LOLA - A Logic Language for Deductive Databases and its Implementation

Burkhard Freitag **Heribert Schütz** **Günther Specht**

Technische Universität München, Institut für Informatik
Orleansstrasse 34, D-8000 München 80, Germany
E-Mail: freitag@lan.informatik.tu-muenchen.dbp.de

Abstract

The purpose of this paper is to give an overview of the *LOLA* logic programming system which serves as the kernel of a deductive database system. The *LOLA* language, the query evaluation strategy, the available query optimizations, the overall compiler architecture, and some implementational details and performance measurements are presented. The *LOLA*-system is fully implemented in Common Lisp and running on SUN UNIX Workstations.

1 Introduction

The logic language *LOLA* has been designed as a query language for a deductive database system. *LOLA* has a clear declarative semantics and integrates logic programming and relational query processing. Automated access to external relational databases and a link to the host language Lisp by built-in predicates are provided.
Instead of applying the top-down query evaluation scheme of resolution based logic programming systems [17] *LOLA* queries are evaluated in a set-oriented, bottom-up fashion starting from the program facts or base relations. The main evaluation scheme for recursive queries is the differential or *seminaive fixpoint iteration* as described in [8], [4], [6], [13]. A number of optimizations, among others the Magic Sets transformation [10], [5], [7] are available and can optionally be applied to the program. Different from comparable logic languages such as LDL-1 [9], [29] and NAIL! [26], [29], type declarations are required for function and predicate symbols. Furthermore, future *LOLA* versions will - like Prolog - be able to handle relations containing variables.
A query is translated into an expression of an extended relational algebra. Given a set of base relations, the resulting expression computes the set of answer tuples corresponding to the query. Base relations can reside in main memory or in any external relational database system accessible via SQL.
The *LOLA* system has as components a user interface accepting programs and queries, the compiler and optimizer and the run time system providing a main memory database, external database access and a link to the host language Lisp. The *LOLA* system is fully implemented in Commonlisp and running on SUN UNIX workstations.
The *LOLA* system has been designed and implemented by the deductive database research group of Prof. R. Bayer at the Technische Universität of Munich. The work has partly been funded by the "Deutsche Forschungsgemeinschaft" under contract "Ba 722/3-2: Effiziente Verfahren zur logischen Deduktion über Objektbanken".
Section 2 below describes the *LOLA* Language. In section 3 the *LOLA* semantics and query evaluation scheme are discussed. An overview of the *LOLA* system is given in section 4, in particular the *LOLA* compiler (4.2) and the optimizer (4.3). Finally, in section 5 some implementational details and performance data are listed. The complete *LOLA* syntax definition can be found in the appendix.

2 The *LOLA* Language

LOLA is a clausal logic programming language allowing to define predicates by rules and to declare the symbols used in a program. Negation, explicit existential quantification and functions are allowed. The programmer may use any type of recursion to define a predicate. *LOLA* provides a direct access to one or more external relational databases. The user only needs to know the name of the external database and the name and schema of the external relations. *LOLA* terms are first order terms without any restriction. Function symbols are - similar to PROLOG - not interpreted. However, a link to the host language is provided via built-in predicates allowing the user to directly implement her or his own interpretation of a predicate operating on terms of certain types.

2.1 Language Features

A sample *LOLA rule* is

```
only_child(X) :-
  person(X),$not($exists(Y, sibling(X,Y))).
```

Informally, the rule should be read as: X is the only child (of parents not named here) if X is a person and there is no Y being a sibling of X. As usual, we call the lefthand side of the rule the *head* and the righthand side the *body* of the rule. A *query* is a rule with empty head. Rules and queries are implicitly universally quantified with a scope of the entire rule or query. Due to the explicit

DATABASE SYSTEMS FOR ADVANCED APPLICATIONS '91
Ed. A. Makinouchi

existential quantifier we do not need to introduce complex rules for the implicit quantification of variables occurring in negations as in NAIL! [26] or LDL-1 [9].

The subgoals of a goal, i.e. a rule body or a query, may be either an atom, a database goal, a built-in goal, or a negated or existentially quantified goal, i.e. negation and existential quantification may be nested to any depth.

A *database goal* provides a means to specify an access to a - possibly external - database relation. The link to the specific database and relation is specified by the declaration of the corresponding predicate. By a *built-in goal* a call to a host language function implementing the semantics of the corresponding predicate is issued. Part of the declaration of a built-in predicate is the specification of a *binding request* defining for each attribute position whether it has to be *bound* (**$b**), i.e. a ground term is expected, or it may be *free* (**$f**), i.e. may contain variables at runtime. Let for instance the symbol **append_bbf** be declared as a built-in predicate requiring the first two attributes to be bound. The built-in goal **append_bbf([a,b],[c],L)** represents a call to the function **append_bbf** with actual parameters **[a,b,c]** and **[c,d]** which will return the relation

{**append_bbf([a,b],[c],[a,b,c])**}

A built-in predicate may be implemented as to evaluate its parameters. This way evaluable functions can be simulated in *LOLA*.

Note, that built-in predicates do not need to evaluate their parameters. In this case they operate on symbolic terms just as the directly implemented version of a normal predicate. A predicate is said to be *normal* if it neither is a built-in predicate, nor names a base relation.

```
$program(flight_connections).

%    *** Declaration Part ***
$predicate(flight,
           [airport,airport]).
$predicate(flight_route,
           [airport,airport,list(airport)]).
$predicate(route_to,[airport,list(airport)]).
$predicate(start_from,[airport]).
$predicate(direct_flight,[airport,airport]).
$built-in(member_bb,
          [A_Type,list(A_Type)],[$b,$b]).
$relation(timetable,
          [airport,airport,time,time,code],
          flight_db).
$database(flight_db,$transbase).

%    *** Definition Part ***
flight(X,Y) :-
    direct_flight(X,Y).
flight(X,Y) :-
    flight(X, Z), direct_flight(Z,Y).
route_to(Y,[Y,X]) :-
    start_from(X), direct_flight(X,Y).
route_to(Y,[Y|R]) :-
    route_to(Z,R), direct_flight(Z,Y),
    $not(member_bb(Y,R)).
direct_flight(X,Y) :-
    timetable(X,Y,T1,T2,N).
```

Figure 1: Sample *LOLA* program unit

```
$query(route_to).

%    *** Query Part ***
 :- route_to(tokyo, Route).

%    *** Declaration Part ***
$function(tokyo,[],airport).
$function(munich,[],airport).
$function(frankfurt,[],airport).

%    *** Definition Part ***
start_from(munich).
start_from(frankfurt).
```

Figure 2: Sample *LOLA* query unit

The programmer organizes her or his *LOLA* rules as a collection of *units*. We distinguish between a *LOLA program unit* and a *LOLA query unit*. A *LOLA* program unit consists of a number of *declarations* and *definitions*. For every constant, function, and predicate symbol a type declaration has to be specified (see section 2.2 below). The definition of a predicate is a set of rules as usual. *Figure 1* shows a sample *LOLA* program unit.

