Extending Datalog To Express Functional Queries: A Language And Its Implementation

Stefano Basta, Sergio Flesca and Sergio Greco
DEIS,
Univ. della Calabria,
87036 Rende, Italy
{basta,flesca,greco}@si.deis.unical.it

Abstract

In this paper we present a "functional" logic language, i.e. a language whose answer is always defined and unique, i.e., it is either true or false in all models. The language is based on a disciplined usage of negation, which allows to achieve the desired level of expressiveness up to $\mathcal{BH}$. Although the semantics of the new language is partial, all atoms in the source program are defined and possibly undefined atoms are introduced in a rewriting phase to increase the expressive power. We present an algorithm for the evaluation of functional queries and we show that exponential time resolution is eventually enabled only for hard problems. Finally we present the architecture of a prototype of the language which is currently under development.

1 Introduction

Stable model semantics introduces a sort of non-determinism in the sense that programs may have more than one "intended" model [5]. *P-stable* (partial stable) model (also called *three-valued stable model*) represents a simple, natural extension of the notion of (total) stable model to the domain of partial interpretations. The problem with stable model semantics is that the expressive power can blow up without control, so that polynomial-time resolution is not any-more guaranteed [11]. A second problem is that the existence of multiple stable models for the same program contrasts with the traditional requirement that the canonical meaning of a (positive) logic program be based on a unique model.

The presence of multiple models requires, generally, to search for more than one stable model in answering queries. This situation can be avoided for those queries that are guaranteed to have all stable models sharing the same truth value for the query goal, i.e., possible and definite semantics coincide. We call such queries *deterministic*. Another desirable property for a query is *totality*, i.e., the query goal is never left undefined in the intended models. Queries that are both deterministic and total are called *functional*; functionality guarantees that querying the negated goal answers the complementary problem. Observe that recognizing whether a query is functional is undecidable [2]. However, as it often happens in computer science, it is possible to devise a language that enforces functionality.

In this paper we consider the extension of DATALOG, denoted $\texttt{DATALOG}^{\neg_s,c,p}$, obtained by adding the *choice* construct and a weak form of constraints called *preference rules* to $\texttt{DATALOG}^{\neg_s}$. $\texttt{DATALOG}^{\neg_s,c,p}$ keeps a declarative, intuitive semantics as well as an efficient implementation. In [7] it has been shown that $\texttt{DATALOG}^{\neg_s,c}$ (DATALOG with stratified negation and choice) has the same expressive power of $\texttt{DATALOG}^{\neg}$ (both capture the classes of queries in $\mathcal{NP}$ and co$\mathcal{NP}$ under the possible and certain version of total stable model semantics). Moreover, under the non-deterministic semantics, $\texttt{DATALOG}^{\neg_s,c}$ captures, exactly, the whole class $\mathcal{P}$ whereas $\texttt{DATALOG}^{\neg}$ is not able to capture this important class. In this paper we show that the introduction of

preference rules permits, under the functional version of least undefined semantics, consisting in the non deterministic selection of a least undefined model, captures the whole boolean hierarchy $\mathcal{BH}$.

Although the semantics of $\texttt{DATALOG}^{\neg_s,c,p}$ is based on partial stable models, all atoms appearing in the source programs are defined (i.e., they can be either true or false) and only the atoms added in the rewriting stage can be undefined. Thus, undefined atoms are only used to increase the expressive power of the language.

Furthermore, the paper presents a unifying algorithm that automatically adapts to the complexity of the problem at hand. The algorithm shows how to make stable model semantics amenable to effective implementations. In summary, the paper presents a simple framework for exploiting recent advances in model-theoretic semantics for non-monotonic DATALOG, without surrendering naturalness and efficiency of stratified negation [3]. Many classical problems such as unique solution or optimization problems have a complexity which is beyond $\mathcal{NP}$ and co$\mathcal{NP}$. Therefore, powerful languages are needed to express these kind of problems.

2 Functional queries

In this section we recall the definition of functional query [8] We assume the reader is familiar with the concepts of relational databases and of the DATALOG language [15] as well as of logic programming [10] and (partial) stable model semantics [5, 13]. We assume also familiarity with the basic notions of complexity classes [12] and of query language complexity evaluation (see, for instance, [1, 4, 16]).

