Towards a Real Horn Clause Language

Ravi Krishnamurthy
MCC
Austin, TX 78759
ravi@mcc.com

Shamim Naqvi
MCC
Austin, TX 78759
shamim@mcc.com

ABSTRACT: Current database languages based on Horn clauses and the bottom-up model of computation, such as LDL and Datalog, are not as expressive as Prolog. For example, such languages do not support schema and higher-order predicates in an integrated framework but rely on a separate language to specify the schema information and on evaluable predicates for expressing higher-order information. Prolog on the other hand while providing powerful features does so in only a procedural setting. Caught between a rock and hard place we ask whether a Horn clause language can be designed which provides most if not all of the power of Prolog in a declarative framework. In this paper we start with a simple logic programming language in which the central notion is that of an object and an expression. We build upon these simple constructs and show that the resulting language has the power of Datalog and a bottom-up semantics. We then successively increase the expressive power of the language to subsume LDL in the sense that we can support sets, stratified negation, and updates to base relations. Finally, we show that our language can support meta, schema and higher-order constructs in an integrated, consistent and clean framework.

1 Motivation

Let us learn how Io's frenzy came—
She telling her disasters manifold.
—ÆSCHYLUS, PROMETHEUS BOUND.

Prolog would have made a wonderful database language if all its power had been provided in a declarative framework. But precisely those very constructs which one finds so useful and powerful, e.g., negation, cut, updates etc., come only in a procedural setting. The pure subset of Prolog is uninteresting as a real programming language.

Many logic-oriented database languages are being proposed, e.g., LDL [13], Datalog, NAIL!, etc., with declarative semantics. These, however, are not as powerful as Prolog. In many cases they do not have cut, meta predicates, schema definition, updates, etc. The question is whether a Horn clause language can be designed which provides most if not all the powerful features of Prolog in a declarative setting and which is amenable to efficient compilation. Towards this goal, in [9] we proposed the addition of procedurality and updates while still retaining the declarative semantics of pure Horn clause languages. In [6] we proposed a feature called non-deterministic choice that encompasses the use of cut in Prolog and is shown to be even more powerful in a certain sense.

Continuing in this vein, in this paper we move a step closer to our goal by proposing a language in which the definition and manipulation of meta information and data is unified. Traditionally, a separate language is provided for this purpose. Our proposed language encompasses meta-data and data by allowing higher-order predicates to be defined in the language. Higher-order languages have been met with skepticism since, among other reasons, the unification problem is undecidable. Our solution is to follow conventional wisdom that led to the development of languages based on bottom-up semantics, summed up in the following statement from [12] "The important fact to be observed in bottom-up processing is that it has replaced full unification with matching (i.e., only one of the two terms contains variables)." The effect of replacing unification with matching is a gain in performance. This has been the foremost reason for the recent burst of interest in bottom-up processing of Horn clause queries. Using the same wisdom, we have asked ourselves whether there exists a higher-order language in which higher-order unification can be replaced by the matching operation. In this paper we present such a language and show its use in providing the capability of expressing and manipulating schema and higher-order information in a database system. Our language allows queries that can not be expressed in Prolog. Further, we show that it is capable of defining schema information as well as being able to use them in the program, e.g., updating all attributes of all relations that have DATE as their data type.

As a side benefit we have also been able to provide a standard treatment of sets. For example, languages such as LDL [4] and LPS [5] treat relations and sets differently. Such a duality is unnatural and as a consequence not all operations defined for sets are applicable to relations. By standardizing the treatment of sets and relations, we have not only simplified the expressions but also allow a richer collection of set selection expressions which also promise to help the set unification problem [11].

Our presentation in this paper is necessarily impressionistic. In particular, we have attempted to impart in a clear and consistent manner the essential ideas of the language. This has necessitated a consistent syntax in which we have liberally used disambiguating tokens. We readily recognize that it may not be user-friendly. We have provided sketches of formal semantics, both declarative and operational, for parts of our language and in some cases arguments which point how semantics can be defined. A formal paper on this subject is forthcoming.