A *LOLA* query unit consists of a query followed by optional declarations and definitions. The latter two allow the user to declare constants introduced by the query as well as to define the shape of the answer relation by giving an "on the fly" definition of the query predicate which is independent of the underlying fixed *LOLA* program. The most important application of this feature, however, is to supply the actual definition of a predicate at query time. This can be used in order to keep the link to a database relation separate from the program or to specify an actual set of facts serving as the starting set of facts for a *LOLA* program which has been transformed by the Magic Sets transformation (see section 4.3). A sample *LOLA* query unit is shown in *Figure 2*.

Of course, queries can also be asked in an ad-hoc fashion during a *LOLA* session. For this type of queries type correctness is tacitly assumed and no declarations are required.

The complete *LOLA* syntax definition can be found in the appendix.

2.2 Declarations

Type declarations for normal function and predicate symbols in *LOLA* serve merely as a means to check program correctness at compile time. They can be viewed as a separate program specifying certain restrictions imposed on the rules in the definition part of a *LOLA* program. As far as this part of the *LOLA* type concept is concerned there are only minor differences to the concept of [19].

The *LOLA* type concept essentially provides a means to group the terms into different classes. *LOLA* types are *disjoint* and there are no union types or type hierarchies. For variables an attempt is made to infer a unique type at type checking time. They cannot be declared. A symbol can only be declared once. Predicate symbols and function symbols are of fixed arity. The sample program of *Figure 1* contains type declarations.

The declarations for built-in predicates, relations and databases have an additional meaning because they provide the link between the symbols and their operational semantics.

The *LOLA* syntax allows to deal with constructed lists similar to Prolog's lists. The list constructs `[X,Y]` and `[X|L]` are simply shorthand for the functional terms `cons(X,cons(Y,nil))` and `cons(X,L)` respectively. The symbols `nil` and `cons` are implicitly declared using the polymorphic type `list` as shown below.

```
$function(nil,[],list(Any_Type)).
$function(cons,[Any_Type,list(Any_Type)],
                list(Any_Type)).
```

Type checking is purely static and performed as a preprocessing step. A *LOLA* program unit is checked when fed to the compiler. At query processing time the type checking is repeated for the program augmented by the query unit.
The type correctness of a *LOLA* program is essentially described by the following rules

- The same unique type must be inferred for each occurrence of a variable in a clause.
- The result type inferred for a function symbol of a functional term must coincide with the type inferred for the argument position being the occurence of the term.

Type correctness is checked by simultaneous unification of the types inferred for a subterm using all possible inference paths.
Consider for example the correctly typed rule taken from the sample program of *Figure 1.*

```
route_to(Y,cons(Y,R)) :-
   route_to(Z,R),direct_flight(Z,Y),
   $not(member_bb(Y,R)).
```

The types `list(Any_Type)`, `list(airport)`, and `list(A_Type)` are attached to the variable `R` through the declarations of `cons`, `route_to`, and `member_bb` respectively. Therefore both `Any_Type` as well as `A_Type` must be bound to the type `airport`. This agrees with the type bindings inferred for `Y`. Thus the type `list(airport)` is inferred for the term `cons(Y,R)` as requested by the declaration of predicate `route_to`.
The type checker is completely written in *LOLA*. For more details and a description of the type checking algorithm see [23].

2.3 The Intermediate Language *LOLitA*

During the preprocessing phase of compilation (see section 4.2). a goal with nested negations and/or existential quantification is broken into a simpler goal together with a set of rules having as subgoals only atoms and negated atoms. We will refer to the simpler intermediate language allowing only unnested subgoals as *LOLitA* .
LOLitA programs and queries have to satisfy a number of constraints in order to be evaluable.

- *Stratifiability* with respect to negation
- *Range Restriction*, i.e. for every rule each variable of the rule head must occur in at least one unnegated normal or database subgoal or in a term at a free position in an unnegated built-in subgoal of the rule body.
- *Safety*, i.e. for every rule and query each variable of a negated subgoal not bound by existential quantification and each variable occuring in a term at a bound position of a built-in subgoal must occur in at least one unnegated normal or database subgoal.
- *Ground Database Relations*

Note that the safety constraint currently disallows the chaining of built-in predicates. This restriction can be relaxed as shown in [29, pp. 805-817], and a future version of the *LOLA* system will be able to deal with chained built-in subgoals.
The propagation of bindings introduced by a query will in many cases instantiate the variables of unrestricted rules making the modified program range restricted. The Magic Set transformation can as well be used to transform e.g. the rule `equal(X,X).` into a range restricted rule. Nevertheless are we currently investigating as to which extent the range restriction constraint can be generally relaxed by applying a more powerful evaluation scheme able to deal with general substitution sets rather than with ground tuple sets [18]. Consequently, the enhanced system will generate nonground answers[1].

2.4 Sample *LOLA* Programs

The sample *LOLA* program `flight_connections` with negation, built-in predicates, functions and recursion is shown in *Figure 1.* By the first five declarations normal predicates defined by *LOLA* rules are declared. `member_bb` is a built-in predicate having a binding request of "bound" for both attributes. The predicate `timetable` is declared to be a the name of a base relation stored in the external database `flight_db` which is maintained by the relational DBMS TransBase [27]. A call to the predicate `timetable` will automatically be translated into a SQL query and an appropriate call to the *LOLA*-SQL-Interface.
The `flight` predicate is defined as the transitive closure of `direct_flight`. Note, that the leftrecursive definition has been chosen. Predicate `route_to` gives a more detailed description of flight connections by the list of airports visited on the way from some start airport to `Y`. The list is constructed in reverse order for the sake of simplicity. The `start_from` predicate serves as an input and specifies the start airports. It is not defined in the program `flight_connections`.
Figure 2 shows the *LOLA* query unit `route_to` asking which flight route leads to Tokyo. The user has supplied a definition of the predicate `start_from` specifying `munich` and `frankfurt` as the start airports. The constants `tokyo`, `munich` and `frankfurt` have been declared in order to preserve type correctness.

