Practical Applications of Extended Deductive Databases in DATALOG*

Dietmar Seipel

University of Würzburg, Department of Computer Science
Am Hubland, D – 97074 Würzburg, Germany
seipel@informatik.uni-wuerzburg.de

Abstract. A wide range of additional forward chaining applications could be realized with deductive databases, if their rule formalism, their immediate consequence operator, and their fixpoint iteration process would be more flexible.
Deductive databases normally represent knowledge using stratified DATALOG programs with default negation. But many practical applications of forward chaining require an extensible set of user–defined built–in predicates. Moreover, they often need function symbols for building complex data structures, and the stratified fixpoint iteration has to be extended by aggregation operations.
We present an new language DATALOG*, which extends DATALOG by stratified meta–predicates (including default negation), function symbols, and user–defined built–in predicates, which are implemented and evaluated top–down in PROLOG. All predicates are subject to the same backtracking mechanism. The bottom–up fixpoint iteration can aggregate the derived facts after each iteration based on user–defined PROLOG predicates.

Keywords.
Deductive databases, PROLOG, forward / backward chaining, bottom–up, top–down, built–in predicates, stratification, function symbols, XML

1 Introduction

Deductive databases allow for efficiently deriving inferences from flat tables using forward chaining and relational database technology. Most deductive database systems support the representation language of stratified DATALOG$^{\mathsf{not}}$ [8, 13]. Disallowing function symbols and enforcing the condition of range–restrictedness (safety)[1] guarantees that the inference process will always terminate. In principle, the fixpoint interation based on forward chaining can also be applied to the extension by function symbols, but termination is not always guaranteed, if function symbols can occur in rule heads. Other DATALOG extensions allow for a very limited set of built–in predicates under an extended safety condition. In DATALOG$^{\mathsf{not}}$, stratified[2] default negation is possible.

[1] all variable symbols of a rule must occur in the positive body
[2] there is no cycle including default negation in the predicate dependency graph

DATALOG rules extended by existentially quantified variables in rule heads are known as *tuple generating dependencies*. They have been used for enabling ontological knowledge representation and efficient reasoning [7] and for specifying generalized schema mappings in databases [12]. In the former paper, also stratified default negation has been considered.

For handling non–stratified default negation and disjunctive rule heads, *answer set programming* (ASP) can be used [2, 11]; the advantage of an efficient query evaluation for large relational databases is partly lost in ASP, but much more complex problems – of a high computational complexity – can be encoded elegantly. Today, even larger graph–theoretic problems, such as graph colouring, can be encoded in a very compact way and solved by ASP in reasonable time.

On the other hand, *logic programming* in PROLOG [5, 9] is not as declarative as deductive databases and ASP, but backward chaining and side effects make it a fully fledged programming language. PROLOG allows for function symbols, and it can handle stratified default negation. However, recursion can lead to non–termination, even if there are no function symbols. Although the PROLOG extension XSB can solve this termination problem using tabling (memoing) techniques, there exist many applications where backward chaining is not suitable.

Usually, dedicated special–purpose problem solvers are used for more general *forward chaining* problems. Due to their lack of declarativity, these solvers often are hard to maintain, difficult to extend and port to similar application domains. Thus, we are proposing *extended deductive databases* (EDDB) based on a DATALOG extension named DATALOG*, which combines declarative forward chaining with meta–predicates, function symbols, and built–in predicates implemented in PROLOG. So far, we have used DATALOG* for the following forward chaining applications:

- diagnostic reasoning in medical or technical domains, e.g., *d3web* [14] or root cause detection in computer networks,
- anomaly detection in ontologies extended by rules, such as the extension SWRL of the ontology language OWL, and
- meta–interpreters, e.g. for disjunctive reasoning with the hyperresolution consequence operator $\mathcal{T}_\mathcal{P}^s$ in disjunctive deductive databases.