The organisation of the paper is as follows. We start by defining a simple subset, called L^0, of our language. We show that L^0 is equivalent to Datalog and that it can be extended to L^1 which allows stratified negation, updates to base relations and set grouping. Moreover, L^1 treats sets and relations uniformly and allows a rich collection of set expressions. The higher-order counterpart of L^1 is called L^2 and we show how higher-order information is expressed and manipulated in L^2. The semantics of L^2 use higher-order matching. Finally, we show how L^2 supports and manipulates schema information.

2 L^0: A language of objects and expressions

An easy commerce of the old and the new
—T. S. ELIOT, LITTLE GIDDING

The conceptual structure of L^0 is based on objects and expressions on objects. An object can be classified into one of four categories: a set of objects, a tuple of objects, a functor object or an atomic object. Examples of atomic objects are integers, characters, etc.

A functor object is recursively defined as an object of the following form, where f is an n-ary functor symbol:
$f(object_1, \ldots, object_n)$

A tuple object is a sequence of named objects. Syntactically, a tuple of objects can be viewed as
$(attr_1 : object_1, \ldots, attr_k : object_k)$ in which, each $attr_j : object_j$ pair refers to the $object_j$ that is the $attr_j$ attribute of the tuple. We usually refer to $object_j$ to be the $attr_j$ object of the tuple and when there is no confusion, we informally refer to the $attr_j$ object as $attr_j$. For example $(name : john, sal : 10K)$ is a tuple object.

A set object is a named (not necessarily homogeneous) collection of objects. For example, $p\{(name : john, sal : 10K), \ldots\}$ is a set object named p whose elements are tuple objects.

Corresponding to each kind of object we have an expression of that type. An expression, evaluated on an object, under a substitution (defined below), returns true or false. An atomic expression (evaluated on an atomic object) returns true only for an atomic object; and, similarly for functor, tuple and set expressions. If an expression evaluates to true on an object then we say that the object *satisfies* the expression.

An atomic expression is an expression of the form αX where X is either a variable (denoted by a capitol letter) or a value (denoted by a lowercase letter or number) and $\alpha \in \{<, \leq, >, \geq, =, \neq\}$. For example, $>$ C and $\leq$ 5 are atomic expressions. The subset of atomic expressions ensuing from restricting α to the symbol = shall be called *simple atomic expressions*. In simple atomic expressions we shall often omit the equality sign. The simple atomic expression of the form $= X$ is also a valid tuple, functor and set expression.

A functor expression is an expression of the form $@f(exp_1, \ldots, exp_n)$ where $exp_j, j = 1, \ldots, n$ is an expression for the j^{th} argument for the n-ary functor f.

A tuple expression is a conjunct of the form
$.a_1exp_1, .a_2exp_2, \ldots, .a_kexp_k$
where each exp_j is an expression on the object associated with the a_j attribute of the tuple. Note that in general an attribute a_k may not be defined for a tuple

object but such an expression is still syntactically valid for the tuple object.

A set expression is defined to be
(exp)
in which exp is an expression on an object, i.e., element, of the set.

We also define the empty expression ϵ which satisfies all objects. In summary we show below the grammar for expressions of L^0 in which words starting with an uppercase letter denote non-terminals.

$Exp \rightarrow$	$Ae \mid Fe \mid Te \mid Se \mid = variable$
$Ae \rightarrow$	$Relop\ constant \mid Relop\ variable \mid \epsilon$
$Fe \rightarrow$	$@Fname(ExpL) \mid \epsilon$
$ExpL \rightarrow$	$Exp, ExpL \mid Exp$
$Te \rightarrow$	$.AnameExp, Te \mid .AnameExp \mid \epsilon$
$Se \rightarrow$	$(Exp) \mid \epsilon$
$Fname \rightarrow$	string constant
$Aname \rightarrow$	string constant
$Relop \rightarrow$	$> \mid \geq \mid \leq \mid < \mid = \mid \neq$

Constants and variables are defined, in the usual manner, as strings starting with lower and upper case letters, respectively.