3 Semantics

The semantics of *LOLA* is fully declarative. In particular, it does *not* depend on the subgoal order within a rule or query. The overall basis of the mathematical semantics of logic programs is described in [17]. The semantics of *LOLA* programs without negations coincides with the semantics of definite programs defined in [17] whereas the semantics of normal *LOLA* programs relies on the minimal model semantics defined by iterated fixpoint computation for a partitioning sequence of disjoint rule sets as described

[1]The acronyme *LOLA* spells "*L*ogic *L*anguage allowing *O*pen *A*nswers"

in [2][2].

The $LOLA$ semantics is defined through the semantics of the simpler language $LOLitA$ (see section 2.3). This point of view is reflected by the $LOLA$ compilation which as a first step generates the $LOLitA$ query and program corresponding to a given $LOLA$ query and program.

3.1 Notation

Our notation is partly adopted from [2]. Let $\mathcal{P}$ be a $LOLitA$ program. The *definition* of a predicate symbol p in $\mathcal{P}$ is the set of all defining rules of p. A rule r is a *defining rule* of p iff p is the head predicate of r. p is *defined* in the program $\mathcal{P}$ iff there is a defining rule $r \in \mathcal{P}$ of p.

A *relation* is a set of ground atoms having the same predicate symbol p. We often denote the relation itself by p. The relations named by predicate symbols that have been declared to be relation names are called the *base relations* of $\mathcal{P}$.

We say that a predicate symbol p *refers positively (negatively) to* a predicate symbol q iff q is the predicate symbol of a nonnegated (negated) atom in the body of a defining rule of p. The meta-relation *depends on*, denoted by $\succ$, is defined to be the transitive closure of the meta-relation *refers to*. A predicate symbol p is *recursive* iff we have $p \succ p$ otherwise p is *nonrecursive*. p and q are *mutually recursive* iff $p \succ q$ and $q \succ p$.

An equivalence relation $\asymp$ on the set of predicate symbols of $\mathcal{P}$ is defined by $p \asymp q$ iff $q \equiv p$ or p and q are mutually recursive. The equivalence classes are called the *predicate clusters* of $\mathcal{P}$. The collection of definitions of the predicates in a predicate cluster forms a *rule cluster* of $\mathcal{P}$. It should be clear that each rule cluster corresponds to a uniquely determined predicate cluster and vice versa. The $\succ$ relation can be defined on the set of rule clusters in a natural way.

The programs $\mathcal{P}_1, \ldots, \mathcal{P}_m$ form a *stratification* of $\mathcal{P}$ iff $\mathcal{P} = \mathcal{P}_1 \oplus \ldots \oplus \mathcal{P}_m$, where $\oplus$ denotes disjoint union, the definition of each predicate symbol q occurring positively in a rule $r \in \mathcal{P}_j$ is a subset of $\bigcup_{k=1}^{j} \mathcal{P}_k$, and the definition of each predicate symbol q occurring negatively in a rule $r \in \mathcal{P}_j$ is a subset of $\bigcup_{k=1}^{j-1} \mathcal{P}_k$. $\mathcal{P}$ is called *stratifiable* iff there is a stratification of $\mathcal{P}$. It is easy to show that the rule clusters of a stratifiable $LOLitA$ program $\mathcal{P}$ form a stratification of $\mathcal{P}$ (cf. [2]).

3.2 Minimal Model Semantics

We assume throughout this section that the language $L_\mathcal{P}$ defined by a $LOLitA$ program $\mathcal{P}$ is determined by the symbols of $\mathcal{P}$, the symbols occuring in the base relations of $\mathcal{P}$, and the symbols occuring in the relations generated by the built-in predicates of $\mathcal{P}$. The *Herbrand basis* of $\mathcal{P}$, denoted by $\mathcal{B}_\mathcal{P}$, is defined as usual using the language $L_\mathcal{P}$. An *interpretation* of $\mathcal{P}$ is a subset of the Herbrand basis.

The *immediate consequence operator* $T_\mathcal{P}$ of the program $\mathcal{P}$ maps the set of Herbrand interpretations onto itself. The usual definition is extended as follows in order to cover built-in predicates: We treat a built-in predicate p as a special base relation, i.e we assume that the semantics of a l-ary built-in predicate p is a priori known and given by the set of ground atoms

$$\{p(t_1, \ldots, t_l) \in \mathcal{B}_\mathcal{P} \mid p(t_1, \ldots, t_l) \text{ is true }\}$$

The above set is the *extension table* $\mathcal{E}_p$ of p.

In [2] it is shown that for a stratifiable program $\mathcal{P}$ a uniquely determined minimal Herbrand model of $\mathcal{P}$ can be computed by iterated fixpoint computation of the immediate consequence operators defined by the rule clusters of $\mathcal{P}$ starting from the base relations of $\mathcal{P}$. More formally, let $\mathcal{P}$ be a $LOLitA$ program and let $\mathcal{P}_1, \ldots, \mathcal{P}_m$ be a stratification of $\mathcal{P}$. Let $b_1, \ldots, b_l$ be the base relations of $\mathcal{P}$, and let $q_1, \ldots, q_r$ be the built-in predicates of $\mathcal{P}$. The sequence

$$M_0 := \bigcup_{k=1}^{l} b_k \cup \bigcup_{s=1}^{r} \mathcal{E}_{q_s}$$

$$M_j := T_{\mathcal{P}}^{\uparrow\omega}(M_{j-1}) \text{ for } 1 \leq j \leq m$$

defines a minimal Herbrand model M_m of $\mathcal{P}$. Furthermore, the model is independent of the particular stratification of $\mathcal{P}$. Let $\mathcal{P}$ be a stratifiable $LOLitA$ program. We define the *standard minimal Herbrand model* $M_\mathcal{P}$ of $\mathcal{P}$ by the model sequence which is determined by the rule clusters of $\mathcal{P}$. The standard minimal model coincides with the pefect model as defined in [20].