A partial-stable ($\mathcal{PS}$) model M is: (1) *well founded* ($\mathcal{WS}$) if M is contained in every $\mathcal{PS}$ model of P; (2) *total stable* ($\mathcal{TS}$) if M is a total interpretation; (3) *maximal stable* ($\mathcal{MS}$) if there exists no $\mathcal{PS}$ model of P which is a proper superset of M; (4) *least-undefined stable* ($\mathcal{LS}$) if the set of its undefined atoms is minimal, i.e., there exists no $\mathcal{PS}$ model N of P such that $\overline{N}$ (the set of undefined atoms in N) is a proper subset of $\overline{M}$. In the following $\mathcal{XS}$ will denote a generic stable model semantics ($\mathcal{XS} \in \{\mathcal{WS}, \mathcal{PS}, \mathcal{TS}, \mathcal{MS}, \mathcal{LS}\}$).

A (boolean) query is a pair $\langle P, G\rangle$ where P is a program and G a ground atom (the goal).

Definition 1 A query $Q = \langle P, G\rangle$ is

1. $\mathcal{XS}$*-total* if for each database D, both (i) there exists an $\mathcal{XS}$-stable model of $P \cup D$ and (ii) for each $\mathcal{XS}$-stable model M of $P \cup D$, either G or $\neg G$ is in M — thus the goal G is never undefined;
2. $\mathcal{XS}$*-deterministic* if the set of database recognized under possible and certain semantics coincide — thus the truth value for the goal G is unique in all model;
3. $\mathcal{XS}$*-functional* if it is both $\mathcal{XS}$-total and $\mathcal{XS}$-deterministic. □

3 Choice e Preference Rules in Datalog

The *choice* construct is used to enforce functional constraints on rules of a logic program [14]. Thus, a goal of the form, choice((X),(Y)), in a rule r denotes that the set of all consequences derived from r must respect the FD $X \rightarrow Y$. In general, X can be a vector of variables—possibly an empty one denoted by "( )" —and Y is a vector of one or more variables.

As shown in [14] the formal semantics of the construct can be given in terms of stable model semantics as follows. A rule of the form

$$p(X, Y, W) \leftarrow q(X, Y, Z, W),\ \mathit{choice}((X), (Y)),\ \mathit{choice}((Y), (Z))$$

is rewritten as:

$$
\begin{array}{lll}
p(X,Y,W) & \leftarrow & q(X,Y,Z,W),\ chosen(X,Y,Z). \\
chosen(X,Y,Z) & \leftarrow & q(X,Y,Z,W),\ \neg diffchoice(X,Y,Z). \\
diffchoice(X,Y,Z) & \leftarrow & chosen(X,Y',Z'),\ Y \neq Y'. \\
diffchoice(X,Y,Z) & \leftarrow & chosen(X',Y,Z'),\ Z \neq Z'.
\end{array}
$$

Observe that the rewritten program is not stratified. Thus, we assume that the semantics of a choice program is given by the stable model semantics of the rewritten program. Let P be a DATALOG$^{\neg}$ program with choice constructs. If P is stratified modulo choice (i.e., by removing the choice predicates) the stable model is not in general unique but existence of at least one of them is guaranteed and can be computed in polynomial time [14, 6].

Let us now introduce the second construct of our language. A *preference* rule is of the form

$$should\_be(A).$$

where A is a literal and is called a *preference goal* or, more precisely, a *should_be* goal. A rule of the form $should\_be(A)$ is rewritten as

$$A' \leftarrow \neg A, \neg A'$$

where if A is a literal of forms $p(t_1,..,t_k)$ (resp. $\neg p(t_1,..,t_k)$), then $A' = p'(t_1,..,t_k)$ (resp. $\neg p'(t_1,..,t_k)$) and p' is a new predicate symbol. The predicate p appearing in the preference rule is called should_be predicate. Observe that the *should_be* predicate has second order syntax but its semantics is first-order. The rewritten program is not stratified and could not have total stable models. Possible undefinedness of A' implements the following intuitive meaning of the rule: one should search for an L-stable model which makes A true. The input database has the wished property when such a model exists or when no such model exists otherwise. Moreover, *should_be* predicates cannot be used as body goals or as head predicate of rules with not empty body.

