Flexible query-answering systems modelled in metalogic programming

Troels Andreasen and **Henning Christiansen**
Roskilde University
P.O. Box 260, DK-4000 Roskilde, Denmark
{*troels, henning*}@*dat.ruc.dk*

Abstract. Metaprogramming adds new expressive power to logic programming which can be advantageous to transfer to the field of deductive databases. We propose metaprogramming as a way to model and develop new, flexible query-answering systems.

A model is shown, extending deductive databases by a classification of the clauses in the database, an integration of nonstandard inference rules, and a notion of proof constraints in which a variety of flexible ways of evaluating database queries can be expressed. Furthermore, it is indicated how techniques developed in metalogic programming for abduction and induction may be applied for modelling knowledge discovery and data mining.

1 Introduction

Deductive databases have never reached a widespread acceptance in practical applications of databases, but at the conceptual level, they are acknowledged for their simplicity combined with high expressibility. As such, the field of deductive databases has proved to be an important research platform and in many ways setting the standard for future database technology. The deductive database formalism is a subset of first order logic, for which the logic programming scheme, predominantly in the shape of Prolog, provides implementations that are sufficient for many applications and also may show the way to fullscale database systems.

Metaprogramming, which has been studied extensively in the recent years, provides an extended expressibility to logic programming and in the present paper, we suggest to use metaprogramming as a methodology in deductive database research with a bias towards flexible query-answering systems. We believe that modelling and experimenting in this way with new database formalisms and query-answering mechanisms is useful, leading from early and vaguely understood proposals to proper formalizations.

In this paper, we propose an extended model for deductive databases and query evaluation characterized by

- a classification of the database clauses into separate spaces of domain knowledge,
- a parameterization by the inference rules, which may be instantiated into a collection of nonstandard rules, and
- a reification of the proof, which we recognize as an important part of the answer to a query, and on which a variety of natural constraints in a database query can be expressed.

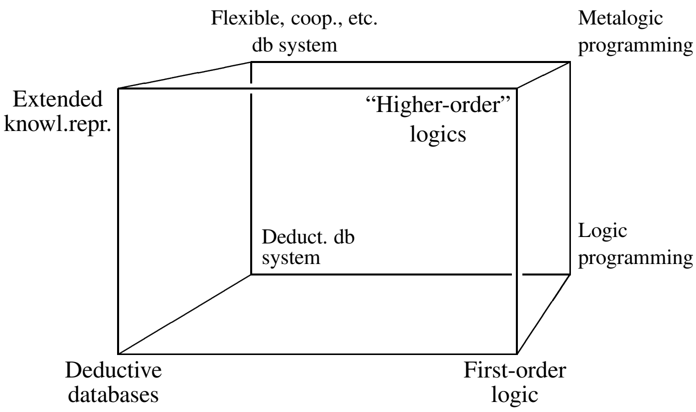
We indicate also how techniques developed in metalogic programming for abduction and induction can be applied for modelling construction and maintenance tasks such as view updates, knowledge discovery and data mining.

1.1 Background

Metaprogramming can be defined as "treating programs as data" and as such, it has always been a central notion in computer science, from the first compilers or even before that, regarding, e.g., the seminal works by Gödel, Turing and Church in the thirties and forties. With the advent of symbolic programming programming languages such as Lisp and Prolog, metaprogramming has been recognized as a powerful and useful programming technique of its own right despite the slight increase in complexity. In logic programming, we may summarize the advantages of metaprogramming as follows,

- it is possible to write generic code, e.g., rules that goes for a group of predicates,
- flexibility in interpretation, the programmer can interfere with the semantics of the language, e.g., adding nonstandard inference rules or controlling the application of inference rules, and
- enhanced functionality, in the simplest case, adding, say, tracing capabilities to an interpreter and more radically, having an interpreter to "run backwards" in order to create programs.