An *answer* to a query is a (possibly empty) set of substitutions for the variables of the query. From a positive atomic query `:-` $p(t_1, \ldots, t_k)$. and an answer substitution set Σ a corresponding *answer relation* can be computed by applying every $\sigma \in \Sigma$ to the query atom. Consider for example the $LOLA$ program of *Figure 1* and assume that

```
timetable =
   { timetable(munich,frankfurt,...),
     timetable(frankfurt,tokyo,...) }
```

An answer to the query `:- flight(U,V).` is the substitution set

```
{ {U/munich, V/frankfurt},
  {U/frankfurt, V/tokyo},
  {U/munich, V/tokyo} }
```

The corresponding answer relation is

```
{ flight(munich,frankfurt),
  flight(frankfurt,tokyo),
  flight(munich, tokyo) }
```

An answer to a negative atomic query `:- $not` $p(t_1, \ldots, t_k)$. is only defined for ground terms $t_1, \ldots, t_k$ due to the safety constraint (see section 2.3). It can either be the empty substitution set (empty relation) or the substitution set containing only the identity substitution (the relation containing only the atom `true`).

An answer Σ to a query `:-` $G_1, \ldots, G_n$. for a program $\mathcal{P}$ is a *correct answer* iff for every $\sigma \in \Sigma$ the instantiated goal $G_1\sigma, \ldots, G_n\sigma$ is true in the standard minimal model $M_\mathcal{P}$.

[2] The minimal model semantics of [2] differs from Clark's semantics of normal programs as defined in [17]. For more details see [2]

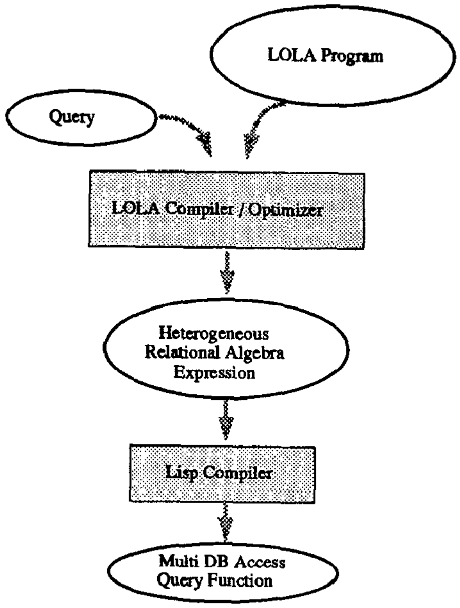

Figure 3: Top-Down Query Compilation

3.3 Operational Semantics

The results listed in the preceding section are the theoretical basis for an operational semantics of *LOLitA* . Essentially, for each rule cluster $\mathcal{P}_j$ a relational expression is constructed which implements the immediate consequence operator $T_{\mathcal{P}_j}$ of $\mathcal{P}_j$. The operational semantics of *LOLA* does therefore not differ from the minimal model semantics of section 3.2. Fixpoint iteration and the relational operations *union, project, select, join, product* working on sets of term tuples are the basic requisites for the definition of the operational semantics of *LOLitA* .

On a conceptual level a distinction is made between relations and substitution sets. However, suitably designed relational operations can be applied to both equally well. To clarify this point let Σ be a substitution set for the variables $x_1, \ldots, x_n$. Then Σ can be represented as the pair $((x_1, \ldots, x_n), R_\Sigma)$ where R_Σ is the set of of term tuples $(t_1, \ldots, t_n)$ such that $\{x_1/t_1, \ldots, x_n/t_n\} \in \Sigma$. The empty and the identity substitution sets are represented by $((), \emptyset)$ and $((), \{()\})$ respectively. Similarly, a n-ary relation R, i.e. a set of atoms $p(t_1, \ldots, t_k)$, can be represented as the pair $(p, Tuples_R)$ where $Tuples_R$ is the set of n-ary term tuples $(t_1, \ldots, t_n)$ such that $p(t_1, \ldots, t_n) \in R$.

For each rule cluster $\mathcal{P}_j$ of a program $\mathcal{P}$ a relational expression $\mathcal{E}(\mathcal{P}_j)$ is inductively defined as follows. Details can be found in [11] and are omitted here for the sake of brevity.

- Predicates defined in lower layers $\mathcal{P}_l$, $l < j$, and predicates naming base relations are taken as relational parameters.
- Recursive predicates which are members of the corresponding predicate cluster are taken as relational parameters. We will call them *local parameters.*
- A normal or a database subgoal $G \equiv p(t_1, \ldots, t_k)$ defines a *select* operation on the relation p. The result is the set of all substitutions σ for the variables of G such that there is an atom $A \in p$ and a substitution τ such that $A\tau \equiv G\sigma$[3].
- Let G be a built-in subgoal $p(t_1, \ldots, t_k)$. Assume without loss of generality that p requires its first $l \leq k$ attributes to be bound. Then, by the safety constraint (section 2.3), the terms $t_1, \ldots, t_l$ are ground[4]. Let $f_p[par_1, \ldots, par_l]$ denote the host language program implementing p. Then G defines an expression having as result the set of all substitutions σ for the variables of G such that $p(t_1, \ldots, t_k)\sigma \in f_p[t_1, \ldots, t_l]$.
- Let G be a negated subgoal `$not` $p(t_1, \ldots, t_k)$. Then, by the safety constraint (section 2.3), G is ground[5]. Let $\mathcal{E}$ be the expression defined by the positive goal $p(t_1, \ldots, t_k)$. G defines an expression having as result the empty substitution set if $\mathcal{E}$ returns the identity subsitution set and vice versa.
- A goal $G_1, \ldots, G_n$ defines a *join*, an *antijoin*[6], an *input join*[7], or a *product* operation on the substitution sets for the G_i computing a substitution set for the variables of $G_1, \ldots, G_n$.
- A rule $p(t_1, \ldots, t_k)$., i.e. a fact, defines the relation $\{p(t_1, \ldots, t_k)\}$.
- A rule $p(t_1, \ldots, t_k)$:- $G_1, \ldots, G_n$. defines a *project* operation on the substitution set Σ defined by the rule body. The result is the relation computed by applying every single substitution $\sigma \in \Sigma$ to the atom $p(t_1, \ldots, t_k)$.
- A nonempty set of rules defines a *union* operation on the set of the relations defined by the single rules. The empty set of rules defines the empty tuple set.