These practical EDDB applications require PROLOG meta–predicates (such as default negation `not/1` and the list predicates `findall/3` and `maplist/2,3`), recursion on cyclic data, and function symbols for representing complex data structures, such as lists or semi–structured data and XML [1]. Sometimes, the standard conjunctive rule bodies are not adequate: the knowledge representation becomes too complicated, and the evaluation is unnecessarily complex due to redundancy, if rules with non–conjunctive rule bodies are normalized to sets of rules with conjunctive rule bodies. Recently, [4] has defined an extended version of range–restricted DATALOG rules with non–conjunctive rule bodies.

In general, meta–predicates need to be stratified to ensure termination. It is easy to decide if a DATALOG$^{\mathsf{not}}$ program can be stratified. But DATALOG*

programs with arbitrary PROLOG meta–predicates have to be analysed carefully using heuristics based on extended call graphs to find out which predicates call which other predicates through meta–predicates. Although this problem is undecidable in general, suitable heuristics have been published in [17], even for the more general context of PROLOG.

The rest of this paper is organized as follows: In Section 2, we indicate how DATALOG* mixes forward chaining with PROLOG's backward chaining for built–in predicates. Section 3 presents three case studies for DATALOG*. In Section 4, we describe a possible meta–interpreter for DATALOG*, which we have implemented in PROLOG. Finally, we give conclusions and sketch some future work.

2 The General Idea of DATALOG*

We distinguish between DATALOG* rules and PROLOG rules. Syntactically, DATALOG* rules are PROLOG rules; i.e., they may contain function symbols (in rule heads and bodies) as well as negation, disjunction, and PROLOG predicates in rule bodies. As forward chaining rules, DATALOG* rules are evaluated bottom–up, and all possible conclusions are derived. The supporting PROLOG rules are evaluated top–down, and – for efficiency reasons – only on demand, and they can refer to DATALOG* facts. The PROLOG rules are also necessary for expressivity reasons: they are used for computations on complex terms, and – more importantly – for computing very general aggregations of DATALOG* facts.

DATALOG* rules cannot be evaluated in PROLOG or DATALOG alone for the following reasons: Current DATALOG engines cannot handle function symbols and non–ground facts, and they do not allow for the embedded computations (arbitrary built–in predicates), which we need for our practical applications. Standard PROLOG systems may loop in recursion forever (e.g., when computing the transitive closure of a cyclic graph), and they may be inefficient, if there are subqueries that are posed and answered multiply. Thus, they have to be extended by some DATALOG* facilities (our approach) or memoing/tabling facilities (the approach of the PROLOG extension XSB).

Since we need forward chaining, and since the embedding system DDK [15] is developed in SWI–PROLOG, we have implemented a new inference machine in standard PROLOG that can handle mixed, stratified DATALOG*/PROLOG rule systems. The evaluation of a DATALOG* program $\mathcal{D}$ mixes forward–chained evaluation of DATALOG with SLDNF–resolution of PROLOG, see Figure 1. The body atoms B_i of a DATALOG* rule $A \leftarrow B_1 \wedge \ldots \wedge B_n$ are evaluated backward in PROLOG based on previously derived facts and based on a PROLOG program $\mathcal{P}$ using SLDNF–resolution.

Due to the PROLOG evaluation of rule bodies, variable symbols appearing only under default negation are implicitely quantified as *existential*. In contrast to [6], we do not need an explicit program transformation. For *range–restrictedness*, we just require that all variable symbols appearing in the head of a rule must also occur in at least one positive body atom. Moreover, similarly to PROLOG,

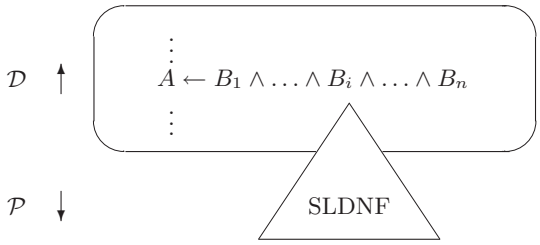

Fig. 1. Mixing Forward and Backward Chaining.

the programmer has to ensure that the standard left–to–right evaluation of the rule bodies will be adequate; in our case, on backtracking it should instantiate all variable symbols of the head in finitely many ways. This will guarantee, that every single iteration of the fixpoint process for DATALOG* derives only a finite set of ground facts. In contrast to [4], we do not allow a reordering of the body literals during the evaluation, since built–in predicates in DATALOG* can be arbitrary PROLOG predicates including meta–predicates and predicates with side effects.