Meta-programming can be viewed as a way to simulate features normally related to higher-order logics, but staying in a first-order setting and as such keeps open the perspectives for efficient implementation. We summarize the possible contributions of metalogic programming to the field of deductive databases by the following commuting cube.



The front of the cube represents theoretical settings on which the implemented technology, shown in the back, is founded. The point of view we defend in the present paper is that the extended power of *meta*programming is higly relevant for the development of new and more flexible knowledge representation formalisms and implemented systems.

Most work concerning metaprogramming in logic takes its origin in the extremely simple self-interpreter for Prolog known as Vanilla.

```
prove(true).
prove((A,B)):- prove(A), prove(B).
prove(A):- clause(A,B), prove(B).
```

Gaasterland, Godfrey, and Minker [17] has used an extension of Vanilla for describing cooperative answering based on relaxation by taxonomy, i.e, generalizing the query in case of an insufficient answer. In our own work [2; 3], we have taken this approach further allowing (in principle) arbitrary nonstandard inference rules kept in order by constraints on the proof produced by the `prove` predicate. Reflective Prolog [12] is a proposal for a programming language that integrates this programming-by-modifying-the-semantics style, making clear the reflections between the object and meta layers that take place. In [13], this framework is extended with a notion of metalevel negation which makes it possible to characterize aspects of nonmonotonic reasoning in an elegant way.

The metainterpretation approach can be taken further by making a representation of the object program an argument of the interpreter as is the case for the `demo` predicated which was suggested by Kowalski [22]. This means that a metavariable can represent an unknown "hole" in the program and in principle, `demo` should be able generate the remaining parts of the program as to make the goal argument provable. However, it took more than a decade before logically satisfactory implementations of `demo` appeared in two simultaneous results by [26; 7]; our own constraint-based Demo system [8; 9; 10] seems to be the first implementation which makes `demo` available as a general metaprogramming tool capable of handling arbitrary uninstantiated metavariables. Experiences with this systems shows that alternative reasoning patterns, e.g., abduction, induction and default reasoning, can be implemented in quite straightforward ways having `demo` to run in parallel with additional metalevel constraints defining the kinds of novel fragments that are allowed.

For an overview of the field of metaprogramming, we refer to the series of workshop proceedings [1; 6; 25; 16], two survey papers [20; 5] and a recent book [4]. We may also refer to the following entries for earlier work on flexible query-answering mechanisms [14; 18; 15; 23; 11].

1.2 Overview of this paper

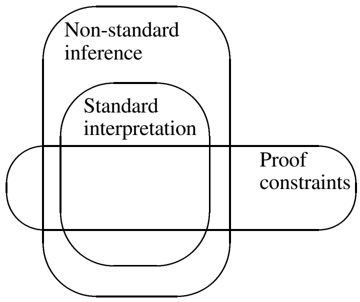
Section 2 describes our prototype for experimenting with nonstandard inference rules and constraints on the proof realized by a few straightforward extensions to the Vanilla interpreter. We indicate application of this framework to model user requirements, extended answering capabilities as well as a new form of semantic optimizations.

In section 3, we sketch the overall principles in the Demo system and indicate its use for modelling dymanic aspects of databases such as view update, knowledge discovery and data mining.

We give a summary and some ideas for future work in section 4.

2 Flexible query-answering by extensions to the Vanilla interpreter

Our model for query-answering systems is presented as an extension of the vanilla interpreter, parameterized by a set of inference rules and with a reification of the proof, which makes possible the principle of having constraints on the proof as a way to direct the application of the inference rules. As shown by a picture, the introduction of new inference rules extend the answer whereas proof constraints reduce it.



It should also be stressed that for certain types of queries, that the proof contains information that is highly relevant for the user as part of the answer given by a system. Consider as an example a database for travel planning. A traditional deductive database system (or Prolog interpreter) can only answer whether or not a certain travel is possible, whereas the actual travel plan consisting of the subdistances making up the whole trip, is what is expressed in the proof.

