause for the predication may be viewed as a watered down version of the *comprehension principle*, which in its general form as in logical type theories admits arbitrary formulae, not just predicate constants. The comprehension principle may be taken to be the source of invention of predicates in the induction process.

[4]Folding and unfolding means to replace a formula by its definiendum and definiens, respectively. Base case: unfolding `append(U,V,W):-fold(U,cons,V,W)` with `fold([],R,Z,Z):-true` yields

As a matter of fact the `map` can be formed as an instance of `fold`.

```
map(U,P,V) :- fold(U,p1(P),[],V).
apply(p1(P),X,Y,[Z|Y]) :- apply(P,X,Z)
```

This example illustrates a variant use of clichés, appealing to currying, cf. also [1].

It turns out, perhaps rather surprisingly, that a comprehensive collection of commonly occurring list predicates falls under this scheme. These draw on a restricted repertoire of argument predicates, supplemented with predicates themselves formed by this scheme.

Let **fold-0** denote the class of predicates which are defined by clauses which do not appeal to (other) predicates, i.e., unit clauses.

Let **fold-1** denote predicates defined by a clause calling on `fold` accepting only **fold-0** predicate as argument. Hence `append` as well as predicates `last` and `member` can be obtained as **fold-1** predicates.

```
last(U,V) :- fold(U,p1,[],[V]).
apply(p1,X,Y,Z) :- p1(X,Y,Z).
p1(X,[],[X]).
p1(X,[Y|Z],[Y|Z]).
```

The list membership predicate can be obtained from `filter` or through the following use of `fold`

```
member(X,U) :- fold(U,p1(X),[],[V|W]).
apply(p1(X),X,Y,[X]).
apply(p1(X),Y,Z,Z) :- X <> Y.
```

One observes that `fold` is bound to terminate with a presumed ground first argument provided that the argument predicate terminates. Consequently some computable relations fall outside the scope of the `fold` scheme.

4.1 Regress to Previously Defined Predicates

The above identified class **fold-1** of list-processing predicates can be extended by recursively appealing to predicates already in the class, giving rise to the more comprehensive class **fold-2**, etc. in a stepwise manner.

For instance the naïve reversal of lists can be expressed as follows

```
reverse(U,V) :- fold(U,p1,[],V).
p1(X,Y,Z) :- append(Y,[X],Z)
```

So naïve reverse belongs to **fold-2** since it appeals to the previously defined `append`. One observes that the recursions inherent in the two applications of `fold` are non-mutual, that

---

`append([],L,L):-true`.
Recursive case: unfolding `append(U,V,W):-fold(U,cons,V,W)` with
`fold([X|L],R,Z,Y):-fold(L,R,Z,U),apply(R,X,U,Y)` yields
`append([F|R1],S,S1):-fold(R1,cons,S,S2),apply(cons,F,S2,S1)`. Unfolding this with
`apply(cons,U,V,[U|V]):-true` yields `append([F|R1],S,[F|R2]):-fold(R1,cons,S,R2)`. Folding this with `append(U,V,W):-fold(U,cons,V,W)` yields `append([F|R1],S,[F|R2]):-append(R1,S,R2)`.

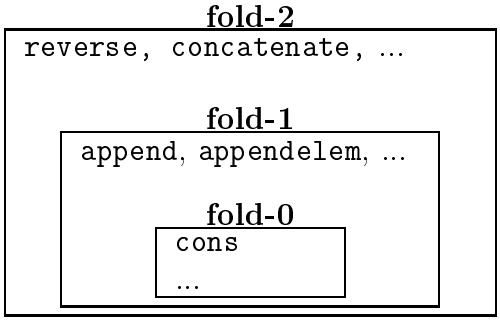
is the recursions are stratified.

Unfolding and folding of the program yields the usual

```
reverse([],[]).
reverse([X|U],W) :- reverse(U,V), append(V,[X],W).
```

Likewise, the concatenation of lists in a list is a **fold-2** predicate applying the predicate `append`.

Below is shown the inclusion hierarchy (including **fold-**i in **fold-**$(i+1)$) of clause programs considered candidates for induction. In the following treatment of induction the base predicates of **fold-0** are confined to unit clauses.