Thus, the preference rules are used to make restrictions on the set of stable models. The language obtained by adding preference rules to DATALOG$^{\neg_s,c}$ will be denoted DATALOG$^{\neg_s,c,p}$.

4 A Functional Language

In this section we present an example of functional obtained as restriction of DATALOG$^{\neg_s,c,p}$, which captures every level in the boolean hierarchy.

Definition 2 A DATALOG$^{\neg_s,c,p}$ program P is said to be functional (denoted F-DATALOG$^{\neg_s,c,p}$) if

1. the preference rules are of the form $should\_be(p)$ — thus should_be predicates have arity zero and should_be goals are positive;
2. the choice rules define predicates not depending on should_be predicates;
3. the rules defining predicates depending on should_be predicates are not recursive;
4. P is *choice independent*, i.e., there are not two *should_be* predicates depending on the same choice predicate. □

Observe that (1) conditions 2, 3 and 4 are only introduced to simplify the computation of queries and they do not reduce the expressive power of the language, (2) groundness of preference rules guarantees that the $\mathcal{NP}$ oracle is called a constant number of time, and (3) choice independency guarantees that programs do not contain two should_be predicates depending on alternative choices of the same choice atom (4) functional programs can be partitioned into separated components and, each component can be computed independently from the others.

Proposition 1 *A* F-DATALOG$^{\neg_s,c,p}$ *query* $Q = \langle P, q\rangle$ *is functional if* q *depends on choice predicates only through should be predicates.* □

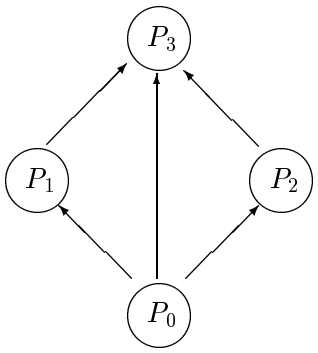

Fig. 1. Structure of $\texttt{F-DATALOG}^{\neg s,c,p}$ programs

The semantics of a program is given by the non-deterministic selection of one least undefined stable model. Under functional semantics, $\texttt{DATALOG}^{\neg s,c}$ programs permit to express the whole class of polynomial problems. $\texttt{F-DATALOG}^{\neg s,c,p}$ programs containing exactly one preference rule permit to express problems in $\mathcal{NP}$ or co$\mathcal{NP}$whereas $\texttt{F-DATALOG}^{\neg s,c,p}$ programs containing more than one preference rule permit to express combinations of problems in $\mathcal{NP}$ or co$\mathcal{NP}$, i.e., problems in the boolean hierarchy.

Theorem 1 $\texttt{F-DATALOG}^{\neg s,c,p}$*, under the functional version of least undefined stable model semantics, captures exactly the complexity class* $\mathcal{BH}$*.* □

Let P be a $\texttt{F-DATALOG}^{\neg s,c,p}$ program and let p and q be two should_be predicates depending both on a predicate s whose definition contains a choice rule. Then, we say that q is *more specific* than p for s if p depends on q. Moreover, we say that q is the *most specific* should_be predicate for s is there exists no should_be predicate p which is more specific than q for s.

As showed in Fig. 1, a $\texttt{F-DATALOG}^{\neg s,c,p}$ program P can be decomposed into 4 distinct groups of rules:

1. the group P_0 consists of the $\texttt{DATALOG}^{\neg s}$ rules defining predicates not depending on choice atoms;
2. the group P_1 consists of the $\texttt{DATALOG}^{\neg s,c}$ rules defining predicates depending on choice atoms and not used to define should_be predicates;
3. the group P_2 consists of the $\texttt{DATALOG}^{\neg s,c}$ rules defining predicates depending on choice atoms and used to define should_be predicates plus the set of preference rules;
4. the group P_3 contains the remaining rules, i.e., the (non recursive) $\texttt{DATALOG}^{\neg s}$ rules defining predicates depending on should_be predicates.