2.1 Evaluation of expressions

A *substitution* is a non-empty finite set of ordered pairs $\{X_1/o_1, \ldots, X_n/o_n\}$ such that $(\forall 1 \leq i \leq n)$ X_i is a distinct variable, o_i is an object. We view a substitution as a mapping on variables that is the identity almost everywhere. If σ is a substitution and X a variable the result of *applying* σ to X is defined as

$$X\sigma = \begin{cases} o, & \text{if } (X/o) \in \sigma \\ X, & \text{otherwise} \end{cases}$$

We extend this mapping to expressions in a manner consistent with the above definition. If $\sigma = \{X_1/o_1, \ldots, X_n/o_n\}$ and e is an expression then
$e\sigma = [\lambda X_1 \ldots \lambda X_n.e]o_1 \ldots o_n$
Evidently the idea behind an application of σ to an expression e is to replace the free occurrences of $X_1, \ldots, X_n$ in e by the objects $o_1, \ldots, o_n$.

If θ and α are two substitutions and $\theta \cup \alpha$ is a substitution then we say that α and θ are *mutually consistent*; otherwise they are said to be *inconsistent*.

A ground (i.e., free of variables) atomic expression, say αc, evaluated on an atomic object, say o, returns true if the comparison $o\alpha c$ is true. An atomic expression, αc, evaluated on an atomic object o returns true, if there exists a substitution σ such that $o\alpha c\sigma$ is true. In such a case we shall say that the object o satisfies the expression. Any object, o, satisfies the expression $= X$ for the substitution X/o.

A functor object o with n-ary functor symbol f satisfies a functor expression $@f(exp_1, exp_2, \ldots, exp_n)$ if there exists a substitution σ such that $(\forall 1 \leq i \leq n) exp_i\sigma$ is satisfied by the object o_i in the i^{th} argument of o.

A tuple object $o = (b_1 : o_1, \ldots, b_k : o_k)$ satisfies a tuple expression
$.a_1exp_1, .a_2exp_2, \ldots, .a_nexp_n, n \leq k$
if there exists a substitution σ such that $(\forall 1 \leq i \leq n).a_iexp_i$ there exists $b_j : o_j \in o, (1 \leq j \leq k)$ such that $b_j = a_i$ and o_j satisfies exp_i under the substitution σ.

A set object s satisfies a set expression (exp) if and only if there exists a substitution σ and an element $o \in s$ such that o satisfies exp.

Example: Below we show examples of objects and expressions that satisfy them.

expression	*object*	substitution
$= X$	b	(X/b)
$.n > X, .m = Y$	$(n : 6, m : a)$	$(X/5, Y/a)$
$(.n > X, .m = Y)$	$\{(n : 6, m : a), (n : 5, m : b)\}$	$(X/5, Y/a)$
$@f(X, g(Y))$	$f(5, g(a))$	$(X/5, Y/a)$

Example: Consider a tuple object in which the emp attribute is a set of tuple objects, i.e.,
$(a_1 : n_1, \ldots, emp : \{e_1, e_2, \ldots\}, \ldots)$
where e_i is a tuple consisting of `name`, `age`, `salary` and `children` attributes. `Name`, `age` and `salary` objects are atomic whereas the children object is a set of names of children. Below we show some expressions on the emp object.

`.emp(.age=30)`
"Is there a 30-year old employee?"

`.emp(.age>30, .name=N)`
"List names of all employees older than 30."
Note that the ordering for the age and name attributes is immaterial because we are naming the attributes.

`.emp(.salary@£(=5000))` .
"Is there an employee with salary of £(5000)?"
This example also exemplifies the need for the *token* denoted by "@" before the functor symbol. Note that the use of the functor £, when juxtaposed with the attribute name `salary` without the token, would be ambiguous with a possible attribute name of `salary£`.

```
.emp(.age=30, .name=N),
  .emp(.name=N,.aged=50)
```

> "List all 30 year old employees who have the same name as a 50 year old employee."
>
> `.emp(.name=N, .children(john))`
> "List names of all employees with a child named john."

2.2 Rules and programs

Define a *database*, *db* to be a tuple object

$$db = (r_1 : s_1, r_2 : s_2, \ldots, r_n : s_n)$$

whose attributes are relations, i.e., a set of tuple objects.