Because of the embedded calls to PROLOG, a formal definition of the semantics of DATALOG* would be rather technical and difficult to understand. Instead, we will describe a compact meta–interpreter for DATALOG* in Section 4.

3 Case Studies for DATALOG*

In the following, we present three case studies for DATALOG* that require forward chaining together with built–in and user–defined PROLOG predicates. These practical applications could not be implemented so elegantly in DATALOG or PROLOG alone.

3.1 Diagnostic Reasoning

Diagnostic reasoning in *d3web* [14] requires forward chaining and built–in predicates for invoking user dialogs. We have implemented diagnostic reasoning in a declarative way using DATALOG*, cf. [18]. The DATALOG* rules use meta–predicates (such as `not/1`, `m_to_n/2`, `maplist/3`, and `findall/3`), and after each iteration of the immediate consequence operator, the derived facts are aggregated by combining the scores of different derivations of the same diagnosis.

For example, the first DATALOG* rule below assigns the value `1` to the intermediate diagnose `I3` by combining the answers to the questions `Q1`, `Q2`, and `Q3` during an interactive examination dialog. The second diagnositic rule assigns a score `16` to the diagnose `D2` based on the intermediate diagnose `I3` and the answer to the question `Q4`.

```
finding('I3' = 1) :-
   condition('Q1' = 2),
   ( condition('Q2' = 3)
   ; condition('Q3' = 2) ).

diagnosis('D2' = 16) :-
   condition('I3' = 1),
   condition('Q4' = 5).
```

The questions are asked while the DATALOG* rules are evaluated. An example of an interactive user dialog is given in Figure 2.

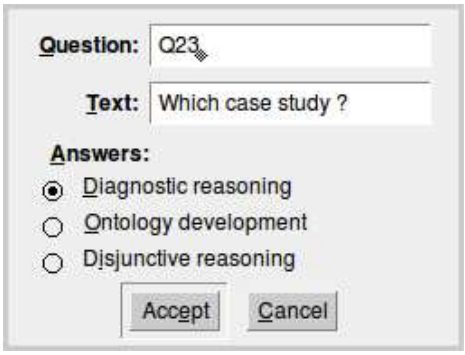

Fig. 2. User Dialog.

If a question `Qid` has not been asked yet, then a body atom `condition(Qid` Θ `V)` in a DATALOG* rule causes the call `dialog(Qid = Val)` of a suitable PROLOG dialog for determining the answer `Val` to the question. This value `Val` is then asserted in the PROLOG database as an atom `finding(Qid = Val)`, and it can be compared with `V` using the operator Θ, which is called `Comparator` below.

```
condition(C) :-
   C =.. [Comparator, Qid, V],
   ( finding(Qid = Val)
   ; dialog(Qid = Val),
     assert(finding(Qid = Val)) ),
   apply(Comparator, [Val, V]).
```

We can prevent the same instance of a DATALOG* rule from firing twice. Given a range–restricted DATALOG* rule `Head :- Body`, the following modified rule will fire at most once:

```
Head :-
   Body,
   not( has_fired(Head, Body) ),
   assert( has_fired(Head, Body) ).
```

The modified rules can be obtained automatically by a simple program transformation.