Observe also that the first, second and third groups can be further decomposed into different groups defining sets of mutually recursive predicates (components); such groups can be computed following the topological order defined by the dependencies among predicates. The next example should clarify the structure of $\texttt{F-DATALOG}^{\neg s,c,p}$ programs.

Example 1 *Unique hamiltonian path.* An undirected graph G has a hamiltonian path if there is a path visiting each node in G exactly once. Assume that the edges in the graph are stored as (oriented) arcs in order to reduce the number of tuples in the database. The sub-program *EDGE*, consisting of the following two rules, derives for each arc (x, y) two edges (x, y) and (y, x).

```
edge(X,Y) ← arc(X,Y).
edge(X,Y) ← arc(Y,X).
```

Consider the family of sub-program HP_i, consisting of the following rules plus the rules in *EDGE*.

$\mathtt{sp_i(root, Y)} \leftarrow \mathtt{node(Y)},\ \mathtt{choice((), (Y))}.$
$\mathtt{sp_i(X, Y)} \leftarrow \mathtt{sp_i(\_, X)},\ \mathtt{edge(X, Y)},\ \mathtt{sp_i(root, Z)},\ \mathtt{Z} \neq \mathtt{Y},$
$\mathtt{choice((X), (Y))},\ \mathtt{choice((Y), (X))}.$
$\mathtt{no\_hp_i} \leftarrow \mathtt{node(X)},\ \neg\mathtt{sp_i(\_, X)}.$
$\mathtt{hp_i} \leftarrow \neg\mathtt{no\_hp_i}.$

Consider now an instance of this program HP_1 plus the following rule

$\mathtt{should\_be(hp_1)}.$

The predicate `sp1` computes a simple path in G whereas the should_be goal is used to verify that all nodes are in the path. Thus, G has a hamiltonian path iff HP has a $\mathcal{LS}$ model satisfying `hp1`. Consider now the F-DATALOG$^{\neg_s,c,p}$ program $2HP$, consisting of the rules in $HP1$ plus two instance HP_2 and HP_3 of HP_i and the following rules

$\mathtt{two\_hp} \leftarrow \mathtt{hp_3},\ \mathtt{hp_2},\ \mathtt{sp_2(X, Y)},\ \mathtt{X} \neq \mathtt{root},\ \neg\mathtt{sp_3(X, Y)},\ \neg\mathtt{sp_3(Y, X)}.$

$\mathtt{should\_be(two\_hp)}.$

The predicates `sp1`, `sp2` and `sp3` select simple paths. The predicate `two_hp` verifies that the two simple paths are distinct and hamiltonian. Finally, consider the program *UHP* consisting of the rules in $2HP$ plus the rule

$\mathtt{unique\_hp} \leftarrow \mathtt{hp1},\ \neg\mathtt{two\_hp}.$

The graph G has a unique hamiltonian path iff *UHP* has a $\mathcal{LS}$ model satisfying `unique_hp`. Consider now the structure of the global program *UHP*. The first group of rules (P_0 in Fig. 1) consists of the rules in *EDGE*; the second group of rules (P_1 in Fig. 1) is empty since the choice rules are used to define should_be predicates; the third group (P_2 in Fig. 1) consists of the rules in HP and $2HP$ and the fourth group (P_3 in Fig. 1) consists of the rule r. Observe that the query $\langle UHP, unique\_hp\rangle$ is functional since the body of the rule defining *unique_hp* contains only should_be atoms. □

Definition 3 Let P be a F-DATALOG$^{\neg_s,c,p}$ program and let q a predicate symbol in P, the set of rules in P which is relevant for q is

$$REL(P,q) = \begin{cases} Rel(P,q) & \text{there is no should\_be predicate depending on } q \text{ or Rel(P,q) is a DATALOG}^{\neg_s} \text{ program; else} \\ Rel(P,s) & \text{where } s \text{ is the most specific should\_be predicate for } q \end{cases}$$

where $Rel(P,s)$ denotes the set of rules defining a predicate q such that s depends on q (directly or undirectly). □

Intuitively, $REL(P,q)$ denotes the set of rules in P which are necessary to answer queries having q as predicate symbol in the goal, i.e., queries of form $\langle P, q(t)\rangle$.