A rule cluster $\mathcal{P}_j$ defining the (mutually) recursive predicates $p_1, \ldots, p_m$, $m \geq 1$ requires a *fixpoint iteration* computing relations for the local parameters $p_1, \ldots, p_m$ and taking the nonlocal parameters as fixed input. The iterative relational expression thus defined does no longer have local but may still have nonlocal relational parameters. If a rule cluster defines a single nonrecusive predicate no fixpoint iteration is needed.

[3]Note, that for a ground relation p the result is simply the set of substitutions σ such that $G\sigma \in p$.

[4]In case of a nonground term at a bound position either G is a subgoal of a goal $G_1, \ldots, G_n$ with a positive normal or database subgoal G_i containing the variables of the term or a syntax error is detected. In the former case an input join expression is generated.

[5]If variables occur in G either G is a subgoal of a goal $G_1, \ldots, G_n$ with a positive normal or database subgoals G_i containing the variables of G or a syntax error is detected. In the former case an antijoin expression is generated.

[6]An *antijoin* operation can be understood as a generalized set difference.

[7]An *input join* operation essentially maps the procedure defining a built-in predicate over the set of tuples serving as input and subsequently joins the result and the input tuple set

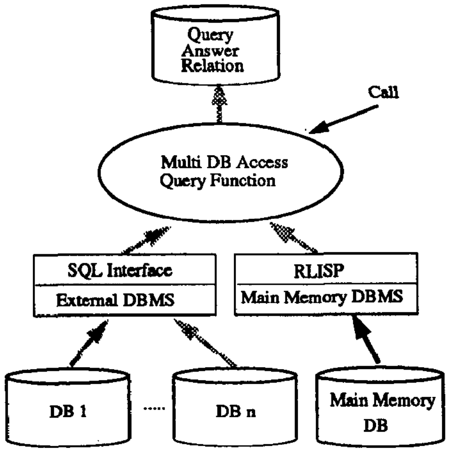

Figure 4: Bottom-Up Query Evaluation

A relational expression computing an answer for a query $:-G_1, \ldots, G_n.$ is defined by induction over the $\succ$-relation between rule clusters. We proceed top-down starting from the query. The query itself is treated like a rule body. For every subgoal $G_i \equiv q(t_1, \ldots, t_k)$ the rule cluster $\mathcal{P}_q$ defining q is selected and the corresponding relational expression $\mathcal{E}(\mathcal{P}_q)$ is constructed. For every nonlocal parameter r of $\mathcal{E}(\mathcal{P}_q)$ the relational expression $\mathcal{E}(\mathcal{P}_r)$ of the rule cluster defining r is generated. A predicate not defined in $\mathcal{P}$ and neither being declared as a relation name nor as a built-in predicate is considered to be defined by the empty rule cluster. This procedure is repeated until no nonlocal parameter is left. The translation procedure terminates because the $\succ$-relation between rule clusters is acyclic. Note, that the answer relation computed by the above defined operator is always ground if the program and query satisfy the constraints of section 2.3.

Under the general assumption that the computed tuple sets are finite[8] it can be shown that the relational expression defined for an atomic query computes the interpretation of the query predicate in the standard minimal model.

The *LOLA* compiler (see section 4.2) takes a program and a query as input and essentially generates a relational expression as described above. The fixpoint iteration scheme used is differential or seminaive fixpoint iteration [8], [4], [6] which has been shown to be complete and correct in [13] The compiler also generates code for the external database access and calls to procedures implementing the built-in predicates.

A schematic view of the query translation and evaluation is shown in *Figures 3* and *4*.

[8]By the above construction only finite subsets of the extension tables of built-in predicates are computed

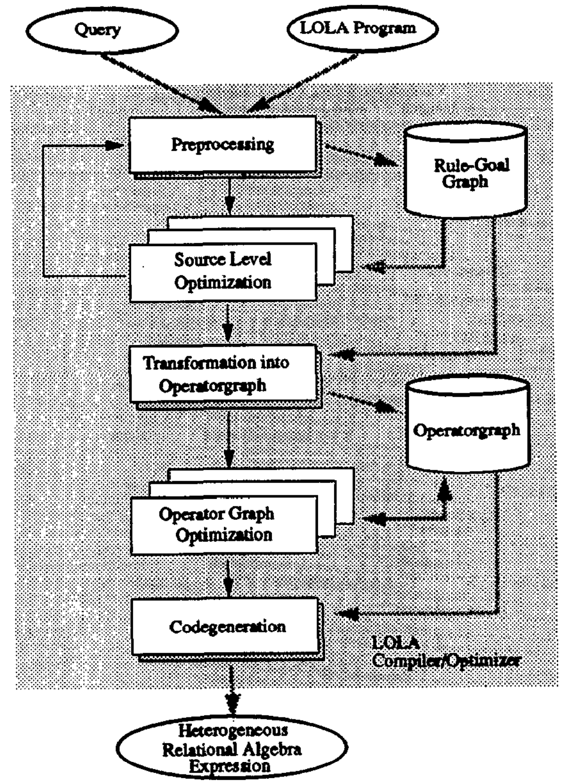

Figure 5: Overview of the *LOLA* compiler

4 The *LOLA* system

The *LOLA* system consists of an *user interface*, the *compiler*, the *optimizer*, and the *run time system*.

4.1 User Interface

The user interface takes a collection of *LOLA* program units as an input and initiates the necessary preprocessing steps. The system is then ready to accept a *LOLA* query unit or an ad-hoc query. The latter queries are automatically converted into query units. For each query optional optimizations can be selected by the user. After query compilation the answer is shown to the user or can be stored as a relation.

It is assumed that the user is interested in a relation, i.e. a set of facts, as the result of a query. For atomic queries a set of *LOLA* facts is shown as the answer, for instance

```
flight(munich,frankfurt).
flight(frankfurt,tokyo).
flight(munich, tokyo).
```

Nonatomic queries are converted into an atomic query having a new predicate and as attributes all the terms of the original query together with a rule defining the new predicate through the query subgoals.