2.1 Classification of the database clauses

The facts and rules in the database are represented by Prolog facts of the form

`klause(` *class* `,` *clause* `).`

Each clause is given a classification that determine the way it can be used by the interpreter.

The following database, which we use below to illustrate proof constraints, consists of clauses all classified as `db` indicating that they are to be understood as database clauses in the usual way.

```
klause(db, (rich:- steal)).
klause(db, (rich:- earn)).
klause(db, (earn:- true)).
klause(db, (steal:- true)).
```

Other relevant classifications can be `tax` to indicate taxonomy clauses intended also for widening a query or subgoal or `ic` for integrity constraints.

In this way, the database can be considered divided into separate knowledge bases of different kinds of domain knowledge. A classification may also be used to distinguish between the knowledge of different agents.

2.2 The interpreter

The Vanilla metainterpreter has been extended by an extra argument in order to collect the proof; notice also that the normal rewriting of an atom by a sequence of other atoms has been replaced by a predicate `derive` which we discuss below.

```
prove(true,E):- dempty(E).

prove((A,B), ProofAB):-
    dappend(ProofA, ProofB, ProofAB),
    prove(A, ProofA), prove(B, ProofB).

prove(A, ProofA):-
    derive(A, B, StepA),
    dadd( StepA, ProofB, ProofA),
    prove(B, ProofB).
```

The proof is a list of descriptions of proof steps, each generated by `derive`. The list predicates `dempty`, `dappend`, and `dadd` covers over an abstract data type of difference lists; the implementation is shown in the appendix. This representation is convenient for the following practical reasons,

- concatenation of subproofs is done in constant time, without recursion, and
- proof constraints can be defined orthogonally to the inference rules and still execute in a lazy-evaluation style by means of coroutines in Prolog during the construction of the proof.

The `derive` predicate in the metainterpreter should be understood as a parameter which represents the set of inference rules available. The usual *modus ponens* rule is defined as follows.

```
derive(A,B,Step):-
   klause(db, (A:- B)),
   Step = step(mp, (A:- B)).
```

This defines the standard interpretation of a database; nonstandard inference can be introduced by additional `derive` rules.

2.3 Implementing constraints on the proof

For illustrating the notions of the proof as part of the answer and constraints on the proof, we consider the query `rich` to the database shown above with *modus ponens* as the only rule; `dmake` is a coercion from normal lists to difference lists.

```
?- dmake(Proof, ProofD), prove(rich, ProofD).
```

This yields the following two values of `Proof` as answers.

```
Proof = [step(mp,(rich:- steal)),
                step(mp,(steal:-true))]
Proof = [step(mp,(rich:- earn)),
                step(mp,(earn:-true))]
```

We can identify two sorts of relevant proof constraints, posed by the user in the query language, and system constraints that characterize a particular query-answering system. A given query-answering system defined by a set of inference rules, a query language, and a set of system constraints may be implemented as follows.

```
answer(Q/Con, Proof):-
    system_constraints(Proof),
    user_constraints(Con, Proof),
    dmake(Proof, ProofD), prove(Q, ProofD).
```

The two constraint predicates should be implemented as coroutines that resume execution each time a step is added to the proof. We illustrate the principle by the following example.[1]

```
:- block honest(-).

honest([step(_, (X :- _)) | Steps]):-
    dif(X, steal),
    honest(Steps).

honest([]).
```

With this as a constraint to the query shown above, only the second proof will be produced as answer.

2.4 Relaxation by taxonomy

As an example of a nonstandard inference rule, we consider relaxation by taxonomy which can be realized adding the following meta-level rule to the interpreter.

```
derive(Sub,Super,Step):-
    klause(tax, (Super:-Sub)),
    Step = step(relax_by_tax, (Super:-Sub)).
```

[1] We use Sicstus Prolog (SICS, 1995). `dif` is a logically correct implementation of syntactic nonidentity. It delays until the arguments have been sufficiently instantiated. The `block` directive causes `honest` to delay until its argument gets instantiated. This control device does not affect the declarative meaning.

