A Formal Semantics for a Rule-Based Language*

Herbert Groiss

Institut für Informatik
Universität Klagenfurt
Universitätsstr. 65
A-9020 Klagenfurt, AUSTRIA
e-mail: herb@ifi.uni-klu.ac.at

Abstract

In this paper a formal declarative semantics of the rule-based language L℗LA is defined. At first we describe informally this language, the semantics of L℗LA is then defined in terms of Datalog$^{f\neg}$, a logic programming language, which has a well-defined minimal model semantics and fixpoint semantics. The availability of a clean semantics has several advantages: it allows program analysis and optimization as well as judging the correctness of implementations.

1 Introduction

The semantics of most rule languages is defined by giving an informal description of the evaluation algorithm and the conflict resolution strategy. In such languages the results of certain programs are indeterministic. Namely, in OPS5 the conflict resolution strategy is incomplete, [For81]), i.e. in certain cases an instantiation is arbitrarily chosen. It is left to the implementation which rule fires next and - as a possible consequence - which result the program has.

To overcome these shortcomings, we propose the rule-based language L℗LA , whose semantics is defined formally. With that, the behaviour of every possible program is well defined. For specifying the semantics of L℗LA we use an extension of Datalog, namely Datalog$^{f\neg}$. This language has a declarative semantics, which can be defined in different ways: as model theoretic semantics and as fixpoint semantics. The result of a program is the minimal model or the unique fixpoint. The situation calculus is used to capture the meaning of database changes performed by a rule program. The semantics of L℗LA is described by giving a translation procedure from L℗LA programs to equivalent Datalog$^{f\neg}$ programs and by defining the mapping from the minimal Model M_P of this program to the set of facts representing the result of the original L℗LA program.

In contrast to the wide application area of rule-base systems in the fields of artificial intelligence and databases (the so called active databases), little work has been done to formalize the semantics of such languages. In [Wid92], Jennifer Widom presents a denotational semantics of the Starburst rule language. The main difference to L℗LA is the kind of evaluation: In Starburst the evaluation is

*appears in IJCAI-93 Workshop on Production Systems and their Innovative Applications

set-oriented, i.e., in each cycle, an action is applied to a set of tuples, not a single tuple. In [ZH90] a fixed point semantics of rule trigger systems is presented. This allows to formulate a sufficient criteria for identifying the class of programs computing an unique least fixpoint independent from the rule evaluation order. The situation calculus is used to formalize database updates in [Rei92].
In the next section we present the formal basics of Datalog and the extensions we define. Afterwards an informal description of L℗LA is given. In section 4 the formal basis of L℗LA is then defined in terms of Datalog$^{f\neg}$. Concluding remarks and an outlook are given in section 5.

2 Preliminaries

In the following we assume the reader is familiar with first order logic. However, we begin by reviewing some well-known concepts of first-order logic and logic programming. The main notations used throughout the paper are presented in this section.
For a deeper introduction into the field of logic programming and databases we refer to [CGT90] or [Ede92].

2.1 Datalog

2.1.1 Syntax

The syntax of Datalog is similar to that of PROLOG. A Datalog program consists of a finite set of $rules$ and $facts$. Both, rules and facts are represented as Horn-Clauses with the following syntax: $L_0 : -L_1$ & ... & L_n. Each L_i is a literal of the form $p_i(t_1, ..., t_n)$, p_i is a predicate symbol and the t_i are terms. A term can be a constant or a variable. Throughout the paper we use lowercase letters from the start of the alphabet to represent constants $(a, b, c, ...)$, lowercase letters from the end of the alphabet to represent variables $(s, ..., x, y, z)$.
The left-hand side of a Datalog clause is called its $head$ and the right-hand side is called its $body$. The body of a clause may be empty. Clauses with an empty body represent facts, clauses with at least one literal in the body represent rules. A literal or clause which does not contain any variables is called $ground$. The set of predicate symbols $Pred$ is divided into two parts, $EPred$ ($extensional$ predicates) contains all predicates occuring in facts stored in the database. $IPred$ (for $intensional$ predicates) is the set of the predicates occuring in the program but not in the extensional database.
To guarantee the $safety$ of a Datalog program P, i.e. the finiteness of the set of facts that can be derived by the program, it must satisfy the following conditions: Each fact must be ground, each variable which occurs in the head of a rule of P must also occur in the body of the same rule.