Example 2 Consider the program of Example 1. We have that *REL(UHP, sp1) = REL(UHP,hp1) = HP1*, *REL(UHP,sp2) = REL(UHP,two_hp) = 2HP* and *REL(UHP,edge)= EDGE*. □

Theorem 2 *Let P be a* F-DATALOG$^{\neg_s,c,p}$ *program and let q_1 and q_2 be two predicate symbols in P such that $REL(P,q_1) \cup REL(P,q_2) = P$. Then, an interpretation*[1] *M is a $\mathcal{LS}$ model for P iff there exist two $\mathcal{LS}$ models M_1 and M_2 respectively for $REL(P,q_1)$ and $REL(P,q_2)$ such that $M = M_1 \cup M_2$.* □

[1] Recall that a partial interpretation is a consistent subset of $B_P \cup \neg B_P$ where B_P is the Herbrand base and $\neg B_P = \{\neg A | A \in B_P\}$. An interpretation I is consistent if there are no two literals A and $\neg A$ in I.

```
var D (database) : set of facts;
    P, P_0, P_1, P_2, P_3 : Program;
    P_0, P_1, P_2, P_3 define a partition of P as shown in Fig.1;

function Top_Down_Evaluator(LG: List of literals): boolean;
var G: literal; LG_1: List of literals;
begin
    if LG = [] then return true;
    let LG = [G|LG_1] and let f be the predicate symbol of G;
    if G is a negative literal then begin
        if Def(f) ⊆ P_3 then answer:= Top-Down_Evaluator([¬G])
        else begin
            if G is not already computed then Bottom-Up_Interpreter(REL(P,f));
            if D contains a fact unifiable with ¬G then answer := false
                                                   else answer := true;
        end;
        if answer  then return Top-Down_Evaluator(LG_1)
                   else return false;
    end
    else begin    { G is positive }
        if Def(f) ⊆ P_3 then R_S := Def(f)
        else begin
            if G is not already computed then Bottom-Up_Interpreter(REL(P,f));
            R_S := {f(x⃗). such that f(x⃗) ∈ D};
        end;
        answer := false;
        while R_S ≠ ∅ and not answer do begin
            extract a rule A ← B_1, ..., B_n from R_S, n ≥ 0;
            if exist an mgu Θ_1 of A and G then
                answer :=Top-Down_Evaluator([B_1Θ_1, ..., B_nΘ_1|LG_1Θ_1]);
        end;
        return answer;
    end;
end;
```

Fig. 2. Mixed Top_down-Bottom_up algorithm.

Corollary 1 *Let $Q = \langle P, g(t)\rangle$ be a* F-DATALOG$^{\neg_s,c,p}$ *query. Then the query $Q' = \langle REL(P,g), g(t)\rangle$ is query equivalent to Q.* □

The above corollary means that to answer the query Q it is sufficient to consider only the program $REL(P,g)$. Note that for DATALOG$^{\neg}$ programs under both total and least undefined stable model semantics it is not possible to consider only the set of relevant rules $Rel(P,g)$.

5 Computing F-DATALOG$^{\neg_s,c,p}$ queries

The evaluation of a (boolean) F-DATALOG$^{\neg_s,c,p}$ query $Q = \langle P, g(t)\rangle$ is carried by means of a *mixed top_down-bottom_up* resolution algorithm, reported in Figure 2 and consisting of two main functions: the *Top_Down_Evaluator* function and the *Bottom_Up_Evaluator* procedure. We assume that the program P is partitioned into the four groups of rules P_0, P_1, P_2 and P_4 as described in Section 4. The set of rules in P having the same predicates symbol in the head, say f, is denoted $Def(f)$.

The computation of the query Q is carried out by calling the function *Top_Down_Evaluator* which receives the goal and returns the answer ("true" or "false"). The evaluation of a query

```
procedure Bottom-Up_Interpreter(P : Program)
var M, M1 : Set of facts; L, L1 : List of choices;
begin
    if P is a DATALOG^{¬s,c} program then M := choice_fixpoint(P, [], L)
    else begin
        let s be the most specific should_be predicate in P;
        P1 := REL(P, s);   P2 = P − P1;
        M := choice_fixpoint(P1, L, L1); L := L1;
        while s ∉ M and L ≠ [ ] do begin
            M1 := different_choice_fixpoint(P1, L, L1)
            M := M1; L := L1;
        end;
    end
    D := D ∪ M
    if P2 ≠ ∅ then BottomUp_Interpreter(P2)
end
```

Fig. 3. Bottom_up procedure.