I.e., the taxonomy clause can be used in reverse compared with a normal *modus ponens* step. Taxonomy clauses should be classified in the database is such, e.g.:

```
klause( tax, (subdist(X,Y):-flight(X,Y))).
```

When queried for a travel composed recursively from one or more flights, the interpreter may additionally suggests alternative travels in which one or more subdistances is replaced by another means of transportation, e.g. using another taxonomy rule in the normal *modus ponens* direction.

```
klause(tax, (subdist(X,Y):-bus_ride(X,Y))).
```

Our approach, here, is inspired by earlier work of Gaasterland, Godfrey, and Minker [17] who performed similar transformations on the initial query, but without going into derived subgoals as we do.

We can illustrate the difference between the two by an example. Having submitted a query for a flight travel from Copenhagen to Budapest, the typical traveller may accept as an answer giving a flight from Copenhagen to Vienna followed a bus ride from Vienna to Budapest if for some reason the all flights into Budapest have been cancelled. The travel agent who only can modify the top level query would not suggest this solution, but instead go directly to suggesting a bus ride all the way from Copenhagen to Budapest. Our traveller is likely not to consider this travel agent very cooperative.

The flexible use of relaxation by taxonomy of arbitrary subgoals, creates another problem, which motivates our notion of proof constraints. If taxonomy clauses can be used also in *modus ponens* steps, this may immediately "undo" the relaxation and it is easy to see that the interpreter is condemned to loop. The following will cure the problem.

PROOF CONSTRAINT: A given instance of a taxonomy rule cannot be used in a relaxation as well as in a *modus ponens* step.

Referring still to the travel planning example, proof constraints may also be used to express natural requirements such as "No intermediate station should be passed more than once" or that the prize and/or travelling time should be minimized.

2.5 Introducing a fragment of linear logic to databases

One of the motivations behind the development of linear logic [19] is to make it possible to reason about aspects of process and time in a logical setting. Linear logic differs from first order logic in the way that some formulas are considered as resources in the sense that they are consumed when used in a proof. This can implemented in our framework as follows.

PROOF CONSTRAINT: A clause classified as `resource` can only occur once.

We have not made any systematic investigation of this option yet or developed interesting examples, but we believe it to be a relevant extension to deductive databases.

2.6 Counterfactual exceptions

It may often be relevant in a query to suppress part of the database, which we so to speak counterfactually deny. For example, asking for a travel without flights can be thought of as asking for a travel in a world similar to the real world, but with all flights cancelled, despite the fact that the real world as well as the database include flights. The example in section 2.3 above, about getting rich in an honest way, is also a very simple special case of the principle, we introduce here.

We developed the notion of counterfactual exceptions in order to express such queries using the general interpreter described above and based on the knowledge gained from it, we have been able to describe model-based and completion semantics for this device as well as giving a specialized metainterpreter for it; this is described in our ECAI paper [3].

Here we need only the standard inference defined by *modus ponens*[2] and consider queries of the form

$$\exists \cdots (\phi \twoheadrightarrow \psi)$$

with

$$\phi = (\forall \cdots \neg \phi_1) \wedge \cdots \wedge (\forall \cdots \neg \phi_n)$$

where $\phi_1, \ldots, \phi_n$ are atoms, ψ a conjunction of atoms; each subformula $\forall \cdots \neg \phi_i$ is called a *counterfactual exception*. Any variable quantified at the outermost level is said to be *global*, all other variables in the ϕ_i's are *local*. For a given user query $\exists \cdots (\phi \twoheadrightarrow \psi)$, ψ is made the goal argument of the metainterpreter whereas ϕ is translated into proof constraints as follows.

PROOF CONSTRAINT: A clause instance *A* :- *B* is only allowed in the proof if *A* and ϕ are consistent.