For diagnostic reasoning in *d3web*, all rules are ground, and it is necessary that each rule can fire at most once. The following aggregation predicate adds the scores in the list of derived facts for the same diagnosis; the other facts remain unchanged:

```
d3_aggregate_facts(I, J) :-
   findall( diagnosis(D=S),
      ( bagof( T, member(diagnosis(D=T), I), Ts ),
        add(Ts, S) ),
      J1 ),
   findall( A,
      ( member(A, I),
        not( functor(A, diagnosis, 1) ),
      J2 ),
   append(J1, J2, J).
```

3.2 Ontology Development

For the development of practical semantic applications, ontologies are commonly used with rule extensions. The integration of ontologies creates new challenges for the design process of such ontologies, but also existing evaluation methods have to cope with the extension of ontologies by rules. Since the verification of OWL ontologies with rule extensions is not tractable in general, we propose to *verify* and *analyze* ontologies at the symbolic level by using a declarative approach based on DATALOG*, where known *anomalies* can be easily specified and tested in a compact manner [3].

Our DATALOG* implementation requires meta–predicates such as `setof/3` and `maplist/2` for aggregation in rule bodies; moreover, for convenience, the junctor or (";") is used in addition to and (",") in rule bodies. The DATALOG* program can be *stratified* into two layers $\mathcal{D}_1$ and $\mathcal{D}_2$ of DATALOG* rules; below, we show a few of the rules. The rules for the predicates `anomaly/2` and `tc_derives/2` are part of the upper layer $\mathcal{D}_2$, and the rules for `derives/2`, `sibling/2`, and `disjoint/2` are part of the lower layer $\mathcal{D}_1$. $\mathcal{D}_1$ is applied to the DATALOG* facts for the basic predicates (such as `subclass_of/2`), which have to be derived from an underlying rule ontology. The resulting DATALOG* facts

are the input for $\mathcal{D}_2$. The stratification is necessary, because $\mathcal{D}_2$ refers to $\mathcal{D}_1$ through negation and aggregation.

```
anomaly(circularity, C) :-
   tc_derives(C, C).

anomaly(lonely_disjoint, C) :-
   class(C), siblings(_, Cs), disjoints(C, Cs),
   not( sibling(C, M), disjoint(C, M) ).
```

Firstly, an obvious equivalence exists between a subclass relationship between two classes C and D and a rule $A \leftarrow B$ with a single body atom B, such that A and B have the same arguments and the unary predicate symbols D and C, respectively. Thus, we combine them into the single formalism `derives/2` and compute the transitive closure in DATALOG*. Every class C contained in a cycle forms an anomaly, which is detected as `tc_derives(C, C)`.

```
tc_derives(X, Y) :-
   derives(X, Y).
tc_derives(X, Y) :-
   derives(X, Z), tc_derives(Z, Y).
```

Secondly, a class C is called a *lonely disjoint*, if it is disjoint to a set of siblings, and it does not have a sibling M with which it is disjoint. The first of the following PROLOG rules *aggregates* the siblings `Y` of a class `X` to a list `Ys` using the meta–predicate `setof/3`, and the second PROLOG rule tests if a given class `X` is disjoint to all classes in the list `Ys` using the meta–predicate `maplist/2`:

```
siblings(X, Ys) :-
   setof( Y, sibling(X, Y), Ys ).

disjoints(X, Ys) :-
   maplist( disjoint(X), Ys ).
```

The call to `setof/3` succeeds for every class `X` having siblings, and it computes the list `Ys` of all siblings `Y` of `X`; on backtracking, the siblings of the other classes `X` are computed. This means, `setof/3` does a *grouping* on the variable `X`. The rule for `siblings/2` could also be evaluated as a forward rule, but the rule for `disjoints/2` could not, since it is not range–restricted.

The lower layer $\mathcal{D}_1$ contains the following rule. We treat it as a DATALOG* rule, instead of a PROLOG rule, since we want to derive all pairs of siblings.

```
sibling(X, Y) :-
   subclass_of(X, Z), subclass_of(Y, Z), X \= Y.
```