$Q = \langle P, G\rangle$ starts by calling the procedure with input parameter $[G]$. Essentially, the function *Top_Down_Evaluator* extends the classical *SLD* resolution algorithm with memoing and to deal with stratified negation. Memoing is used to compute subprograms only once. The function selects the first goal G from the resolvent LG and, let f be the predicate symbol in G, if $Def(f)$ is recursive or contains choice rules or contains rules which are relevant for some should_be predicate, then the *Bottom_Up_Evaluator* procedure is called to compute the set of rules relevant for f (*REL(P,f)*). Thus, the global algorithm uses a mixed strategy and works like the classical "Query-Subquery" algorithm [17]. For the sake of simplicity, in the algorithm we have not considered comparison predicates.

The procedure *Bottom_Up_Evaluator* computes a set of facts M which are added to the current database D. In particular, it receives in input a subprogram P (the set of relevant rules selected by the *Top_Down_Evaluator*) and computes a $\mathcal{LS}$ model for P. If P contains preference rules then, let s be the most specific should_be predicate for P, the set of rules of P which are relevant for s, say P', are extracted from P and computed first. Next, the procedure is called with input parameter the set of remaining rules $P'' = P - P'$. The computation of P' is performed by calling the procedure *choice_fixpoint* computing a total stable model M for the program P' without preference rules, which is called repeatedly until the should_be goal s is satisfyed or all models are computed.

The function *choice_fixpoint* receives in input a program and the list of previous choices and computes a stable model and a new list of choice. The function *choice_fixpoint* can be found in [7] and is not reported here for the sake of brevity.

6 Architecture of the Prototype

The system prototype consists of two components: compiler and interpreter. The compiler analizes the source programs and define a partition of the rules in the first three groups into components which can be evaluated separately. The interpreter executes a query on a given database and answers the query submitted by the user.

6.1 Compiler.

The architecture of the compiler is showed in Fig. 4 and consists of two modules: the syntax/semantics analyzer and the rewriter.

The analyzer verifies the correctness of the source program and build the *dependency graph*

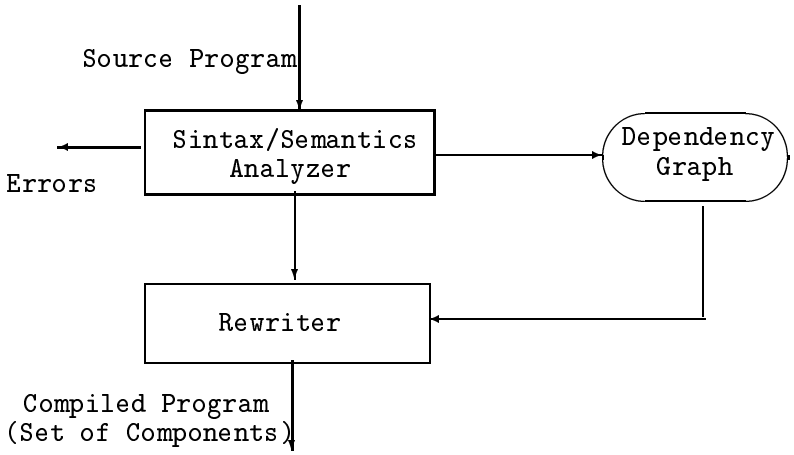

Fig. 4. Architecture of the module compiler

of the program which stores the dependencies among the predicates in the program. More specifically, it verifies that the program is a safe F-DATALOG$^{\neg_s,c,p}$.

The module *Rewriter* rewrites the input program to be efficiently evaluated by the interpreter. In particular it decomposes the input program into a set of components and for each component stores the associated group (recall that programs are decomposed into four groups of rules).

6.2 Interpreter.

The architecture of the interpreter is showed in Fig. 5. It is implemented as a Prolog meta-program and consists of the following modules:

- *Top-down Interpreter* which implements the *Top_Down_Evaluator* function of Figure 2, and
- *Bottom-up Interpreter* which implements the *Bottom_Up_Evaluator* procedure of Figure 3.