We define a *rule* as an implication *head* $\leftarrow$ *body* in which *head* is an expression of the form $.p(exp)$ such that p is an attribute of db (also referred to as a derived predicate) and exp is a list of simple atomic expressions or objects and *body* is a conjunct of expressions on objects $e_1, \ldots, e_n$
$.e_1 exp_1, \ldots, .e_n exp_n$
Without loss of generality we shall restrict each e_i to be an attribute of the database object, db. Therefore, each exp_i is a tuple expression involving exactly one attribute of the database object (i.e., a single relation). We can thus view the body as a conjunct of expressions on the database object using the variables $X_1, \ldots, X_k$. We emphasize that we are initially restricting all expressions in rules to be defined over the database object.

We define a *query* to be a rule with an empty head and denote it as $?.e_1 exp_1, \ldots, .e_n exp_n$.

A note on notational convenience: It is more common to write rules without attribute names. We can allow this notational convenience by assuming that each predicate has an ordering of attributes in the tuples. For base predicates, such an ordering is declared in the schema; for derived predicates, left-to-right ordering in the rule definition may be assumed. As an example consider the emp relation of the previous example in which each tuple in `emp` consists of `name`, `age`, `salary` and `children`, in that order.

```
.sameName(N) ← .emp(N,30, —, —),
  .emp(N,50, —, —)
```
> "sameName is a set of 30 year old employees who have the same name as a 50 year old employee."
>
> `?.emp(N,—,—,.children(john))`
> "List names of all the employees having a child named john."

End of Note

A *program* is defined to be a triple (rdb, edb, q) where rdb is a finite collection of rules, edb is a finite set of (base) relations, and q is a query.

2.3 Semantics of L^0

We start by informally explaining the notion of the universe, U, of a program P of L^0. Initially we take U_0 to be the set of all atomic objects in P. In case there are none then U_0 is to consist of a special atomic object, say $\perp$. Next, U_1 is defined as the set of all possible tuple, functor and set objects that can be formed from the elements in U_0. Proceeding inductively in this manner we let U be the infinite union of all U_i.[1]

We now define the satisfaction of a rule. Consider a rule of the form

$$.p(\ldots) \leftarrow p_1, \ldots, p_m$$

Let σ be a substitution and $I \subseteq U$. The rule is satisfied if

> whenever each $p_i, i \in \{1, \ldots, m\}$ is satisfied by some object $o_i \in I$ under substitution σ then $.p(\ldots)\sigma$ is also satisfied by some object in I, or
>
> there is some p_i which is not satisfied by any object in I.

Note that a fact is trivially satisfied by the empty atomic object under the identity substitution.

Given a collection of rules rdb, $I \subseteq U$ is a model of rdb if I satisfies all the rules in rdb. A model M of a given collection of rules rdb is said to be minimal if no proper subset of M exists such that it is also a model of rdb.

[1]Formally, we may define U as follows. Let U_0 be the set consisting of all atomic objects. For $n > 0$ U_n is defined inductively as follows (let $\mathcal{P}(S)$ denote the set of subsets of set S):

$$\begin{aligned}
G_{m,n,0} &= U_{n-1} \cup \mathcal{P}(U_{n-1}) \\
G_{m,n,j} &= G_{m,n,j-1} \cup \{f(o_1, \ldots, o_k) \mid \\
&\quad f \text{ is a functor of arity of } k, \\
&\quad \text{and } o_i \in G_{m,n,j-1}, 1 \leq i \leq k\} \\
G_{m,n,j} &= G_{m,n-1,j} \cup \{\cup_k \{(o_1, \ldots, o_k) \mid \\
&\quad o_i \in G_{m,n-1,j}, 1 \leq i \leq k\}\}
\end{aligned}$$

$$U_m = \bigcup_{j=0}^{\infty} \bigcup_{n=0}^{\infty} G_{m,n,j}$$

$$U = \bigcup_{i=0}^{\infty} U_i$$

Proposition: A program L^0 has a unique minimal model. ∎

We define the meaning of a program to be the unique minimal model of the program as defined above.

Define an operator T, similar to [1], as follows: Given a set of rules *rdb* and a set I

$$\begin{aligned} T_{rdb}(I) = \{&(\overline{X})\sigma \in p \mid \\ &.p(\overline{X}) \leftarrow .p_1, \ldots, .p_n \text{ is a rule in rdb}, \\ &\exists\sigma\ satisfying\ .p_1, \ldots, .p_n\} \end{aligned}$$

Define powers of the operator T as follows:

$$\begin{aligned} &T \uparrow 0(I) = T(I) \\ &T \uparrow n(I) = T(T \uparrow n-1(I)) \cup T \uparrow n-1(I) \\ &\qquad (n \geq 1) \end{aligned}$$

We define the *least fixpoint* of the program (rdb, edb, q) to be the set of objects $T_{rdb} \uparrow \omega(\phi)$ where ω denotes the first ordinal number. Note that T is monotonic.