This consistency condition corresponds roughly to a condition of non-unifiability which can be implemented using the lazy `dif` predicate described earlier.

The treatment of negative hypotheses as exceptions is computationally much easier to handle than the possible world counterfactual implication suggested by Lewis in [24] and adopted in most studies of counterfactual reasoning. Although the latter view may be philosophically more pleasing in many context, our simplified version seems appropriate in database queries as shown by the following examples. We assume a database of travel information where a *travel* between two points is composed of one or more *link*s, which may be either *train*, *boat*, or *flight*.

The query "I want to travel from a to d, but I refuse to sail from b to c on my way", is formalized

$$(\neg boat(b, c)) \twoheadrightarrow travel(a, d).$$

"I want to travel from a to d, but I refuse to fly":

$$(\forall X, Y \neg flight(X, Y)) \twoheadrightarrow travel(a, d).$$

We can show the use of global variables in the query "I want to travel from a to a place where I do not arrive by train".

$$\exists X((\forall Y \neg train(Y, X)) \twoheadrightarrow travel(a, X))$$

These examples show that many natural requirements in a query which cannot be expressed in any traditional query language fits quite well with constraints on the proof, here in the special fitting called counterfactual exceptions.

2.7 Semantic optimization by proof constraints

Semantic optimization is a method to restrict the search space by extending the query by means of intensional knowledge, e.g., contained in integrity constraints.

Assume, for example, an integrity constraint

$$\forall X(p(X) \wedge q(X) \rightarrow r(X)).$$

In case the extension of r is know to be small compared with the rest of the database, the query $s(X), p(X), q(X)$ can be extended to $r(X), s(X), p(X), q(X)$ without changing the answer but with a much faster evaluation of the query.

We can go a step further extending a query with counterfactual exceptions. Assume, for example, the following integrity constraint,

$$\forall X(s(X) \rightarrow \neg r(X) \wedge X \neq a).$$

[2]However, it is clear that the principle can be combined with nonstandard inference and other proof constraints as well.

This means that we can extend the query $\exists X\, s(X)$ with exceptions as follows without changing the answer.

$$\exists X((\neg r(X) \wedge \neg s(a)) \twoheadrightarrow s(X)).$$

This affects the execution in the following ways,

- whenever the subgoal $r(X)$ appears for an X sought, it fails immediately,
- whenever the subgoal $s(a)$ appears, it fails immediately without consulting the extension of s.

In certain cases this can lead to a drastic reduction of the search space and it should be compared with the fact that the processing of counterfactual exceptions only amounts to a constant slowdown of each proof step performed.

We consider another example with an integrity constraint

$$\forall X(p(X) \wedge q(Y) \rightarrow X \neq Y).$$

Here the query $\exists X\, p(X)$ can be extended to

$$\exists X(\neg q(X) \twoheadrightarrow p(X)).$$

3 Using a complete demo predicate to model dynamic aspects of databases

In this section, we take up a different theme in metalogic programming which seems to be relevant when modelling dynamic properties of databases such as updating and knowledge discovery.

A proof predicate such as the two-argument `demo` is well-suited for specifying such problems, and thus it is obvious to use our implemented version of it for experimental purposes. Our metalogic programming system called Demo differs from earlier implementations by providing a fully logical treatment of metavariables standing for unknown parts of the object program interpreted by `demo`. The `demo` predicate can be specified as follows.

$\texttt{demo}(P', Q')$ iff P' and Q' are names of program and query, P and Q, such that there exists substitution σ with

$$P \vdash Q\sigma$$

A meta-variable in P' will thus stand for a piece of program text and `demo` will produce program fragments which make Q provable. The implementation is fairly efficient due to the use of constraint techniques and the usefulness of the approach comes from the ability to have user-defined conditions to the program fragments sought run interleaved with the actions of demo.

The full description of Demo is given in [10]; here we give a brief overview focusing on potential database applications.

