Conditional Term Rewriting as a Deductive Database Language

Zeki O. Bayram, Barrett R. Bryant
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, Alabama 35294-1170, U.S.A.
InterNet: bayram@cis.uab.edu, bryant@cis.uab.edu

1 Introduction

The logic data model, as defined by the Datalog query language [14], is the foundation on which deductive databases are built [11]. In Datalog, the database is partitioned into two parts: the extensional database (EDB), which contains the "hard" facts, stored as relations, and the intensional data base (IDB), which is derived from the EDB through the use of rules, or clauses, which are part of the database. Datalog (with negation) is more powerful than the relational data model. For example, we can define clauses in Datalog to compute the transitive closure of a relation, a task which is not possible in the relational model. However, to see a limitation of Datalog, consider the following set of facts/rules in *Datalog* that represent a certain family relationship.

parent(john,jack).
parent(jack,mary).

How are we to interpret the *parent* predicate? Is *john* the parent of *jack*, and *jack* the parent of *mary*, and thus *mary* the grand–daughter of *john*? Or is it the other way around, and *mary* is the grandmother of *john*? We develop a data model whereby, for example, we would represent the above data as:

parent(john) $\rightarrow$ jack.
parent(jack) $\rightarrow$ mary.

This kind of representation removes any ambiguity from binary relations. Furthermore, we can have more intuitive queries. For example: if we want to find the grandparents of *john*, we could query as *parent(parent(john))*. Therefore, we need to add the capability of representing functions in the logical database, effectively adapting functional/logic programming to the area of deductive databases, much as logic-based deductive databases are an adaptation of logic programming.

Functional/logic programming languages integrating the two paradigms of functional and logic programming have been well studied and investigated [7]. Most of these languages are based on narrowing as their operational semantics and have as their programs functions specified in the form of a set of equations, or rewrite rules. Using narrowing as the operational semantics of functional languages was pioneered by Reddy [13]. Functional/logic programs composed of conditional term rewriting systems, with conditional narrowing as the operational semantics, have a special appeal due to their conciseness and expressiveness [8].

In this paper, we describe a functional/logic deductive data base model based on conditional term rewriting systems for defining logic data bases with functional syntax. In this approach, we use conditional rewrite rules as a database language, and develop a bottom-up evaluation algorithm that utilizes narrowing for evaluating queries. We demonstrate the soundness and completeness of this query evaluation algorithm with respect to a more standard reduction semantics, and prove that it terminates. The model we develop subsumes the logic data model with Datalog programs making up the database. The advantages of our model over Datalog stem mainly from the fact that our model makes available within a unified framework both relational and functional styles of programming. The availability of the functional style of programming, in addition to the relational style, can be used to good advantage to write programs that have a less ambiguous declarative reading than their purely logical counterparts, to more authentically and naturally represent data (some data can best be represented using a functional notation rather than relational), and pose queries more intuitively through the use of function nesting.

The remainder of the paper is organized as follows. In the next section, we develop the *DataFunLog* (DFL) model which permits the definition of a database with conditional rewrite rules. Also an algorithm for evaluating queries is presented. In section 3 we extend our rules to permit the "not" function to appear anywhere within the body or the condition part of a rule. We call the resulting language *DFLN, "DataFunLog with Negation."* Also in section 3 we introduce the notion of a *stratified* database, and describe an extended bottom−up evaluation algorithm to handle negation for stratified DFLN programs. Section 4 describes other approaches to database languages, and compares them to DFLN. The last section is the conclusion and future research directions.

2 The *DataFunLog* (DFL) Data Model

Our data model is based on conditional term rewriting systems. Since we are dealing with atomic data, and following the lead of *Datalog*, we shall not allow constructors in our rules: just variables and constants. A rule will, in general, be of the form

lhs : condition $\rightarrow$ rhs

with the following restrictions:

1. All variables appearing in the *lhs* must appear in the *rhs* or in the *condition*.
2. All variables must be *restricted*: they must be an argument to a function call in the *condition* or in the *rhs*. e.g. "f(X) $\rightarrow$ X" is not allowed.
3. *lhs* can have no function applications in argument positions: only variables and constants. Thus "f(g(a),X) $\rightarrow$ rhs" is not allowed.
4. All variables occuring in the condition part must be arguments to some function. "f(...):X $\rightarrow$ rhs" is not allowed.

These restrictions on rewrite rules are necessary for the bottom-up query evaluation algorithm to terminate.