3.3 Disjunctive Reasoning

In disjunctive deductive databases [11], the definite consequence operator $\mathcal{T}_\mathcal{P}$ has been generalized to the disjuntive hyperresolution operator $\mathcal{T}^s_\mathcal{P}$:

$$\mathcal{T}^s_\mathcal{P}(S) = \{\, C \vee C_1 \vee \ldots \vee C_m \mid C, C_1, \ldots, C_m \in D_{HB_\mathcal{P}} \text{ and there is a rule } C \leftarrow B_1 \wedge \ldots \wedge B_m \in gnd\,(\mathcal{P}) : \forall\, i \in \langle\, 1, m \,\rangle : B_i \vee C_i \in S \,\}.$$

Encoding disjunctive reasoning in DATALOG* requires built–in predicates for standard operations on disjunctions and disjunctive Herbrand states $S \subseteq D_{HB_\mathcal{P}}$, such as disjunction, union, and subsumption. A disjunction can be represented as a list of atoms. In [16], a disjunctive rule $r = C \leftarrow B_1 \wedge \ldots \wedge B_m$ without default negation is translated to a definite DATALOG* rule

$$dis(C_0) \leftarrow dis(B_1, C_1) \wedge \ldots \wedge dis(B_m, C_m) \wedge merge([C, C_1, \ldots, C_m], C_0),$$

where $B_1, \ldots, B_n$ are the body atoms of r, the list C represents the head of r, and $C_0, C_1, \ldots, C_n$ are distinct fresh variables.

The PROLOG calls $dis(B_i, C_i)$ ground instantiate the atoms B_i, and they instantiate the variables C_i to lists of ground atoms. If r is range–restricted – i.e., all variable symbols in C occur in at least one of the body atoms B_i – then this will also instantiate C to a list of ground atoms. The call $dis(B_i, C_i)$ finds already derived disjunctions containing the atom B_i and returns the list C_i of the remaining atoms as follows:

```
dis(B, C) :-
   dis(D), delete_atom(B, D, C).
delete_atom(B, D, C) :-
   append(D1, [B|D2], D), append(D1, D2, C).
```

The PROLOG predicate *merge/2* computes the disjunction of a list of disjunctions. After each iteration of the immediate consequence operator, an *aggregation* operator eliminates subsumed disjunctions.

Alternatively, the disjunctive rule $r = C \leftarrow B_1 \wedge \ldots \wedge B_m$ can be represented as a DATALOG* fact of the form $rule(C\text{–}[B_1, \ldots, B_n])$; then, a single, generic DATALOG* rule is sufficient, namely the following rule:

```
dis(D) :-
   rule(C-Bs),
   maplist( dis, Bs, Cs ),
   merge([C|Cs], D).
```

In [16] it is shown, that also disjunctive rules with default negation can be translated while preserving the stable model semantics. It is known that the well–founded model is a subset of all stable models (considered as sets of literals). Thus, by computing the well–founded semantics of the resulting program, we could approximate the stable model semantics based on DATALOG*.

4 A Meta–Interpreter for DATALOG*

In the following, we sketch an inference engine for stratified DATALOG*, which we have used successfully for our particular applications. We have implemented it as a meta–interpreter using the well–known PROLOG system SWI [19]; the user dialogs have been built with its publicly available graphical API.

4.1 The Immediate Consequence Operator

The generalized immediate consequence operator $\mathcal{T}_{\mathcal{D},\mathcal{P}}$ operates on a forward program $\mathcal{D}$ and an auxilliary PROLOG program $\mathcal{P}$. In the implementation below, $\mathcal{D}$ is given as a list `Datalog` of rules, whereas $\mathcal{P}$ is stored in the modules `M` and `user` (the standard module) of the PROLOG database. The following predicate calls all rule bodies of `Datalog`; the calls are executed in the module `M`. The set `Facts` of derived head facts will be stored in `M` only afterwards.

```
tp_operator(Datalog, M, Facts) :-
   findall( Head,
      ( member(Head :- Body, Datalog),
        call(M:Body) ),
      Facts ).
```

For all body predicates of $\mathcal{D}$ there have to be either rules (or facts) in $\mathcal{D} \cup \mathcal{P}$ or `dynamic` declarations in $\mathcal{P}$; otherwise, a call to such a predicate would raise an exception. And there can be rules in both $\mathcal{D}$ and $\mathcal{P}$. A body predicate that is solely defined by rules in $\mathcal{P}$ which do not refer to predicates from $\mathcal{D}$ could be considered as a built–in predicate of $\mathcal{D}$. The rules of $\mathcal{P}$ are evaluated top–down using PROLOG's SLDNF–resolution.