4.2 The *LOLA* Compiler

The *LOLA* compiler takes a *LOLA* program unit and a query unit as input and generates a relational expression as described in section 3. The *LOLA* compiler has a modular architecture allowing to implement optimization algorithms or different code generators as "plug in" modules. Figure 5 shows an overview of the compilation process. The compiler makes use of precompiled and preoptimized queries thus supporting incremental compilation and modular program developement.

During the *preprocessing phase* a *LOLA* unit is transformed into a *LOLitA* unit by unnesting negations and eliminating existential quantification. The resulting *LOLitA* program is internally represented as a set of relations from which a *rule-goal graph* can easily be generated. The rule-goal graph[9] consists of rule nodes, goal nodes, and edges pointing from a rule to each of its body subgoals and from a goal to each of its defining rules. Every module performing a *source level optimization* actually transforms the rule-goal graph of the *LOLitA* program.

The rule-goal graph is subsequently transformed into an *operator graph* which represents an expression of a very high level extended relational algebra. A number of *optimizations* such as the introduction of indexes or the symbolic differentiation required for the differential fixpoint computation can better be performed on the operator graph as a common data structure.

During the *code generation phase* the heterogeneous relational expression is generated, i.e. relational expressions with embeded calls to the main memory database,the external database interface or a host language procedure (See section 4.4). The codegeneration can easily be adapted to different run-time systems.

4.3 The *LOLA* Optimizer

A number of optimization modules are available and can optionally be applied to a *LOLA* program. There are mainly two groups, the first performing source level optimizations and the second acting on the intermediate code represented by the operator graph.

4.3.1 Source Level Optimization

The available source level optimizations are actually designed as transformations of the rule-goal graph. They are currently invoked by the user rather than by strategy rules (cf. [26], [29, p. 992]). Future versions of the *LOLA* system will incorporate a metalevel strategy module governing the optimization.

Bindings introduced by the query may be propagated through nonrecursive rules down to the base relations. In case of recursive rules care has to be taken in order to leave the instantiated program semantically correct. An algorithm has been developed which generates a sufficient condition for the correctness of *selection propagation* [14]. The algorithm is applicable to any syntactically correct *LOLA* program even in the presence of functions or mutual recursion. The selection optimization transforms a given program into a specialized program. In the current version the *selection optimization* is always invoked.

The *Magic Sets transformation* [10], [5], [7] can be applied to any syntactically correct *LOLA* program and is *not* limited to DATALOG. The transformation also recognizes nonground facts and unrestricted rules and tries to convert them into range restricted rules. After application of the Magic Sets transformation a formerly stratifiable program need no longer be stratifiable. The BPR-labelling algorithm of [5] has been generalized in order to cover any possible removable violation of stratifiability [3].

The *optimization of projections* as described in [21] has been combined with a special selection propagation and is available for arbitrary *LOLA* programs [14]. The projection optimization often makes an early termination of the fixpoint iteration possible.

4.3.2 Operator Graph Optimization

Common subexpressions are detected on the basis of subgoals. Two subgoals define a common subexpression (CSUB) if they are variants or one is an instantiation of the other. CSUBs are translated into function objects knowing about their evaluating expression and the number of goal nodes referencing them. They are controlled by reference counts and free memory as soon as there are no open references left. Note, that particularly the nonlocal parameters of iterative relational expressions (see section 3.3) are subject to common subexpression optimization.

Common subexpression optimization (see above) makes it profitable to generate and maintain *indexes* even to support simple selection operations. The *index optimization* is always applied.

Database goals specify the access to one single external relation. As long as no fixpoint iteration or complex attribute terms are involved it is possible to combine database goals and generate more complex *SQL queries* according to the rules of the *LOLA* program thus putting a bigger part of the computational load onto the external DBMS. SQL queries can optionally be combined.

The *differential fixpoint iteration* scheme requires a modified relational expression which can be generated by symbolic differentiation of the original expression. This is done at the operator graph level. This optimization is always applied.

4.4 The *LOLA* Run Time System

Lisp has been chosen as the host language of the *LOLA* system. The *LOLA* run time system consists of the following components

- the Lisp system
- the extendend relational algebra *R–LISP*
- a main memory database
- a SQL interface to external relational database management systems

[9]The *LOLA* rule-goal graph is comparable to the Predicate Connection Graph of LDL-1 [9], [29] rather than to the rule-goal graph of NAIL! [26], [29]

Optimization	Compilation LOLA (sec)	Compilation Lisp (sec)	Evaluation (sec)
1413 Tuples in Base Relation `timetable` Cyclic Data			
Query: `:- flight(munich,Y).` No. Tuples: 101			
none	0.5	1.5	75.0
Selection	0.5	2.0	0.2
Magic Sets	2.0	6.0	1.3
Supp Magic Sets	2.0	6.0	1.0
Query: `:- flight(X,tokyo).` No. Tuples: 110			
none	0.5	1.5	76.0
Selection	not applicable		
Magic Sets	2.0	7.0	73.0
Supp Magic Sets	2.0	7.0	70.0
Query: `:- flight(munich,tokyo).` No. Tuples: 1			
none	0.5	1.5	77.0
Selection	0.5	2.0	0.3
Magic Sets	2.0	6.0	2.0
Supp Magic Sets	3.0	16.0	3.2

Table 1: Performance Data

R-LISP [22] is a Lisp macro package implementing the usual relational operations able to process relations having complex structured attribute values. In addition, *R-LISP* provides a general purpose differential fixpoint iteration operator and a number of operations for index creation and maintenance. *R-LISP* is the target code of the *LOLA* compiler.
The main memory database is currently just a set of *R-LISP* relations. A catalogue management will be integrated in future *LOLA* versions. Built-in predicates are implemented in Lisp.
SQL queries are issued by the Lisp system via the SQL interface [16] which makes use of the application interface usually provided by a relational database management system (DBMS). Relations can be transferred back and forth between the Lisp system and the external DBMS.
The heterogeneous relational expression as defined in section 4.2 is compiled into the *multi database access query function* by the Lisp compiler (see *Figure 3*). When a call to the query function is issued by the Lisp system the answer is computed starting from base relations and functions implementing built-in predicates (see *Figure 4*).