3.1 View update by abduction

We use an example from [21] as an introduction the use of Demo for database application. We have retouched away a few technical details, that are unnecessary for the points we want to illustrate here; all details can be found in [10].

We consider a database with *extensional* predicates `father` and `mother` and *view* predicates `sibling` and `parent`. We assume an initial database with the following contents; the `object_module` directive recognized by the Demo system associates the database with the name `db0`, the backslash is a quotation operator that indicates a ground representation.

```
:- object_module( db0,
  \[ (sibling(X,Y):- parent(Z,X),parent(Z,Y),
```

```
                   dif(X,Y)),
      (parent(X,Y):- father(X,Y)),
      (parent(X,Y):- mother(X,Y)),
       father(john,mary),
       mother(jane,mary)      ]).
```

The father and mother predicates being the only extensional predicates means that new knowledge has to be absorbed in the database solely by facts about these predicates, also if the knowledge is reported in terms of the view predicates. We formalize as follows — at the metalevel — what it means for a database (extension) to consist such facts.

```
extensionals(\ []).

extensionals(\ [(father(?A,?B):-true)
                              | ?More]):-
    constant_(A), constant_(B),
    extensionals( More ).

extensionals(\ [(mother(?A,?B):-true)
                              | ?More]):-
    constant_(A), constant_(B),
    extensionals( More ).
```

The question mark is an unquote operator that indicates the presence of a metavariable, so together with the indicated syntax constraints, it is expressed above that the arguments, whose names are given by A and B must be constants (i.e., not variables or arbitrary Prolog structures). Co-routine control is assumed for delaying this metalevel predicate, exactly as described for proof constraints above in section 2.

Integrity constraints for a knowledge base also needs to be defined at the metalevel.

```
integrity_check(DB):-

% You can only have one father:
for_all(
(constant_(A),constant_(B),constant_(C),
 demo(DB, \\ (father(?A,?C),father(?B,?C)))),
A=B ),

% You can only have one mother:
for_all(
(constant_(A),constant_(B),constant_(C),
 demo(DB, \\ (mother(?A,?C) mother(?B,?C)))),
A=B ),

% A mother cannot be a father:
for_all(
(constant_(A),constant_(B),
 demo(DB, \\ (mother(?A,?_),father(?B,?_)))),
dif(A,B) ).
```

We have now what is needed to implement a predicate for updating the database properly so new knowledge can be explained.

```
update(DB, Obs, NewDB):-
    extensionals(UpdateFacts),
    NewKB = \ ( ?DB & ?UpdateFacts ),
    demo( NewDB, Obs ),
    integrity_check( NewDB ).
```

Given a data DB and some observed facts Obs, a new knowledge base NewDB is produced. The knowledge base is extended with new extensional facts without violating the integrity constraints. The expression $P_1 \& P_2$ denotes the program consisting of the union of the clauses of P_1 and P_2.

The following test queries show the overall behaviour of the update predicate defined above.

```
?- update( \kb0, \\sibling(mary,bob), N).

N = \ (kb0 & [(father(john,bob):-true)]) ? ;

N = \ (kb0 & [(mother(jane,bob):-true)]) ?

?- update( \kb0, \\ (sibling(mary,bob),
                     mother(joan,bob)), N).

N = \ (kb0&[(father(john,bob):-true),
            (mother(joan,bob):-true)]) ?
```

So the update predicate reasons in an abductive way in order to explain the observed facts and in this way suggests the possible ways the extensional database can be updated in order to become consistent with the world. If there is only one possible update, it can be executed right away, otherwise more information may be required from the user.

3.2 Using induction for data mining or knowledge discovery

Under this headline, we consider the general problem of identifying appropriate rules in order to identify automatically a structuring inherint in a large set of data given in an unstructured way, in this context, typically in terms of a set of facts.