Since the evaluated forward program $\mathcal{D}$ is not part of the PROLOG database, the forward rules do not call each other recursively within a single $\mathcal{T}_{\mathcal{D},\mathcal{P}}$–operator, and they cannot be called from backward rules. The recursion is only reflected in the bottom–up fixpoint iteration of $\mathcal{T}_{\mathcal{D},\mathcal{P}}$.

4.2 Managing and Aggregating Facts in a Module

We use two elementary predicates for asserting/retracting a given list `Facts` of facts in a module `Module` of the PROLOG database using the predicate `do/2` from the well–known `loops` package of Schimpf:

```
assert_facts(Module, Facts) :-
   foreach(A, Facts) do assert(Module:A).
retract_facts(Module, Facts) :-
   foreach(A, Facts) do retract(Module:A).
```

For adding a list `Facts` of facts to `Module`, we need to know the list `I` of facts that are already stored in `Module`. First, these facts are retracted from `Module`, then `I` and `Facts` are aggregated using a user–defined plugin predicate (if there is no such predicate, then no aggregation is done), and finally the result is asserted in `Module`.

```
aggregate_facts(Module, I, Facts, J) :-
   retract_facts(Module, I),
   ( aggregate_facts(I, Facts, J)
   ; ord_union(I, Facts, J) ),
   assert_facts(Module, J).
```

`aggregate_facts/3` is a plugin predicate that can be specified using application specific PROLOG rules. For example, for diagnostic reasoning in *d3web*, we use the following plugin predicate:

```
aggregate_facts(I, Facts, J) :-
   append(I, Facts, K),
   d3_aggregate_facts(K, J).
```

Also Δ–iteration with subsumption [10] could be implemented by a suitable plugin.

4.3 The Fixpoint Iteration with Aggregation

For a given set `Datalog` of forward rules, `tp_iteration/3` derives a set `Facts` in module `M`:

```
tp_iteration(Datalog, M, Facts) :-
   tp_iteration(Datalog, M, [], Facts).

tp_iteration(Datalog, M, I, K) :-
   tp_operator(Datalog, M, Facts),
   ( tp_terminates(I, Facts) -> K = I
   ; aggregate_facts(M, I, Facts, J),
     tp_iteration(Datalog, M, J, K) ).

tp_terminates(I, Facts) :-
   not(member(A, Facts), not(member(A, I))).
```

Given a set `I` of facts, `tp_iteration/4` derives a new set `K` in module `M`. The derived facts are stored in `M` and kept on the argument level (`I`, `J`, `K`). The

iteration terminates, if all derived facts have already been known. The derived facts in `M` are necessary for `tp_operator/3`. Since they are mixed with the facts and the rules of the auxilliary PROLOG program $\mathcal{P}$ in `M`, they cannot be extracted from `M` at the end of the iteration; therefore, they also have to be kept on the argument level.

4.4 Stratification

We have implemented a PROLOG library for the stratification of DATALOG* programs. Sophisticated, heuristic methods of program analysis, which can also be extended by the user, are used to determine embedded calls in PROLOG meta–predicates based on suitably extended call graphs, cf. [17]. Thus, we can partition a DATALOG* program into strata, which can be evaluated successively.

For stratification, we have to analyze $\mathcal{D} \cup \mathcal{P}$; thus, $\mathcal{P}$ has to be available on the argument level, too. In practice, all PROLOG rules in the system could be used by the immediate consequence operator; but we only need to analyze the portion $\mathcal{P}'$ of the rules that access the facts for DATALOG* predicates in the PROLOG database.