5 Implementation

The preprocessing modules are fully written in *LOLA* [24], [12]. Some optimization modules are also written in *LOLA* but in the current version the compiler kernel is directly implemented in Lisp. Since all the data structures are relationally represented the compiler modules can in general be specified as a *LOLA* program as long as built-in predicates are available for the basic operations on the `list` datatype. Thus a bootstrapping of the *LOLA* compiler seems to be possible.
Allegro Common Lisp [1] is the implementation language of the *LOLA* system. Thus, the evaluable code of the *LOLA* system, either generated by the *LOLA* compiler or directly coded, is Allegro Common Lisp. The system is currently running on SUN 3 and SUN 4 workstations under UNIX. The standard external database management system is TransBase [27] which is coupled to the *LOLA* system [16] via the foreign function interface of Allegro Common Lisp and the TBX interface of the TransBase DBMS [28]. Any other relational DBMS can be coupled to the *LOLA* system as long as the query language is SQL and it provides an application programming interface. Experiments with other query languages have been made but a fully operational interface is only available for SQL.
Even complex queries are translated into relational algebra or SQL code in a few seconds. The main bottleneck is the Lisp compiler. Experiments have shown that the automated partitioning of the relational algebra expression defined by the query and subsequent Lisp compilation of the resulting smaller parts will solve the problem. Code reduction by code sharing, i.e. a more extensive use of common subexpression elimination, is of course another way to be considered.

6 Performance

The run time performance and sometimes even the effective computability of a query depends heavily on the number and size of the intermediate relations computed during fixpoint iteration. The propagation of selection does not affect the structure of the program but reduces the size of intermediate results generated during the iterative evaluation of recursive predicates. The Magic Sets transformation, on the other hand, is often the source of complex mutually recursive predicate definitions. Selection propagation, if applicable, mostly outperforms the Magic Sets Transformation. The *LOLA* compiler is able to handle arbitrary recursion. For the sake of run time efficiency, however, it has always been a good strategy to apply selection optimization as extensively as possible and to apply the Magic Set transformation subsequently, if necessary at all.
The performance data listed in *Table 1* have been measured on a SUN 4 workstation with 16 Megabytes main memory. The underlying *LOLA* program is shown in *Figure 1*. Different from the declaration specified there the `timetable` relation has been stored in main memory as a *R-LISP* relation. Time figures for *LOLA* compiling include optimization time. The first and third query permit selection propagation on the first attribute whereas the second does not. Note, that the efficiency gain obtained through the Magic Set transformation is rather low for the binding pattern (`$f, $b`). This is due to the cyclic base data because every airport is contained in the *magic* relations and thus the size of intermediate relations is actually not reduced whereas selection propagation provides for a drastic reduction of the

starting tuple set.

7 Applications

The *LOLA* system serves as a tool for several application projects in an industrial environment as well as in academic research projects.

- In the CSKB (Common Source Knowledge Base) project a structuring instrument for the product support of automobile manufacturers is developed. An object oriented approach is used in order to represent the structural knowledge whereas integrity constraints will be represented as a logic program. CSKB is a joint project of the Bavarian Research Center for Knowledge Based Systems and the ESG/FEG company as an industrial partner.
- The overall goal of the Knowledge Base Consistency project of the Bavarian Research Center for Knowledge Based Systems is to enhance deductive database systems by a reasoning maintenance component. To this end integrity constraints are included in the rules of a deductive database system and can be used to guide an intelligent backtracking mechanism.
- The *LOLA* project itself makes intensive use of the *LOLA* system as a tool. In particular, a compiler version suitable for bootstrapping is developed.
- AMOS, a large scale system for the morpho-syntactical analysis of Old Hebrew text [25], is intensively used by linguist at the University of Munich.
- A system able to give advice in the area of automobile diagnosis based on an abstraction mechanism is developed.
- The prototype of a travel assistant fully written in *LOLA* has been implemented. The system serves as a midsize example for *LOLA* applications

8 Conclusion

The *LOLA* language and an overview of the *LOLA* system, a prototype of a deductive database system have been presented. The *LOLA* system is fully implemented and running. Main features are

- declarative semantics
- set-oriented bottom-up query evaluation
- type declarations
- direct and easy access to external DBMS
- link to the host language by built-in predicates
- modular architecture
- flexible query optimization

Future work will essentially concentrate on relations containing nonground terms, language extensions allowing user controllable sideways information passing, a strategy module for automatically controlling the type and order of possible optimizations, a debugger which is critical for medium and large scale applications and, of course, additional optimization modules.

Acknowledgement

Prof. Rudolf Bayer and Prof. Ulrich Güntzer started the *LOLA* project and have always supported the research presented in this paper. The authors wish to thank Elli Prochaska and Anne Einenkel for technical and organizational support and all the many students for contributing to the project by their theses.

References

[1] *Allegro* COMMON LISP *User Guide*, Manual, Franz Inc., Berkeley, California, 1988

[2] Apt K.R., Blair H.A. and Walker A.: *Towards a Theory of Declarative Knowledge*, in: *Foundations of Deductive Databases and Logic Programming*, Minker J. (ed.), Morgan Kaufmann, Los Altos, 1987

[3] Argenton H.: *Magic Set Transformation und Negation*, Diploma Thesis, Technische Universität München, Munich, 1990

[4] Balbin I., Ramamohanarao K.: *A Differential Approach to Query Optimisation in Recursive Databases*, Technical Report 86/7, Department of Computer Science, University of Melbourne, 1986

[5] Balbin I., Port G.S., Ramamohanarao K.: *Magic Set Computation for Stratified Databases*, University of Melbourne, Dep. of Computer Science, Technical Report 87/3, Parkville, Australia, 1987

[6] Bancilhon F., Ramakrishnan R.: *An Amateur's Introduction to Recursive Query Processing Strategies*, Proc. ACM SIGMOD, 1986

[7] Bancilhon F. et al.: *Magic Sets and Other Strange Ways to Implement Logic Programs*, Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, 1986

[8] Bayer R.: *Query Evaluation and Recursion in Deductive Database Systems*, TUM-I8503, Technische Universität München, Internal Report, Munich, 1985