To demonstrate the elegance and power of the proposed data model, we give an example *DataFunLog with Negation (DFLN)* program. We also give the intuitive meaning of the rules defining the program. The following rules define certain family relationships:

```
male(joe)→true.
female(mary)→true.
parent(tom)→joe.
brother(X):parent(X) = parent(Y) and male(Y) and not(X=Y) → Y.
father(X):Y=parent(X) and male(Y)→Y.
```

The rules defining "=" and the logical connectives are assumed. In fact, as we shall see later, the rules defining them is included by default in the database. The functions "male" and "female" are boolean valued functions. The "parent" predicate is straightforward, where "parent(X)→ Y" means "the parent of X is Y." The way we read the last two rules are as follows: "For all *constants* X and Y, the brother of X is Y if the parents of X and Y are the same, Y is male and Y is not equal to X" and " the father of X is Y if Y is the parent of X and Y is male."

Given these rules defining the database, we can, for example, ask queries of the form "Whose father is *joe*" by "father(X) = joe" where X is a logical variable. We have developed an algorithm using bottom-up evaluation for answering such queries. For the case involving negation, we have developed a stratification algorithm, and an algorithm which makes use of the bottom-up evaluation algorithm developed for evaluating DFL programs.

2.1 The Meaning of DataFunLog Programs Without Negation

Let us define certain terms that we shall make use of in the remainder of this paper.

We partition all symbols used in a DFL program into three sets: The set of function names F, the set of variables V and the set of atomic objects (also called *constants*, or *data objects*) DOM.

We define a *term* recursively as:

- A variable is a term
- A data object is a term
- If f is a function name, and $a_i, 1 \leq i \leq n$, are terms, then $f(a_1, a_2, \cdots, a_n)$ is a term

An atomic object is also called a term in *normal form*.

An *occurrence* is a string of integers of the form $i.j.k\ldots$, where $1 \leq i, j, k \ldots$ We use occurrences to give an unambiguous address for a particular subterm of a term. If t is a term, we denote the subterm of t at occurrence u as t[u]. A few examples will serve best to describe the meaning of occurrences.

f(g(a,b),c)[1] = f(g(a,b),c)
f(g(a,b),c)[1.1] = g(a,b)
f(g(a,b),c)[1.1.2] = b
f(g(a,b),c)[1.2] = c

If u is pointing to a subterm of t that does not exist, we define $t[u]$ to be t itself. For example, f(g(a,b),c)[1.2.4.2]=f(g(a,b),c).

Obtaining a subterm of a term by the above process is called an *extraction* operation. Replacing a subterm s of a term t at some occurrence u by another term w is called a *replacement* operation and is denoted by t[w/u].

Conditional rewrite rules, in the presence of *confluence*, have a natural declarative reading, in addition to a procedural one, whereby a conditional rewrite rule can be seen as conditional equality between two terms. However, we cannot use this approach to give a declarative reading to DataFunLog programs, since by necessity, we do not have confluence in the rewrite rules. For example, with a database consisting of the following rules,

f(a)→b.
f(a)→c.

where a, b and c are terms in normal form, if we interpreted the rules to mean (unconditional) equality, we would be able to prove that b and c are equal, an obvious falsity.

Instead, the declarative reading of conditional rewrite rules will be that they define set-valued functions. In the above example, f is a (partial) function that maps the constant "a" to the set of constants {b,c}. Answering a query which possibly can contain variables means finding the set of values denoted by the expression that makes up the query, as well as the values for the variables in the query for which the expression denotes this set of values.

In order to give meaning to rules, we need to give a meaning to expressions. Below, we define what an expression denotes, and then what each rule means. First, we define some more terms.

Let σ be a substitution. We say that if all substituted terms in σ are ground, then σ is a *ground substitution.* If all the substituted terms are in normal form, then we say σ is a *normal substitution.* We can combine these different qualifiers to describe a particular substitution.

Let R be the set of rules that make up a DFL program. Let R'={(rule)σ | rule $\in$ R and σ is a ground normal substitution and (rule)σ is ground}.

The meaning of any ground expression *exp*, $Den(exp)$, is the set of constants (terms in normal form) *exp* can be *(conditionally) reduced* to using ONLY the rules in R', i.e. $Den(exp) = \{ a \mid exp \Longrightarrow^* a$, and a is a constant$\}$. The meaning of any non-ground expression *exp* is $\bigcup_\sigma Den((exp)\sigma)$, where σ is a ground normal substitution and $(exp)\sigma$ is ground.