2.1.2 Semantics

Each Datalog Fact F can be identified with an atomic formula of First-Order-Logic. Each Datalog rule R of the form specified above represents a first order formula $R*$ of the form $\forall x_1 ... \forall x_m (L_1 \wedge ... \wedge L_n \rightarrow L_0)$, where $x_1, ..., x_m$ are all the variables occuring in R. A set S of Datalog clauses corresponds to the conjunction of all formulas $C*$ such that $C \in S$.
The semantics of a logic program is defined by means of particular models of the program. Again, we make some definitions. The Herbrand base HB is the set of all ground facts of the form

$p(c_1, ..., c_n)$, where p is a predicate symbol in $Pred$ and all c_i are constants. Analogous to EPred and IPred we define the extensional HB (EHB) and the intensional HB (IHB).
A Herbrand Interpretation HI is a subset of the HB, i.e. the set of ground facts holding in the interpretation. Implicitly, ground facts not in HI are false for HI. If a clause C is true under a given interpretation I, we say that this interpretation $satisfies$ C and that I is a $model$ for C. A Herbrand Interpretation M is a *minimal model* of a set of clauses F iff M is a model of F and for each M' such that M' is a model of F, $M' \subseteq M$ implies $M = M'$.
A ground fact $p(c_1, ..., c_n)$ is true under the interpretation I iff $p(c_1, ..., c_n) \in I$. A Datalog rule is true under I iff for each substitution θ which replaces variables by constants, whenever $L_1 \in I \wedge ... \wedge L_n \in I$, then it also holds that $L_0 \in I$.
The declarative semantics of a Datalog program P is simply defined as the minimal Herbrand model of P. The semantics of Datalog can also be defined with fixpoint theory, see [CGT90].

2.2 Functions

Built-in predicates (or "built-ins") are expressed by special predicate symbols such as $<, >, \leq$, etc. with a predefined meaning.
In most cases built-ins represent infinite relations, therefore the Herbrand model of a Datalog program using built-ins is not necessarily finite. Safety can be guaranteed by requiring that each variable occurring as an argument of a built-in predicate in a rule body must also occur in an ordinary predicate of the same rule body, or must be bound by an equality (or a sequence of equalities) to a variable of such a predicate or a constant.
In a similar way, functions can be used, for example arithmetic functions $(+, -, *, /)$ or user-defined functions. A predicate $plus(x, y, z)$ can be used to express the relation $x + y = z$. The "input variables", x and y must occur in an ordinary predicate of the rule body. The function can then be evaluated as soon as these variables are bound. Note, that for guaranteeing the finiteness of the Herbrand model all arguments would have to be bound in ordinary predicates.

2.3 Negation

In pure Datalog, negated literals in rules or facts are not allowed. However, we may infer negative facts by adopting the closed world assumption (CWA): If a fact F does not follow from a set of Datalog clauses, then we conclude that the negation of F, $\neg F$, is true.
The extension of pure Datalog including negated literals in the body of rules is called $Datalog^{\neg}$. For safety reasons we require that each variable occuring in an negative literal in the rule body also occurs in a positive literal in the same body. A set of Datalog$^{\neg}$ clauses may have more than one minimal model. Stratified Datalog$^{\neg}$ programs are a subclass of Datalog$^{\neg}$, where one distinguished minimal model can be selected as the model of the program. For this we require, that each negative literal in the body of a rule can be evaluated before the predicate of the head of the rule is evaluated. If a program fulfills this condition it is called stratified. Any stratified program can be partitioned into disjoint sets of clauses $P = P^1 \cup ... \cup P^n$ called strata, such that each P^i contains only clauses whose negative literals correspond to predicates defined in lower strata. The evaluation is now done

stratum-by-stratum. First, P^1 is evaluated, by applying the CWA locally to the EDB. Then the other strata are evaluated in ascending order.
A refinement of stratification is local stratification, where the Herbrand Base instead of the predicates is divided into strata. If the HB is infinite (due to the use of functions), we can have an infinite number of strata. As above, the evaluation can be done with fixpoint iteration.
A subclass of locally stratified programs are the so-called *acyclic* programs [AB91]. We define a level mapping $\mathcal{L}$: HB $\rightarrow \mathbb{N}$ of ground facts to natural numbers. If in every ground instance of every rule of P, $\mathcal{L}(L_i) < \mathcal{L}(L)$, i.e., the level of all literals of the body is less than that of the head, then program P is acyclic. The level of a literal also defines its stratum, every acyclic program is also locally stratified.