5 Inductive Metalogic Programming

The induction strategy explored here has as its point of departure the above general program scheme `fold`. The clause hypothesis space is constituted by the class of programs definable through `fold`. This space may be expanded by extending the repertoire of program clichés.

The inductive computation of a clause program from a given behaviour suite of input-output atoms is conducted by MetaInduce in a mixed top-down/bottom-up mode as follows:

1. *Abduction step*: Invoke the considered program cliché successively with members of the observation suite, and then consider the suite of resulting instantiations of the arguments to `apply` in the cliché. This suite constitutes recursively a behaviour suite for the unknown predicate argument for the cliché.

2. *Generalization step*: Compute the least general generalization, LGG, of the above abduced atoms for the unknown predicate argument, using an anti-unification algorithm, cf. e.g. [6]. (In a simplified version this step alternatively may simply refer to a given collection of base predicates.)

3. *Acceptance/goal-regression step*: Apply acceptance criteria to the resulting LGG unit clause with the program cliché, e.g. checking that no members of a negative test suite are produced, that ground outputs follow from ground input terms, etc. If acceptable, then stop. Otherwise dispense with this LGG, and instead recursively

invoke MetaInduce with the test suite resulting from the above abduction step, thereby inventing an auxiliary predicate with a recursive definition of its own.

In this step appeal might also be made to a library of predicates already induced through this method, cf. the above-stated hierarchy of **fold-i** predicates.

5.1 Example: Inducing Append

As a case study let us consider the induction of `append` given, say, the following test suite

```
append([],[],[]),
append([a],[],[a]),
append([b,c],[d,e],[b,c,d,e]),
append([],[f],[f])
```

The abduction step in the MetaInduce induction scheme abduces the following suite for the unknown predicate argument `p` from the above observations[5]:

```
p(a,[],[a|[]]),
p(b,U,[b|[c,d,e]]), p(c,[d,e],U)
```

A remark is in order concerning the variable `U`. Observe that it is shared by several abduced observations. Such a variable has the status of a metavariable ranging over expressions of the language of the observations. All the observations should adhere to a common pattern which will determine the binding to `U`. This specifies the observations, i.e., they will either become ground or contain only object variables. Anti-unification can then be applied to obtain their LGG.

The common pattern is identified by selecting a ground observation, `p(a,[],[a|[]])` here, and abstracting its structure by simultaneously replacing each term by a variable, obtaining `p(X,Y,[X|Y])`. The pattern is then unified with each of the remaining observations yielding `p(b,[c,d,e],[b|[c,d,e]])`, `p(c,[d,e],[c,d,e])`. The binding to `U` is thus `[c,d,e]`. (Note that here the pattern is in fact the LGG for the observations. Several patterns may of course be obtainable. An observation like `p(m,[m,n,o],[m,m,n,o])` for instance, adheres both to `p(X,Y,[X|Y]` and `p(X,[X|Y],[X,X|Y])`. Selecting the pattern that matches the other observations is then a non-deterministic problem.)

In the second step of generalization, the LGG unit clause

```
p(U,V,[U|V]).
```

is formed from this suite. This clause is recognized as the `cons` list constructor predicate and unfolding yields the following successful result of the induction

```
fold([],cons,Z,Z).
fold([X|L],cons,Z,Y) :- fold(L,cons,Z,U), cons(X,U,Y).
```

which is recognised as the append program after "tidying up".

[5]The first and last observation do not contribute, since they be handled by the first clause for `fold` (empty list case). The terms have been syntactically sugared to pinpoint the common structure of the two atoms.

5.2 Example: Predicate Invention

To demonstrate the predicate invention feature, consider the reversal predicate presented to MetaInduce say, as the following suite of examples

```
rev([],[]),
rev([a],[a]),
rev([b,c,d],[d,c,b]),
rev([e,f],[f,e])
```

The abduction step produces the following suite for the unknown predicate

```
p(a,[],[a]),
p(b,U,[d,c,b]), p(c,U1,U), p(d,[],U1)
p(e,V,[f,e]), p(f,[],V).
```

The computed LGG of this suite is `p(U,V,l(W1,W2))` which fails on the requirement that ground inputs (first two arguments) would yield ground output.