We can now give meaning to the rules in a DFL program: If $lhs : cond \rightarrow rhs$ is a rule, for any ground normal substitution σ that makes the rule ground, true $\in Den((cond)\sigma)$ implies $Den((rhs)\sigma) \subseteq Den((lhs)\sigma)$.

Alternatively, we can see rules as functions mapping substitutions to sets of constants. For this interpretation, we would need to standardize, or *rectify* the rules in the database. The process of *rectification* proceeds as follows: for any function name f, let "$f(a_1, \ldots, a_n) : cond \rightarrow rhs$" be one of the rules defining f. Replace this rule with "$f(V_1, \ldots, V_n) : V_1 = a_1$ *and* $V_2 = a_2$ *and* $\ldots$ *and* $V_n = a_n$ *and cond* $\rightarrow rhs$," where $V_1, V_2, \ldots, V_n$ are new variables not occurring in *any* defining rule of f. Repeat the same procedure for all defining rules of f. Note that the new rules defining the function f will now have the same variables in their i^{th} argument positions. Then, for any given ground normal substitution σ that makes all the rules defining f ground, we can say f maps σ to $Den((lhs)\sigma)$.

2.2 An Algorithm for Computing the Answer to a Given Query

We shall assume henceforth that every database includes the following rules by default:
$\{x = x \rightarrow true | x \in DOM\}$ as well as the rules

Algorithm 1 *Finding the answer to a given query:*

1. *For every function name f in F, create two sets: call the first set raw_f, and the second set $success_f$. Initialize: $success_f = \emptyset$ and $raw_f =$ all rules defining f.*

2. *Repeat steps 2a through 2d until neither of the sets $success_f$ and raw_f change any more (f is any function name occuring in the program)*

   (a) *Let $f(a_1, a_2, \ldots, a_n)$: $Cond \rightarrow rhs \in raw_f$. If $Cond$ = "true," then add $f(a_1, a_2, \ldots, a_n) \rightarrow rhs$ to raw_f.*

   (b) *Let $f(a_1, a_2, \ldots, a_n)$: $Cond \rightarrow rhs \in raw_f$. Let $Cond[u]$ be a nonvariable subterm of $Cond$ at occurrence u. Let $g(b_1, b_2, \ldots, b_m) \rightarrow some\_constant \in success_g$ for some function name g. Let $g(b_1, b_2, \ldots, b_m)$ be unifiable with $Cond[u]$ with most general unifier σ. Let $Cond' = Cond[some\_constant/u]$, i.e. replace the subterm of $Cond$ at occurrence u with $some\_constant$. Add the rule $f(a_1, a_2, \ldots, a_n)\sigma : (Cond')\sigma \rightarrow (rhs)\sigma$ to raw_f. This operation is called a* narrowing operation on a condition.

   (c) *Let $f(a_1, a_2, \ldots, a_n) \rightarrow rhs \in raw_f$. Let $rhs[u]$ be a nonvariable subterm of rhs at occurrence u. Let $g(b_1, b_2, \ldots, b_m) \rightarrow some\_constant \in success_g$ for some function name g. Let $g(b_1, b_2, \ldots, b_m)$ be unifiable with $rhs[u]$ with most general unifier σ. Let $rhs' = rhs[some\_constant/u]$, i.e. replace the subterm of rhs at occurrence u with $some\_constant$. Add the rule $f(a_1, a_2, \ldots, a_n)\sigma \rightarrow (rhs')\sigma$ to raw_f. This operation is called a* narrowing operation on the right hand side.

   (d) *Let $f(a_1, a_2, \ldots, a_n) \rightarrow rhs \in raw_f$. If rhs is a constant, then add $f(a_1, a_2, \ldots, a_n) \rightarrow rhs$ to $success_f$*

3. *The answer is the set $success_{answer}$.*

Figure 1: Algorithm for computing the rule set to answer a query

true and true → true	false or false → false
true and false → false	true or false → true
false and true → false	false or true → true
false and false → false	true or true → true

Note that we have adopted an infix notation for the connectives "and" and "or," as well as the "=" function. The precedence of these operators, from highest to lowest, are: =, *and*, *or*.

In figure 1 we give the algorithm for computing the answer to a given query Q. We shall generate a temporary rule of the form "$answer(Q) \rightarrow Q$," where Q is the query. F is the set of function names in the database (as we mentioned before) and D is the database itself made up of (conditional) rewrite rules. D also contains the rule for "answer" generated by the query. To illustrate this algorithm, consider example 1 as follows.