Example 1. Let us consider a logic program that defines the predicate $even$ for natural numbers:

$even(0).$
$even(y) :- succ(x, y)\ \&\ \neg even(x).$

$succ$ is the successor function, defined as: $succ(i, i + 1)$ for all $i \in \mathbb{N}$. The level mapping can be defined as $\mathcal{L}(even(x)) = x, \mathcal{L}(succ(x, y)) = 0.$
It is easy to see, that: $\mathcal{L}(even(x + 1)) > \mathcal{L}(even(x))$ and $\mathcal{L}(even(y)) > \mathcal{L}(succ(x, y))$. The program is therefore acyclic and, as a consequence, locally stratified.

In the following we denote by *Datalog*$^{f\neg}$ acyclic Datalog$^{\neg}$ with functions.

2.4 Situation Calculus

For formalizing database updates we use a variation of the *situation calculus*, originally developed by McCarthy, [McC68]. Those relations, whose truth values may vary from state to state, are called fluents and are denoted by predicate symbols taking a state term as additional argument. This state term specifies the particular state (or situation), in which the fact is true.
The main difficulty with this formalism is the so-called *Frame problem*: "Which facts holding in an earlier situation are still valid in a later situation?" or in other words: which facts are not invalidated by an action?
Here we propose a simple solution, which is, however, less general than others. For each fluent p we define the predicates p' and $p^{\neg}$, either with the state term as additional argument. p' is used to emphasize that p holds in a state s, $p^{\neg}$ represents that p becomes false in a situation s.
A term $p(x_1, ..., x_n)$ is then true, if p' with the same arguments was true in an earlier state and has not been invalidated by $p^{\neg}$ since then. More formally:

Definition 2.
$p(x_1, ..., x_n)$ is true in situation s, iff $\exists s_1 \forall s_2 (p'(x_1, ..., x_n, s_1) \wedge \neg p^{\neg}(x_1, ..., x_n, s_2) \wedge s > s_2 > s1)$

By using single numbers as state term, we can formulate this calculus in Datalog$^{f\neg}$. To guarantee acyclic programs, we demand that:

1. each fact has a unique state constant,
2. for all rules and all possible substitutions: the state variable of the head of the rule is greater than all state variables of literals in the body.

The level mapping can then be defined as: $\mathcal{L}(p(x_1, ..., x_n, s)) = s$.
So far, we have defined the formal basis for the formulation of the semantics of L²LA . Before doing this, we describe the syntax and behaviour of L²LA programs informally.

3 L²LA - a logic oriented language for active databases

3.1 Syntax

In this section we assume that the reader is familiar with the principles of rule-based programming. A detailed description of L²LA is given in [Gro92]. Programs consist of facts and rules. Facts are ground facts, denoted by the predicate and a list of constants separated by commas, for example `father("John", "Bill");` The quotes are used to distinguish constants from variables.
The principal structure of a L²LA rule is shown below:

```
rule rulename
    condition_1 &
       ...      &
    condition_n
 -->
    action_1 , ... , action_n
```

Rules start with the keyword `rule`, have a rule name, a condition part, and an action part. The condition part or left-hand-side (LHS) of a rule is a conjunction (denoted by `&`) of conditions. Each condition can be negated, denoted by a preceding `~`, or not. Moreover, one non negated condition can be preceded by ++ or −− for specifying the triggering event. The first symbol means that the rule triggers whenever a fact matching the literal is inserted, the latter means the deletion of a fact. If no such event is specified, the rule triggers, whenever all conditions are met.
The right-hand-side (RHS) or action part of the rule comes after the symbol −−> and is a list of database modification actions separated by commas. The possible actions are insert, modify and remove.
Each rule has a priority, which can be specified in two ways: by the lexical order of the rules in the source file, where the first rule has highest priority, or by specifying it explicitly with a natural number, where 1 is the lowest priority. This numbering must be complete and unambigous. In the following we use the first kind of specification.