Failing all else, the abducer is recursively invoked with the abduced suite. However, this necessitates a "type conversion", since three of the arguments to `fold` must be of (the same) list type. The abduced suite is therefore rejected. Preparatory to the recursive invocation, an alternative suite is obtained by applying a variant form of `fold`, with the subgoal `apply(R,U,[X],Y)`, to the original observations. The alternative is

```
p([],[a],[a]),
p(U,[b],[d,c,b]), p(U1,[c],U), p([],[d],U1)
p(V,[e],[f,e]), p([],[f],V)
```

leading to the predicate invention `append`, confer with the induction of `append` above. Combining with the first step yields the naïve reverse as successful result of the entire induction.

6 Metalogical Framework for the Induction

Let us now explain our metalogic programming methodology for representing program clichés.

As outlined in sect. 3.2 there are two alternatives for a metalogic representation of induction. The first is illustrated by the goal

$$\leftarrow demo(T, \lceil observation \rceil), good\_theory(T)$$

where a binding is computed to the variable T and then assessed by the predicate "*good_theory*". This approach can be made less naïve by reversing the conjuncts so that T is bound to, say, a theory schema before commencing the search for a theory logically implying "*observation*". However the search still will be blindfold and the chance of succeeding with an appropriate binding to T small.

The more viable alternative is the multilayered representation, i.e.,

$$\leftarrow demo(t_2, \lceil demo(T_1, \lceil observation \rceil) \rceil)$$

where t_2 is a theory representing program clichés and controlling the inferencing from proposed instantiations of these clichés. An instance can be rejected at an early stage. A selection of primitive predicates for the cliché can be represented and then reasoned about as the test suite is tried out.

This leads to the following metainterpreter which is the core of the METAINDUCE program

```
demo(_,_,true,[]).
demo(TheoryName,Theory,A,AbdObs):-
     theory(TheoryName,Theory),
     member((A<=B),Theory),
     demo(TheoryName,FreshTheory,B,AbdObs).
demo(TheoryName,T,and(A,B),[AbdObs1,AbdObs2]):-
     demo(TheoryName,FT1,A,AbdObs1),
     demo(TheoryName,FT2,B,AbdObs2).
demo(cliches,_,A,A):-
     demo(basepreds,_,demo(cliches,_,A,A),_).
```

This metainterpreter tests whether a theory (second argument) exists from which the observation (third argument) logically follows. Before each inference a fresh copy of the fold cliché is retrieved from

```
theory(cliches,
       [(p(X,Y,Z)<=fold(X,F,Y,Z)),
        (fold([],F,Z1,Z1)<=true),
        (fold([X2|L],F,Z2,Y2)<= and(fold(L,F,Z2,Y0),
                                    apply(F,X2,Y0,Y2)))]).
```

The retrieved cliché is a metarepresentation of a partial object theory (a theory schema). During the course of the inferencing this representation needs to be completed when insufficient. The required metametareasoning is initiated by the reflection rule.

```
demo(cliches,_,A,A):-
     demo(basepreds,_,demo(cliches,_,A,A),_).
```

On reflection the metatheory is retrieved from clauses like

```
theory(basepreds,
       [(demo(t1,Theory,apply(F,X,Y0,Y),AbdObs) <=
           lookup(F,X,Y0,Y)),
        (lookup(cons,X1,L,[X1|L])<=true)]).
```

The sample metatheory here contains information about what base predicates in **fold-0** that fold may exploit, in this case only *cons* is stored.

If there is no suitable base predicate, i.e., the metatheory looks like

```
theory(basepreds,
       [(demo(t1,Theory,apply(F,X,Y0,Y),AbdObs) <=
           lookup(F,X,Y0,Y)),
        (lookup(aux,X1,Y1,Z1)<=true)]).
```

then examples of the unknown predicate *aux* are abduced. These are recorded in the

fourth argument *AbdObs* of *demo*. The METAINDUCE program looks like

```
meta_induce(Theory,Obs):-
     demo(cliches,Th1,Obs,AbdObs),
     anti_unification(AbdObs,LGG_Clause),
     theory(cliches,Th2),
     append(Th2,[(LGG_Clause<=true)],Theory).
```

A query with the append examples discussed above

```
meta_induce(Theory,and(p([],[],[]),and(p([a],[],[a]),
                   and(p([b,c],[d,e],[b,c,d,e]),p([],[f],[f])))))).
```

yields as output this theory

```
Theory = [p(A,B,C) <= fold(A,R,B,C),
          fold([],R,Z,Z) <= true,
          fold([X|L],R,Z1,Y) <=
               and(fold(L,R,Z1,U),
               apply(R,X,U,Y)),
          apply(aux,X2,L2,[X2|L2]) <= true]
```

The unknown predicate thus has been identified as *cons* so unfolding and folding of the theory gives the usual definition of append.