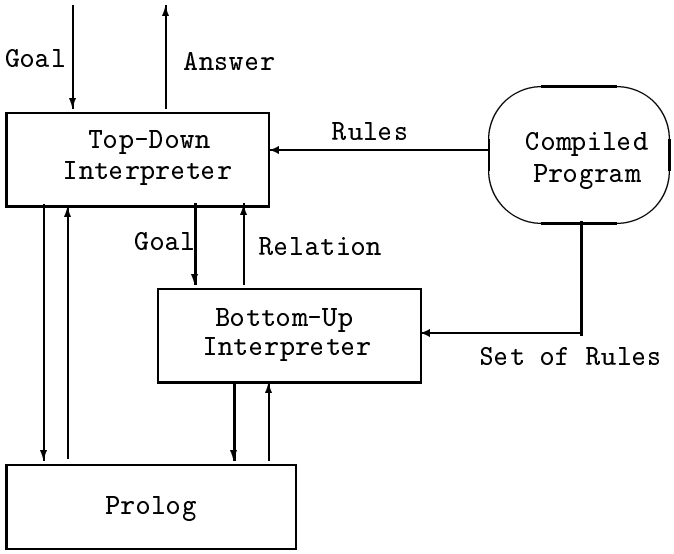

Fig. 5. Architecture of the module interpreter

The database as well as the set of tuples computed are stored Prolog tuples. A prototype of the system has been implemented by means of a meta-interpreter written in SWI-Prolog under the operating system Windows95.

References

[1] Abiteboul S., Hull R., Vianu V., *Foundations of Databases*, Addison-Wesley, 1994.

[2] Abiteboul S., Simon E., Vianu V., "Non-deterministic languages to express deterministic transformations", *Proc. ACM PODS Symp.*,1990, pp. 218–229.

[3] Apt K., Blair H. and A. Walker, Towards a theory of declarative knowledge, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming* Morgan Kauffman, USA, 1988, 89-142.

[4] Chandra A., Harel D., "Structure and Complexity of Relational Queries", *Journal of Computer and System Sciences 25*, 1, 1982, pp. 99–128.

[5] Gelfond M., Lifschitz V., "The Stable Model Semantics for Logic Programming", *Proc. 5th Int. Conf. on Logic Programming*, 1988, pp. 1070–1080.

[6] Giannotti F., Pedreschi D., Saccà D., Zaniolo C., "Non-Determinism in Deductive Databases", *Proc. 2nd DOOD Conf.* , 1991.

[7] Greco S., Saccà D., Zaniolo C., "DATALOG Queries with Stratified Negation and Choice: from $\mathcal{P}$ to $\mathcal{D}^p$", In *Proc. Fifth ICDT Conf.*, 1995.

[8] Greco S., Saccà D., "Possible-is-Certain, is desiderable and can be expressive", Annals of Mathematics and Artificial Intelligence , 1997.

[9] Greco S., "Choice and Weak Constraints in Datalog", AGP Conf., 1996.

[10] Lloyd J.W., *Foundations of Logic Programming*, Springer Verlag, Berlin, 1987.

[11] Marek W., Truszcynski M., "Autoepistemic Logic", *J. of the ACM 38*, 3, 1991, pp. 588–619.

[12] Papadimitriou C.H., *Computational Complexity*, Addison Wesley, 1994.

[13] Saccà D., "The Expressive Powers of Stable Models for Bound and Unbound DATALOG Queries", *Journal of Computer and System Science*, 1995.

[14] Saccà D., Zaniolo C., "Stable Models and Non-Determinism in Logic Programs with Negation", *Proc. ACM PODS Symp.*, 1990, pp. 205–218.

[15] Ullman J.D., *Principles of Database and Knowledge Base Systems*, Vol 1-2, Computer Science Press, 1989.

[16] Vardi M.Y., "The Complexity of Relational Query Languages", *Proc. ACM Symp. on Theory of Computing*, 1982, pp. 137–146.

[17] Vielle L., Recursive Query processing: The Power of Logic. *Theoretical Computer Science*. No. 69, 1989, pp. 1–53.