Example 3. In a table `teams` the names of teams and their points are stored. The points of a team are raised by two points, when a play between the team and another one took place and the team shot more goals than the other team.

```
rule winner
++ play(team1,team2,g1,g2) &
   >(g1,g2) &
   team(team1,points)
  --> modify(3, team1, points+2);
```

The first argument of `modify` specifies the condition element, the remaining are the arguments of the modified element. In this example, the fact matching the third condition (`team(...)`) is modified.

3.2 Evaluation

In this subsection we informally describe the evaluation of L²LA programs. The traditional way doing this is by specifying the recognize-act cycle as a function and describing the conflict resolution strategy.

FUNCTION INTERPRET
INPUT: R, the set of rules, and DB the initial Database facts
BEGIN
MATCH(DB);
WHILE cs $\neq \emptyset$ DO
    delta := APPLY(NEXTACTION(cs));
    MATCH(delta);
    ENDDO
END.

The instantiations are stored in the so-called conflict stack, denoted by cs. The functions called in INTERPRET are defined as follows:

MATCH: All instantiations of rules with the database and delta are searched. When an instantiation is found, it is pushed onto the conflict stack. MATCH processes all rules beginning with the rule with the lowest priority, and beginning with the most recent data elements.

NEXTACTION: The first action of the instantiation on the top of the stack is returned and removed from the stack.

APPLY: The action is executed. The result is a database tuple, which has been inserted, deleted or modified.

Due to the fact, that always the actions on top of the stack are evaluated, the search strategy of MATCH reflects the conflict resolution strategy. The evaluation is depth-first and for this respect similar to the conflict resolution strategies of OPS5.

4 Semantics of L⁰LA

In this section, we define the semantics of L⁰LA by giving an algorithm for translating a L⁰LA program L into an equivalent Datalog$^{f\neg}$ program P. The result of P, the minimal model M_P, is then re-translated to a set M_L, the result of the original program L.
As mentioned above, we use a kind of situation calculus for describing the updates in the database. Each fact initially in the database or asserted by a rule has a unique state variable s. We have constructed a formula that guarantees this uniqueness and specifies a total order of all possible facts in the database. The value of s for a fact asserted by a rule is computed from the state variables of the elements matching the conditions of the rule (c_i), the rule priority (R), and the position of the action in the RHS of the rule (k):

$$s_r = max(c_i) + R/h^d + c_1/h^{2*d} + ... + c_n/h^{d*(n+1)} + k/h^{d*(n+2)}$$

The constant h is the maximum of three values: the number of rules, the number of facts initially in the database, and the maximum number of actions of any rule (a):

$$h = max(|Rules|, |EDB|, a) + 1$$

The depth of the evaluation tree, d, is the maximum of $de(s)$ for all elements of the instantiation. de is computed by the function below:

$$de(s) = \begin{cases} 1 & \text{if } s * h = \lfloor s * h \rfloor \\ de(s * h) + 1 & \text{otherwise} \end{cases}$$

In the following the relations representing the functions which compute the state variables for a rule r and action k are denoted by $next_{rk}(s_r, c_1, ..., c_n)$, where the c_i's are the input variables and s_r is the output variable.
The translation of the facts from a L⁰LA program to Datalog$^{f\neg}$ is simply done by adding a state constant to each of them. For this constants we use natural numbers, mapped to the facts according to their order in the source file, starting with 1.
We can now formulate the translation procedure for the rules:

INPUT: a L⁰LA rule
OUTPUT: one or more Datalog$^{f\neg}$ rules.

1. Replace the RHS-action modify by remove and insert.
2. If there is more than one literal in the RHS, split the rule R in n rules R1 to Rn with the corresponding literals in the RHS.
3. In the following X stands for $x_1, ..., x_n$.
   A literal `p(X)` in the left hand side of a rule is translated to $p'(X, s_i)$ & $\neg p^{\neg}(X, s'_i)$ & $s_i < s'_i < s_R$ (where i is the position of the literal in the LHS of the rule).
4. A literal `++p(X)` is translated to $p'(X, s_m)$, a literal `--p(X)` is translated to $p^{\neg}(X, s_m)$.