7 Concluding Summary

We have sketched an approach to induction of logic programs from examples based on a repertoire of higher order program schemes catering for the recursion. Moreover, the ability to perform goal directed predicate invention within the suggested metalogic program scheme has been indicated.

Of course this excursion into inductive logic programming is yet to establish itself as a viable and competitive general-purpose approach. A natural improvement to its robustness is to extend the repertoire of higher order clichés so as to cover a comprehensive class of logic programs. Double recursions (as found e.g. in clauses for flattening lists) and difference lists (as used e.g. in clauses derived from clause grammars) call for appropriate clichés.

In a longer perspective metalogic (programming) offers means of reflecting on information including the absence of information. In the context of induction such reflection mechanisms might be applied to reason about presence vs. absence of test examples.

References

[1] H. Aït-Kaci & R. Nasr: Integrating Logic and Functional Programming, *Lisp and Symbolic Computation*, 2, pp. 51–89, 1989.

[2] D. Barker-Plummer: Cliche Programming in PROLOG, in M. Bruynooghe (ed.): *Proc. Second Workshop on Meta-programming in Logic*, pp. 247–256, Department of Computer Science, Katholieke Universiteit Leuven, 1990.

[3] R. Bird & Ph. Wadler: *Introduction to Functional Programming*, Prentice Hall (C.A.R. Hoare Series), 1988.

[4] K. A. Bowen & R. A. Kowalski: Amalgamating Language and Metalanguage in Logic Programming, in K. L. Clark & S.-Å. Tärnlund (eds.): *Logic Programming*, pp. 153–172, Academic Press, 1982.

[5] K. Eshghi: *Meta-language in Logic Programming*, Ph.D. Thesis, Department of Computing, Imperial College, London, 1987.

[6] P. Flach: *Simply Logical, Intelligent Reasoning by Example*, Wiley, 1994.

[7] A. Hamfelt: *Metalogic Representation of Multilayered Knowledge*, Ph.D. Thesis, Uppsala Theses in Computing Science 15, Uppsala University, Uppsala, 1992.

[8] A. Hamfelt, Å. Hansson: Representation of Fragmentary Multilayered Knowledge, in A. Pettorossi (ed.): *Meta-Programming in Logic*, Lecture Notes in Computer Science 649, pp. 321–335, Springer-Verlag, 1992.

[9] J. Kietz and S. Wrobel: Controlling the complexity of learning in logic through syntactic and task-oriented models, in [12].

[10] R. A. Kowalski: Problems and Promises of Computational Logic, in J. W. Lloyd (ed.) *Computational Logic*, pp. 1–36, Springer-Verlag, 1990.

[11] N. Lavrač & S. Džeroski: *Inductive Logic Programming, Techniques and Applications*, Ellis Horwood, 1994.

[12] S. Muggleton (ed.): *Inductive Logic Programming*, Academic Press, 1992.

[13] J. Fischer Nilsson: Combinatory Logic Programming, in M. Bruynooghe (ed.): *Proc. Second Workshop on Meta-programming in Logic*, pp. 187–202, Department of Computer Science, Katholieke Universiteit Leuven, 1990.

[14] G. Plotkin: A Note on Inductive Generalization, in B. Meltzer & D. Michie (eds.): *Machine Intelligence 5*, pp. 153–163, Edinburgh University Press, 1969.

[15] T. Sato: A Complete Set of Rules for Inductive Inference, report TR-92-44, Electrotechnical Laboratory, Tsukuba, Japan, 1992.

[16] D. H. D. Warren: Higher Extensions of Prolog - are they needed?, J. E. Hayes, D. Michie & Y-H Pao (eds.), *Machine Intelligence 10*, pp. 441–454, Edinburgh University Press, 1982.