Example 1 Assume the data base is made up of the rules given below. The query is "f(Z)."We show how the given query is computed (using algorithm 1).

f(X) → h(g(X))
g(a) → b
h(b) → c
answer(f(Z)) → f(Z)

We give the sets at each iteration as they are computed.

$DOM = \{a, b, c, true, false\}$.

Initialize: $success_{answer} := success_f := success_g := success_h := success_{and} := success_{or} := success_{=} := \emptyset$. $raw_f := \{f(X) \to h(g(X))\}$. $raw_g := \{g(a) \to b\}$. $raw_h := \{h(b) \to c\}$. $raw_{answer} := \{answer(f(Z)) \to f(Z)\}$. raw_{and} :={rules defining "and"}. raw_{or} :={rules defining "or"}. $raw_{=}$:={rules defining "="}.

Iteration 1: $success_g := success_g \bigcup \{g(a) \to b\}$. $success_h := success_h \bigcup \{h(b) \to c\}$. $success_{and} := success_{and} \bigcup raw_{and}$. $success_{or} := success_{or} \bigcup raw_{or}$. $success_{=} := success_{=} \bigcup raw_{=}$. Note that Iteration 1 places in the success sets the actual data from the "extensional" database.

Iteration 2: $raw_f := raw_f \bigcup \{f(a) \to h(b)\}$.

Iteration 3: $raw_f := raw_f \bigcup \{f(a) \to c\}$.

Iteration 4: $success_f := success_f \bigcup \{f(a) \to c\}$.

Iteration 5: $raw_{answer} := raw_{answer} \bigcup \{answer(f(a)) \to c\}$.

Iteration 6: $success_{answer} := success_{answer} \bigcup \{answer(f(a)) \to c\}$.

At this point, since none of the sets can change any further, we take the contents of $success_{answer}$ to be the answer to the query, in this case $\{answer(f(a)) \to c\}$. □

Since every new rule added to a *raw* set is simpler than some other rule in the same raw set, we can observe that algorithm 1 terminates. Furthermore, if Q is a query, possibly containing variables and "$answer(Q)\sigma \to rhs$" $\in success_{answer}$ at the termination of algorithm 1, then $(Q)\sigma$ is ground, rhs is a constant, and all variables in Q have been replaced by constants (terms in normal form). To see this, replace $answer(Q) \to Q$ with $answer(V_1, V_2, \ldots, V_n) \to Q$ where $V_1, V_2, \ldots, V_n$ are the variables occuring in Q, in the order in which they appear in Q. This is true because for any function f, if $f(a_1, a_2, \ldots, a_n) \to rhs \in success_f$ at any stage of the execution of algorithm 1, then $a_1, a_2, \ldots, a_n$ are all constants, and so is rhs. Therefore, we claim that:

1. Given a DFL program P, $f(a_1, a_2, \ldots, a_n) \to rhs \in success_f$ for any function name f in P iff $rhs \in Den(f(a_1, a_2, \ldots, a_n))$.
2. Given a DFL program P, and a query Q, and a normal ground substitution σ, upon termination of algorithm 1, $answer((Q)\sigma) \to rhs \in success_{answer}$ iff $rhs \in Den((Q)\sigma)$.

2.3 Translation of DataLog to DFL

A Datalog program is composed of facts and implications. If "fact" is a fact, we generate the DFL rule "fact → true." If "a ← b & c & d ..." is an implication, we generate the rule "a → b and c and d ....” If Datalog programs are translated in this fashion to DFL programs, then the DFL program has the same meaning as the source Datalog program. In fact there is almost a one-to-one correspondence between bottom-up evaluation of the original Datalog program, using “naive” evaluation, and the execution of algorithm 1 on the DFL program translated from the Datalog program. Datalog programs with negation are treated similarly, with “not”s remaining intact. For example, “a ← b & not(c) &d ...” would be translated as “a → b and not(c) and d ....” In the next section we discuss how we can extend DFL and the query evalaution algorithm to deal with negation.

3 DataFunLog With Negation

In this section, we generalize DFL programs to allow for the function “not” to appear either in the condition or body of a rule. We shall call such programs *DFLN programs.* In order to accomodate negation, we introduce a special atom “failure” into DOM.