To exemplify this, we modify the example above by deleting the rule defining the sibling relation and introduce a few more extensional facts.

```
:- object_module( db1,
   \[ (parent(X,Y):- father(X,Y)),
      (parent(X,Y):- mother(X,Y)),
       father(john,mary),
       mother(jane,mary),
       father(john,bob),
       mother(jane,pedro),
       father(manuel,pedro) ]).
```

Assume now, a new property named sibling is reported with the facts

$\mathcal{F}$ = sibling(mary bob), sibling(mary pedro).

We do not accept any new extensional predicates added to the database, so the only way to assimilate the new facts will be by a new rule defining the sibling predicate in terms of other predicates in the database. The problem can be stated as follows, where we will discuss the possible choices of the metalevel predicate simple_rule below.

?- simple_rule(R), demo(\ (db1 & ?R), $\mathcal{F}$).

It may be the case that the only rules we allow should correspond to either a natural join, a union or intersection of two existing predicates defined in a suitable way (in the first place, if this fails, we may extend the scope to cover more complicated rules). With this, the query to demo above will suggest the rule

```
sibling(X,Y):- parent(Z,X),parent(Z,Y).
```

With more sophisticated metalevel rules it may even be possible to have the condition dif(X,Y) added to the rule.

It should be stressed, however, that the Demo system only have been used for small induction problems as the one shown above.

In [10] we have also shown how induction can be made with Demo under assumption corresponding to default logic, so that Demo invents the rule "all birds fly, except penguins" from a suitable collection of facts.

4 Concluding remarks

We have advocated the use of metalogic programming as a powerful tool suited for experimenting with new mechanisms in deductive databases, ranging from the design of query languages to "flexible" or "cooperative" ways of answering queries.

We showed a straightforward extension of the Vanilla interpreter which served as a generic model for flexible query-answering systems using nonstandard inference combined with proof constraints. We are not aware of any earlier work that uses constraints on the proof in this way, and we have intended to show that this notion is highly relevant in the statement as well as the evaluation of database queries. As a special case, we considered the notion of counterfactual exceptions and which also gave rise to a novel kind of semantic optimizations.

Furthermore, we sketched how our Demo system, with its logically complete `demo` predicate, might be used to model dynamic properties related to the construction and maintenance of databases. With our current experience with Demo for abductive and inductive problems, we believe that it is useful for formulating and experimenting with new models for these aspects. However, it needs more work before we can conclude anything about whether it is relevant to use it as a platform for implementation methods that can be scaled up to realistic problems. A possible next step will be to try to integrate the two paradigms we have shown.

Appendix, difference lists

The following Prolog unit clauses defines the abstract data type used for difference lists

```
% Append two diff. lists
dappend(L1/L2, L2/L3, L1/L3).

% Add element to front of list
dadd(E,L1/L2,[E|L1]/L2).

% Normal list to diff. lists
dmake(L, L/[]).

% Empty diff. list
dempty(L/L).
```

References

[1] Abramson, H., and Rogers, M.H., eds., *Meta-programming in Logic Programming*. MIT Press, 1989.

[2] Andreasen T., Christiansen H. An experimental prototype for flexible query-answering mechanisms, A metainterpretation approach. In: [11], 1996.

[3] Andreasen, T., Christiansen, H. Counterfactual exceptions in deductive database queries. *Proc. ECAI'96, 12th European Conference on Artificial Intelligence* pp. 340–344, 1996.

[4] Apt, K.R., Turini, F., eds., *Meta-Logics and Logic Programming*, MIT Press 1995.

[5] Barklund, J., Metaprogramming in Logic. In: Encyclopedia of Computer Science and Technology, Vol. 33 (eds. A. Kent and J. G. Williams), pp. 205–227, Marcel Dekker, New York, 1995.

[6] Bruynooghe, M., ed., *Proc. of the Second Workshop on Metaprogramming in Logic*. April 4–6, 1990, Leuven, Belgium.