[9] Beeri C. et. al.: *Sets and Negation in a Logic Database Language (LDL1)*, Proc. ACM PODS 1987, ACM, 1987, pp. 21-37

[10] Beeri C., Ramakrishnan R.: *On the Power of Magic*, Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, 1987

[11] Freitag B.: *Bottom-Up Evaluation of Logic Queries*, Technische Universität München, Internal Report, Munich, 1989

[12] Freitag B., Specht G.: *A Parsing System based on a Deductive Database*, Proc. German Workshop on Artificial Intelligence GWAI '89, Springer, 1989

[13] Güntzer U., Kiessling W, Bayer R.: *On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration*, Proc. 3rd International Conference on Data Engineering, Los Angeles, 1987

[14] Hager J.: *Optimierung von Selektion und Existenzquantifikation in Logikprogrammen*, Diploma Thesis, Technische Universität München, Munich, 1990

[15] Huber Th.: *Typ-Überprüfung für die Logiksprache LOLA*, Diploma Thesis, Technische Universität München, Munich, 1990

[16] Kempe H., Lenz Th.: *Eine Programmierschnittstelle für TransBase in Allegro* COMMON LISP, in german, Technische Universität München, Internal Report, Munich, 1989

[17] Lloyd J.W.: *Foundations of Logic Programming*, Springer, 1987

[18] Moser M.: *Eine relationale Algebra mit Unifikationsoperatoren*, Diploma Thesis, Technische Universität München, Munich, 1990

[19] Mycroft A., O'Keefe R.A.: *A Polymorphic Type System for Prolog*, Artificial Intelligence 23, 1984, pp. 295-307

[20] Przymusinski T.C.: *On the declarative semantics of deductive databases and logic programs*, in: Minker J.,Ed.: Foundations of deductive databases and logic programming, Morgan Kaufmann,1988, pp. 193-217

[21] Ramakrishnan R., Beeri C., Krishnamurthy R.: *Optimizing Existential Datalog Queries*, Proc. ACM PODS, ACM, 1988

[22] Schütz H.: *R-Lisp - Eine erweiterte relationale Algebra in Lisp*, in german, Technische Universität München, TUM-I9049, Munich 1990

[23] Schütz H.: *Ein Typkonzept für eine Hornklausel-Sprache*, in german, Technische Universität München, Internal Report, Munich, 1989

[24] Specht G.: *Die Logiksprache LOLA und ihre interne Darstellung durch Relationen*, in german, Technische Universität München, TUM-I8910, Munich, 1989

[25] Specht G.: *Wissensbasierte Analyse althebräischer Morphosyntax; Das Expertensystem AMOS* EOS-Verlag, 1990

[26] Takashi C.: *Design Overview of the NAIL! System*, E. Shapiro (Ed.), Third International Conference on Logic Programming, London, Juli 1986, Springer Verlag, 1986, pp. 554-569

[27] *TransBase Relational Database System, System Guide, Version 3.3*, Manual, TransAction Software GmbH, Munich, 1989

[28] *TransBase Relational Database System, Programming Interface TBX, Version 3.3*, Manual, TransAction Software GmbH, Munich, 1989

[29] Ullman J.D.: *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies*, Computer Science Press, Rockville, 1989

Appendix: *LOLA* syntax definition

```
<Unit>              ::=  <LOLA-program>
                       | <LOLA-query>
<LOLA-program>      ::=  <program-id> <program>
<LOLA-query>        ::=  <query-id> <query> <program>
<program-id>        ::=  $program(<program-name>).
<query-id>          ::=  $query(<query-name>).
<program>           ::=  {<declaration> | <rule>}*
<query>             ::=  :- <goal>.

<declaration>       ::=  $predicate(<predicate>, <type-list>).
                       | $function(<function>, <type-list>, <type>).
                       | $built_in(<predicate>, <type-list>,
                                  <binding-request>).
                       | $relation(<predicate>, <type-list>, <db-name>).
                       | $database(<db-name>, <db-type>).
<type-list>         ::=  []
                       | [<type> {, <type>}*]
<type>              ::=  <type-variable>
                       | <type-constructor>
                       | <type-constructor>(<type-list>)

<type-variable>     ::=  <variable>
<type-constructor>  ::=  <functor>
<binding-request>   ::=  []
                       | [<binding> {, <binding>}*]
<binding}           ::=  {$f | $b}
<db-name>           ::=  <functor>
<db-type>           ::=  {$main | $transbase}

<rule>              ::=  <head> :- <body>.
                       | <head>.
<head>              ::=  <atom>
<body>              ::=  <goal>
<goal>              ::=  <subgoal>
                       | <subgoal>, <goal>
<subgoal>           ::=  <atom>
                       | <term> = <term>
                       | $not (<goal>)
                       | $exists(<variable>, <goal>)
<atom>              ::=  <predicate>
                       | <predicate>(<term> {, <term>}*)

<term>              ::=  <variable>
                       | <number>
                       | <functor>
                       | <functor>(<term> {, <term>}*)
                       | <list>
<list>              ::=  []
                       | [<term> {, <term>}*]
                       | [<term> {, <term>}* '|' <term>]
                       | <string>

<program-name>      ::=  <functor>
<query-name>        ::=  <functor>
<predicate>         ::=  <functor>
<function>          ::=  <functor>
<functor>           ::=  {a-z} {a-z | A-Z | 0-9 | $ | ! | _}*
                       | <quoted-sequ>
<variable>          ::=  { A-Z | _} {a-z | A-Z | 0-9 | $ | ! | _}*
<number>            ::=  {+ | -} {0-9}+
<string>            ::=  "<allowed-char>*"
<quoted-sequ>       ::=  '<allowed-char>*'
<allowed-char>      ::=  <ASCII-Char.between ! and ~ except \,  " >
                       | \{<any character>}

<separation>        ::=  { <separator> }*
<separator>         ::=  <TAB> | <BLANK> | <CR> | <comment>
<comment>           ::=  /* { <any character except '*/'> }*  */
                       | %  { <any character except <CR> }* <CR>

<keywords >         ::=  {$program, $query, $predicate, $function,
                          $built_in, $relation, $database, $main,
                          $transbase, $f, $b, $not, $exists}
```