We also assume the following rules will be part of every database, in addition to those added to a DFL database:

true and failure → false	true or failure → true
failure and true → false	failure or true → true
failure and false → false	failure or false → false
false and failure → false	false or failure → false
failure and failure → false	failure or failure → false
not(true) → false	not(false) → true
not(failure) → true	

as well as the set of rules {x=x → true | x ∈ DOM} $\bigcup$ {x=y → false | x,y ∈ DOM and x ≠ y}

In order for the query evaluation algorithm for DFLN programs to work, the DFLN program needs to be *stratified.* A DFLN program is *stratified* if there are no two function names f and g such that g occurs negatively in either the condition or the right hand side of a rule defining f, and g depends on f. The reason we want to classify function names into strata is that we want to put an order on the computation of functions. Suppose we are given the set of rules $\{f(\ldots) : \ldots not(g(b)) \ldots \rightarrow rhs,\ g(a) \rightarrow true\}$. DOM=$\{a, b, true, false, failure\}$. We want to deduce that $g(b)$ must be $false$, since it is not provable ¿from the given facts (i.e. we want to make the so-called *closed world assumption*). We can achieve this by computing all the values X for which $g(X)$ is true, and setting to $false$ ($failure$) all other instantiations of X in $g(X)$. This process would result in our adding “$g(b) \rightarrow failure$” to the provable facts (as well as the rules $g(true) \rightarrow failure$, $g(false) \rightarrow failure$ and $g(failure) \rightarrow failure$) so that when we need to evaluate “$not(g(b))$,” we will have a value for $g(b)$. Of course this should be done BEFORE we try to compute f. Since the computation of f needs g to be already computed before it can proceed, the computation of g should not depend, even indirectly, on f.

The rules making up a DFLN program have a *stratification* if all the function names can be assigned integer values such that for any two functions f and g, if there exists the rule “f(...) : cond → rhs,” then:

- stratum(f) ≥ stratum(g) if g occurs positively in “cond” or in “rhs”

```
Algorithm 2 Finding the answer to a given query for DFLN programs:

1. Let Q be the query. Stratify the DFLN program. Let S[X] = the set of function names at
   stratum X.
2. For X:=1 to NumberOfStrata do
   • Compute the success sets for the functions in S[X], using the success sets of functions in
     S[1] $\bigcup$ S[2] $\bigcup \ldots \bigcup$ S[X] and the raw sets of functions in S[X] by executing algorithm 1
   • Let f be an n-ary function name in S[X]
   • Let $temp_f = \{ f(a_1,\ldots,a_n) \rightarrow failure \mid a_1,\ldots,a_n \in DOM$ and $f(a_1,\ldots,a_n) \rightarrow rhs \notin$
     $success_f$ for any constant $rhs \}$
   • $success_f := success_f \bigcup temp_f$
3. Endfor
4. The answer is the contents of $success_{answer}$
```

Figure 2: Finding the answer to a given query for DFLN programs

- stratum(f) > stratum(g) if g occurs negatively in "cond" or in "rhs"

The algorithm for computing the result of a query is given in figure 2. Our reasoning about termination and the nature of the computed answer holds true for DFLN programs in the same manner as for DFL programs.

4 Related Work

The logic data model has been extended to include objects [1, 2, 9], higher order logic that supports structured data, object identity and sets [10], higher order syntax [5] and various other features to improve its expressiveness. Chomicki [6] extends the logic model by allowing function symbols to appear in logic programs in a restricted way. Besides these, there are countless others which extend the logic data model in various ways, and we couldn't hope to cite them all here.

However, we shall not compare DFL with any of these models which extend the logic model. We shall instead compare it with Datalog, and assert that it acts as a good a platform as Datalog for the extensions made to Datalog that are cited above.

DFL, as we have seen, subsumes the logic model as defined by Datalog. As such, DFL shares all the advantages of the Datalog model, such as treating procedural and factual information in a uniform manner. Its syntax is simple and uniform, just like Datalog's. In addition, however, DFL gives us much more flexibility than Datalog in that we have a choice between the relational and functional styles of programming in one unified framework, which permits us to write easier to understand programs and pose more intuitive queries. The examples we have given in this presentation help to demonstrate this point.

Along similar lines with DFL, Poulovassilis [12] adapts a functional language to the area of deductive databases. The language developed (PFL) has features in common with both logic based deductive database languages, and with functional programming languages. The data model developed there is elegant. However, the syntax and semantics of PFL is not as simple and uniform as in Datalog or DFL, a major strength of both models.

Another approach to deductive databases is given by Bryant and Pan in [4] whereby the *Two-Level Grammar* specification language is adopted to the area of deductive databases. This scheme also has some of the advantages of logic and functional programming, in addition to a natural-language like syntax. It represents however a different line of research than what we presented here.