In DATALOG*, most PROLOG meta–predicates require a stratified evaluation (e.g., `not/1`, `findall/3`, `setof/3`, and `maplist/2,3`). Only the ASP extension of deductive databases and logic programming can handle non–stratified default negation (`not/1`) as well; but, ASP solvers do not support function symbols and general built–in predicates.

4.5 Side Effects in Forward Rules

In our current implementation, the forward rules are fired successively, and it is possible that the forward rules update the PROLOG database module using `assert/1` and `retract/1` in rule bodies. We call these updates *side effects* – in contrast to the assertions of derived facts done by `assert_facts/4`, that are inherent to our approach.

The temporary facts asserted by side effects need not be derived by the rules, but they can nevertheless be used by other forward rules. These temporary facts are hidden in the PROLOG database module, they are not derived by the immediate consequence operator, and normally they will not be part of the final result of the fixpoint iteration. However, if desired, the user can bring these hidden facts to the surface by suitable helper rules for deriving them. For example, the second rule in the DATALOG* program below is a helper rule for deriving the atom b that was asserted by the first rule:

```
a :- assert(b).
b :- b.
```

During fixpoint iteration, DATALOG* makes asserted facts available to further derivations by subsequent rules within the same iteration. In DATALOG, this could be simulated by the so–called *Gauss–Seidel* evaluation of deductive databases, cf. [8]. Consequently, a larger part of the transitive closure is computed during a single iteration for the following DATALOG* program:

```
tc(X, Y) :-
   arc(X, Y), assert(tc(X, Y)).
tc(X, Y) :-
   arc(X, Z), tc(Z, Y), assert(tc(X, Y)).
```

For example, given a graph containing – amoung others – the two edges $arc(a, b)$ and $arc(b, c)$, the transitive edge $tc(a, c)$ is already derived in iteration 1, since the asserted fact $tc(b, c)$ of the first rule can be used in the second rule together with $arc(a, b)$, whereas – without assert – it is only derived in iteration 2 under the standard *Jacobi* evaluation of deductive databases.

Due to PROLOG's evaluation, an `assert` statement in a DATALOG* rule $r = A \leftarrow B_1 \wedge \ldots \wedge B_n$, where $B_i = assert(X)$, is relevant for all subsequent rules and for all atoms within the body of the same rule r. This can be simulated without `assert` by Gauss–Seidel evaluation: let $\beta = B_1 \wedge \ldots \wedge B_{i-1}$, then we can replace r by a helper rule $r' = X \leftarrow \beta$ for deriving X followed by a reduced rule $r'' = A \leftarrow \beta \wedge B_{i+1} \wedge \ldots \wedge B_n$ without B_i. By rule extraction we could of course avoid the redundancy caused by the double evaluation of the conjunction β in r' and r''. The only remaining differences are, that r asserts the atom X only as a temporary fact, whereas r' derives X, and that r' is fully evaluated before r'', which makes all asserts of r' available to r''.

For avoiding problems, a more controlled use of update predicates for the PROLOG database could be required. For example, in our DATALOG* implementation of diagnostic reasoning, `assert` is used only after interactive dialogs. This does not effect the derivation process so drastically; the computation simply behaves as if all findings had been known before the computation.

5 Conclusions

We have presented an extension of deductive databases with a generalized immediate consequence operator and fixpoint iteration, called DATALOG*, that is useful for implementing practical EDDB applications nicely in a compact way. In future work, we will investigate the theoretical properties of the extensions.

The described *meta–interpreter* could efficiently handle the case studies on diagnostic reasoning and ontology development with a few thousand facts and rules. The main advantage of the PROLOG–based approach was the flexibility in modelling applications requiring more general concepts of forward reasoning than deductive databases usually offer. Moreover, the meta–interpreter can be easily

extended to fit further needs. For other potentially very large applications, such as disjunctive reasoning, we will conduct experimental evaluations to compare our proposal to other approaches in the future.

Based on the generalized immediate consequence operator $\mathcal{T}_{\mathcal{D},\mathcal{P}}$, it also seems to be posssible to implement an extended form of the magic sets transformation method for rules with non–conjunctive rule bodies in a very simple way.