5. If there is a literal preceded by ++ or −− in the LHS of the rule, the body is completed with the following literals: $s_m > max(s_1, ..., s_n)$ & $next_{rk}(s_r, s_m, s_1, ..., s_n)$, where s_m is the state variable of this literal and $s_1, ..., s_n$ are the state variables of the other literals of the LHS.

   In the other case the body is completed with: $next_{rk}(s_r, s_1, ..., s_n)$

6. Terms containing (arithmetic) functions in the RHS of the rule are replaced with variables, the predicates for these functions are added to the rule body.

7. The actions in the RHS of the rules are handled in the following way:

   a) insert `p(X)`: the head of the rule is $p'(X, s_r)$

   b) remove `p(X)`: the head is $p^{\neg}(X, s_r)$

With this steps done for all rules of a LOLA program L an equivalent Datalog$^{f\neg}$ program P can be generated. The program P is acyclic, because $s_R > max(s_i)$, by definition of $next_{rk}$. The result of P is the minimal model M_P. The result of the original program L is the set of facts M_L, defined in analogy to definition 2:

Definition 4.
$M_L := \{p(x_1, ..., x_n) | \exists s_1 : p'(x_1, ..., x_n, s_1) \in M_P \wedge \forall s_2 > s_1 : p^{\neg}(x_1, ..., x_n, s_1) \notin M_P\}$

The following example shows how the translations of a LOLA program is performed:

Example 5. This program computes the sum of all `x` in elements `a(x)`, whenever an element `sumcmd()` is inserted into the database.

```
sum(0,0);
rule sum1  ++ sumcmd() & a(i) --> suma(i);

rule sum2 suma(i) & sum(j,k) -->
   remove(1), modify(2, +(i,j), +(k,1));
```

The translation of the first rule is:

$suma'(i, s_r)$:–
$a'(i, s_1)$ & $\neg a^{\neg}(i, s_1')$ & $s_1 < s_1' < s_r$ & $sumcmd(s_m)$ & $s_m > s_1$ & $next_{sum1}(s_r, s_m, s_1)$

The second rule must be split into three rules, all with the following body (i runs from 1 to 3):

BODY: $suma'(i, s_1)$ & $\neg suma^{\neg}(i, s_1')$ & $s_1 < s_1' < s_{ri}$ & $sum'(j, k, s_2)$ &
$\neg sum^{\neg}(j, k, s_2')$ & $s_2 < s_2' < s_{ri}$ & $next_{sum2i}(s_{ri}, s_1, s_2)$

the complete rules are:

$suma^{\neg}(i, s_{r1})$:– BODY.
$sum^{\neg}(j, k, s_{r2})$:– BODY.
$sum'(m, l, s_{r3})$:– BODY & $plus(i, j, m)$ & $plus(k, 1, l)$.

The next example shows that the behaviour of LOLA programs can be analyzed using the corresponding Datalog$^{f\neg}$ program.

Example 6.

```
p(0);
rule r1 p(x) --> p(x+1);
rule r2 p(x) --> remove(1);
```

Intuitively it is clear that the program behaves different depending on the priority of the rules: If rule r2 is fired first, the program stops. Otherwise, it does not terminate, because rule r1 computes p(x) for all $x \in \mathbb{N}$.
Due to the formal semantics we can investigate under which conditions the program does not terminate. The translation to Datalog$^{f\neg}$ yields to the following program:

$$p'(0,1).$$
$$p'(y,s_{r1}) :- p'(x,s_1) \;\&\; \neg p^{\neg}(x,s_1') \;\&\; s_{r1} > s_1' > s_1 \;\&\; next_{r1}(s_{r1},s_1) \;\;\&\;\; plus(x,1,y).$$
$$p^{\neg}(x,s_{r2}) :- p'(x,s_1) \;\&\; \neg p^{\neg}(x,s_1') \;\&\; s_{r2} > s_1' > s_1 \;\&\; next_{r2}(s_{r2},s_2).$$