5 Conclusions

A data model (called *DataFunLog*, or DFL for short) has been presented which adopts functional/logic programming to the area of deductive databases. We have developed a bottom-up query evaluation algorithm for answering queries under this model. The DFL model was then extended to the DFLN model which deals with negation by explicitly making the *closed world assumption*. We introduced the notion of a *stratified* DFLN program, and explained its desirability. We developed an extended query evaluation algorithm which can answer queries on DFLN programs. Our model subsumes the logic model as defined by Datalog. The availability of both the functional and relational (logic) styles of programming in one unified framework facilitates more natural representation of data, both procedural and factual, and more intuitive formulation of queries.

DFL (also DFLN) currently is at the conceptual level; we have not implemented the bottom-up query evaluation algorithms. In an actual implementation, the query evaluation algorithm for DFL programs (which is the counterpart of the *naive* evaluation algorithm for Datalog programs) would need to be modified so that unnecessary computation is avoided (much in the spirit of the *semi-naive* evaluation algorithm for Datalog programs) and also made more goal-directed by making use of the function dependencies. These modifications are relatively straightforward. Other database issues that would need to be considered in an actual implementation include updates, deletions, integrity constraints and concurrency control. Our algorithms have been proven to be sound and complete with respect to a standard reduction semantics and to terminate and these proofs are given in [3].

Future directions for further developing the DFL model include the incorporation of higher order functions (a very distinctive characteristic of functional programming), complex objects, which would allow more authentic representation of certain kinds of information, and program transformation techniques similar to the *magic sets* transformation for Datalog programs (e.g. see [14]) that preserve the meaning of the original program but result in more efficient execution of the bottom-up evaluation procedures.

References

[1] Abiteboul, S., Grumbach, S., Voisard, A., Waller, E. (1989) An Extensible Rule-Based Language with Complex Objects and Data-Functions. Proceedings of the Second International Workshop on Database Programming Languages, pp. 298-314.

[2] Alashqur, A. M., Su, S. Y. W., Lam, H. (1991) A Rule-Based Language for Deductive Object-Oriented Databases. Proceedings of the Sixth International Conference on Data Engineering, pp. 58-67.

[3] Bayram, Z. O., Bryant, B. R. (1992) A Deductive Database Model Based on Conditional Term Rewriting Systems. Technical Report, Department of Computer and Information Sciences, University of Alabama at Birmingham.

[4] Bryant, B. R., Pan, A. (1992) Two-Level Grammar: A Functional/Logic Query Language for Database and Knowledge-Base Systems. Proceedings of the 1992 International Conference on Logic Programming and Automated Reasoning, Springer-Verlag Lecture Notes in Artificial Intelligence, Vol. 624, pp. 78-83.

[5] Chen, W., Kifer, M., Warren, D. S. (1989) HiLog as a Platform for Database Languages (or why predicate calculus in not enough). Proceedings of the Second International Workshop on Database Programming Languages, pp. 315-329.

[6] Chomicki, J. O. (1990) Functional Deductive Databases: Query Processing in the Presence of Limited Function Symbols. Ph.D. Dissertation, Rutgers University.

[7] DeGroot, D., Lindstrom, G. (1986) Logic Programming: Functions, Equations, and Relations. Prentice-Hall.

[8] Dershowitz, N., Okada, M. (1988) Conditional Equational Programming and the Theory of Conditional Term Rewriting. Proceedings of the International Conference on Fifth Generation Computer Systems, pp. 337-346.

[9] Lou, Y., Ozsoyoglu, Z. M. (1991) LLO: an Object-Oriented Deductive Language with Methods and Method Inheritance. SIGMOD Record, Vol. 20, No. 2, pp. 198-207.

[10] Manchanda, Sanjay. (1989) "Higher-Order" Logic As a Data Model. Proceedings of the Second International Workshop on Database Programming Languages, pp. 330-341.

[11] Minker, J. (1988) Foundations of Deductive Databases and Logic Programming. Morgan Kaufman.

[12] Poulovassilis, A., Small, C. (1991) A Functional Programming Approach to Deductive Databases. Proceedings of the Seventeenth International Conference on Very Large Data Bases, pp. 491-500.

[13] Reddy, Uday S. (1985) Narrowing as the Operational Semantics of Functional Languages. Proceedings of the 1985 Symposium on Logic Programming, pp. 138-151.

[14] Ullman, Jeffrey D. (1988) Principles of Database and Knowledge–Base Systems, Volumes I and II. Computer Science Press.