[7] Christiansen, H., A complete resolution method for logical meta-programming languages. Proc. of META-92, Third International Workshop on Metaprogramming in Logic. Ed. Pettorossi, A., *Lecture Notes in Computer Science* 649, Springer-Verlag, pp. 205–219, 1992.

[8] Christiansen, H., Efficient and complete demo predicate. for definite clause languages. *Datalogiske skrifter* 51, Roskilde University, 1994.

[9] Christiansen, H., On proof predicates in logic programming. A.Momigliani and M.Ornaghi, eds. *'Proof-Theoretical Extensions of Logic Programming'*, CMU, Pittsburgh, PA 15231-3890, USA. Proceedings of an ICLP-94 Post-Conference Workshop, 1994.

[10] Christiansen, H., Automated reasoning with a constraint-based meta-interpreter. *To appear* 1996.

[11] Christiansen, H., Larsen, H.L., Andreasen, T., Eds. Flexible Query-Answering Systems, Proc. of the 1996 workshop (FQAS96), Roskilde, Denmark, May 22–24, 1996. *Datalogiske skrifter* 62, Roskilde University, 1996.

[12] Costantini, S., Lanzarone, G.A., A metalogic programming language, *Logic Programming: Proc. of the Sixth International Conference*, pp. 133–145, MIT Press, 1989.

[13] Costantini, S., Lanzarone, G.A., Metalevel negation and non-monotonic reasoning. *Methods of Logic in Computer Science* 1, pp. 111–140, 1994.

[14] Cuppens F. and Demolombe R. Cooperative Answering : a methodology to provide intelligent access to Databases. in Pro-

ceedings Proc. of the Second International Conference on Expert Database Systems. 1988.

[15] Demolombe, R., Imielinski, R., eds., *Nonstandard Queries and Nonstandard Answers*, Studies in Logic and Computation 3, Oxford Science Publications, 1994.

[16] Fribourg, L., Turini, F., Eds. Logic Program Synthesis and Transformation — Meta-Programming in Logic. 4th International Workshops, LOBSTR'94 and META'94. *Lecture Notes in Computer Science* 883, Springer-Verlag, 1994.

[17] Gaasterland T., Godfrey P., and Minker J., Relaxation as a Platform for Cooperative Answering. *Journal of Intelligent Information Systems*, 1, 3/4, pp. 293-321, 1992.

[18] Gaasterland T., Godfrey P., and Minker J., An Overview of Cooperative Answering. Journal of Intelligent Information Systems, 1, 2, 1992. p. 123–157.

[19] Girard, J.Y., Linear logic. *Theoretical Computer Science* 50, pp. 1-101, 1987.
International Logic Programming Symposium, 1991.

[20] Hill, P.M. and Gallagher, J.P., Meta-programming in Logic Programming. To be published in Volume V of *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press.
Currently available as *Research Report Series* 94.22, University of Leeds, School of Computer Studies, 1994.

[21] Kakas, A.A., Kowalski, R.A., Toni, F., Abductive logic programming. *Journal of Logic and Computation* 2, pp. 719–770, 1993.

[22] Kowalski, R., *Logic for problem solving*. North-Holland, 1979.

[23] Larsen, H.L., Andreasen, T., Flexible Query-Answering Systems, Proc. of the 1994 workshop (FQAS94), Roskilde, Denmark, Nov. 14–16, 1994. *Datalogiske skrifter* 58, Roskilde University, 1995.

[24] Lewis, D, *Counterfactuals*. Harward University Press, 1973.

[25] Pettorossi, A., ed. Proc. of META-92, Third International Workshop on Metaprogramming in Logic. *Lecture Notes in Computer Science* 649, Springer-Verlag, 1992.

[26] Sato, T., Meta-programming through a truth predicate. *Logic Programming, Proc. of the Joint International Conference and Symposium on Logic Programming*, ed. Apt, K., pp. 526–540, MIT Press, 1992.

[27] *SICStus Prolog user's manual*. Version 3.0, SICS, Swedish Institute of Computer Science, 1995.