We investigate now under which conditions the first rule is fired by analyzing the Datalog$^{f\neg}$ program. The second rule can be inserted into the first, this leads to:

$$p'(y,s_{r1}) :- p'(x,s_1) \;\&\; s_{r1} > s_{r2} > s_1 \;\&\; next_{r1}(s_{r1},s_1) \;\&\; next_{r2}(s_{r2},s_1) \;\;\&\;\; plus(x,1,y).$$

The body can be further simplified to: $p'(x,s) \;\&\; next_{r1}(s) > next_{r2}(s) \;\;\&\;\; plus(x,1,y)$.
From the definition of the function $next$, we can follow that this expression becomes true only if $R_{r1} > R_{r2}$, i.e. the priority of r1 is higher than that of r2, otherwise the rule never fires.

The next example shows how the semantics defines an ordering between multiple instantiations of the same rule:

Example 7. The rule r1 fires once and remove the tuples of the instantiation from the database.

```
p(1);
p(2);
q(1);
rule r1 p(x) & q(y) --> remove(1), remove(2);
```

The corresponding Datalog$^{f\neg}$ program is:
$p(1,1), p(2,2), q(1,3)$,
$p^{\neg}(x, s_{r11})$:– BODY,
$q^{\neg}(y, s_{r12})$:– BODY.
where BODY is: $p'(x,s_1) \;\&\; \neg p^{\neg}(x,s_2) \;\&\; s_1 > s_2 > s \;\&\; q'(y,s_3) \;\&\; \neg q^{\neg}(y,s_3) \;\&\; s_3 > s_4 > s \;\&\; next_{r1}(s_1,s_2,s_{r1i})$.
We compute the state variable of the head of the first rule for both instantiations: h is 3, d is 1, the rule priority is 1, the state variable of $p^{\neg}(1, s_{r11})$ is therefore:

$s_{r11} = 3 + 1/4 + 1/16 + 3/64 + 1/256 = 3.36328125$
and of $p^{\neg}(2, s'_{r11})$ it is: $3 + 1/4 + 2/16 + 3/64 + 1/256 = 3.42578125$
s_1 is smaller than s_2, the $level$ of the corresponding fact is therefore lower and the fact is derived. The next fact which is derived is $q^{\neg}(1, s_{r12})$. With this a minimal model is found. Note that selecting the other instantiation does not lead to a model of the program.
The resulting database contains therefore only the fact `p(2)`.

5 Conclusions

A formal declarative semantics of the rule-based language L°LA has been presented. The advantages of having a well-defined semantics are numerous: The semantics can be used as a basis for program analysis, for example termination checking, and optimization. The various evaluation and optimization techniques developed for logic programming become applicable. Moreover, the formulation of L°LA in terms of Datalog$^{f\neg}$ should make the unification of active and deductive databases easier, because most languages for deductive databases are based on Datalog. This combination allows the development of languages suitable for a wider application area.

References

[AB91] Krysztof R. Apt and Marc Bezem. Acyclic Programs. *New Generation Computing*, 9:335–363, 1991.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

[Ede92] Johann Eder. Logic and Databases. In V. Marik, editor, *Advanced Topics in AI*. Springer-Verlag, 1992.

[For81] C. L. Forgy. OPS5 Users's Manual. Technical report, Department of Computer Science, Carnegie-Mellon University, 1981.

[Gro92] Herbert Groiss. *Eine logikorientierte Sprache für aktive Datenbanken*. PhD thesis, Technical University Vienna, 1992.

[McC68] John McCarthy. Programs with common sense. In M. Minsky, editor, *Semantic Information Processing*, pages 403–418. The MIT Press, 1968.

[Rei92] Raymond Reiter. On Formalizing Database Updates: Preliminary Report. In *Proc. of EDBT*, 1992.

[Wid92] Jennifer Widom. A Denotational Semantics for The Starburst Rule Language. *SIGMOD Record*, 21(3):4–9, 1992.

[ZH90] Yuli Zhou and Meichun Hsu. A Theory for Rule Triggering Systems. In *Proc. of EDBT*, 1990.