So far, only stratified evaluation is possible for DATALOG*. But, it would be interesting to extend $\mathcal{T}_{\mathcal{D},\mathcal{P}}$ to handle non–stratified negation using ASP technology (stable or well–founded models). Sometimes, guessing strategies on the truth of special atoms can be used, whereas the whole extension of the called predicates has to be guessed, before a call to a meta–predicate such as `setof/3` can be evaluated.

Logic programming and extended deductive databases can be used as a declarative *mediator technology* between different data sources (like relational databases, XML databases/documents, and EXCEL sheets) and tools. We are planning to integrate similar diagnostic problem solvers by mapping them to DATALOG*, and to combine data mining tools by processing their input and output, such that a declarative data mining workflow can be specified in DATALOG*.

References

1. *S. Abiteboul, P. Bunemann, D. Suciu:* Data on the Web – From Relations to Semi–Structured Data and XML, Morgan Kaufmann, 2000.
2. *C. Baral:* Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2003.
3. *J. Baumeister, D. Seipel:* Smelly Owls – Design Anomalies in Ontologies, Proc. 18th International Florida Artificial Intelligence Research Society Conference, FLAIRS 2005, AAAI Press, 2005, pp. 215–220.
4. *S. Brass:* Range Restriction for General Formulas, Proc. 22nd Workshop on (Constraint) Logic Programming, WLP 2009.
5. *I. Bratko:* PROLOG– Programming for Artificial Intelligence, 3rd Edition, Addison–Wesley, 2001.
6. *P. Cabalar:* Existential Quantifiers in the Rule Body, Proc. 22nd Workshop on (Constraint) Logic Programming, WLP 2009.
7. *A. Cali, G. Gottlob, T. Lukasiewicz:* A General Datalog–Based Framework for Tractable Query Answering over Ontologies, Proc. International Conference on Principles of Database Systems, PODS 2009, pp. 77–86.
8. *S. Ceri, G. Gottlob, L. Tanca:* Logic Programing and Databases, Springer, 1990.
9. *W.F. Clocksin, C.S. Mellish:* Programming in PROLOG, 5th Edition, Springer, 2003.
10. *G. Köstler, W. Kießling, H. Thöne, U. Güntzer:* Fixpoint Iteration with Subsumption in Deductive Databases, Journal of Intelligent Information Systems, Volume 4, Number 2, Springer, 1995.
11. *J. Lobo, J. Minker, A. Rajasekar:* Foundations of Disjunctive Logic Programming, MIT Press, 1992.

12. *B. Marnette:* Generalized Schema–Mappings: From Termination To Tractability, Proc. International Conference on Principles of Database Systems, PODS 2009, pp. 13–22.
13. *J. Minker (Ed.):* Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1987.
14. *P. Puppe et al.:* D3. `http://d3web.informatik.uni-wuerzburg.de/`
15. *D. Seipel:* The DisLog Developers' Kit (DDK), `http://www1.informatik.uni-wuerzburg.de/databases/DisLog`
16. *D. Seipel:* Using Clausal Deductive Databases for Defining Semantics in Disjunctive Deductive Databases. Annals of Mathematics and Artificial Intelligence, vol. 33, Kluwer Academic Publishers, 2001, pp. 347-378.
17. *D. Seipel, M. Hopfner, B. Heumesser:* Analyzing and Visualizing Prolog Programs based on Xml Representations. Proc. International Workshop on Logic Programing Environments, WLPE 2003.
18. *D. Seipel, J. Baumeister:* Declarative Specification and Interpretation of Rule–Based Systems, Proc. 21st International Florida Artificial Intelligence Research Society Conference, FLAIRS 2008, AAAI Press, 2008.
19. *J. Wielemaker:* Swi–Prolog 5.0 Reference Manual and *J. Wielemaker, A. Anjewierden:* Programming in Xpce/Prolog, `http://www.swi-prolog.org/`