Proposition: $T_{rdb} \uparrow \omega(\phi)$ exists and is equivalent to the unique minimal model of a given program $P = (rdb, edb, q)$. ∎

We assume that the least fixpoint of a program gives the meaning of the program in the sense that the answer to a query ϕ with respect to the least fixpoint M of a program P is

$$\{(d_1, \ldots, d_k) \mid (\phi(X_1, \ldots, X_k)\sigma) \in M\}$$

where σ is the substitution $\{X_1/d_1, \ldots, X_k/d_k\}$.

Proposition: L^0 subsumes Datalog, i.e., if P is a Datalog program with unique minimal model M, then M is also a model of the L^0 program P. ∎

3 Extending L^0 to L^1

No one shall drive us from the paradise that Cantor has created.
—HILBERT

We extend L^0 to L^1 by generalising set expressions of L^0. These generalisations give the ability to negate set expressions, operate on or construct sets and insert or delete elements from sets.

$$\begin{aligned} &Sexp && \rightarrow Sign\ Sexpl \mid \neg Sexp \mid \langle Sexp \rangle \\ &Sexpl && \rightarrow (exp) \mid \{\} \mid \{ExpList\} \mid \epsilon \\ & && \rightarrow \{\ldots, ExpList\} \mid \{ExpList, \ldots\} \\ & && \rightarrow \{\ldots, ExpList, \ldots\} \\ &Sign && \rightarrow + \mid - \mid \epsilon \end{aligned}$$

3.1 Set Selection

Recently the problem of allowing sets in logic programming languages has received attention from [4, 5]. It is a consequence of the definition of L^0 that relations and sets are treated uniformly. Thus no special provisions have to be made to support sets. In fact, by generalizing set expressions we can allow subsets of sets to be selected. The need for such a selection mechanism has been felt for some time in LDL [11]. We shall allow sets with unknown cardinality, with a known cardinality, i.e., exactly an integer "k", and sets in which the cardinality is bounded by some integer, i.e., at most "k". It is this third case which allows the associativity problems to creep in and requires ACI unification which is semi-decidable[11].

The meaning of set selection expressions is as below:

{} denotes the empty set.

{X,Y,Z} denotes a set of cardinality 3.

{(X,Y,Z)} denotes a singleton set.

{X,Y,Z,...} denotes a set of cardinality $\geq$ 3.

{...,X,Y,Z} denotes a set of cardinality $\leq$ 3.

{...,X,Y,Z ...} denotes a set of arbitrary cardinality.

Note that $\{X, \ldots\}$ as defined above is equivalent to (X) in L^0.

Example: Consider as before the tuple object db which has a set object family(Mother, Children) as an attribute; the elements of `family` are as follows:
$\{(mary, \{bill, jack\}), (jill, \{peter, paul, mary\}), (nancy, \{\})\}$.

Here are some sample queries:

`?.family{—,=Kids}`
"List all sets of children"
Answer: Kids/ {bill,jack},
Kids/ {peter,paul,mary}, Kids/ {}

`?.family{=mary,{=bill}}`
"Does Mary have a single child"
Answer: no.

`?.family{=X,{...,=Y,=Z}}`
"Who has at most two children"
Answer: X/mary, X/nancy and commutative bindings for Y and Z

`?.family{=X,{=Y,=Z,=W}}`
"Who has exactly three children"
Answer: (X/jill, Y/peter, Z/paul, W/mary) where we have omitted other answers due to commutativity of the set elements.

```
?.family{=X,{=Y,=Z,...}}
```
"Who has more than two children"
Answer: X/mary, X/jill with commutative bindings for Y and Z.

Example: Constructing enumerated sets.

```
.p(X,Y,Z) ← .book(X,<30),
  .book(Y,<30), .book(Z,<30)
```
"Collect sets of three books individually costing less than $30 each."
```
.bookDeal{{X,Y,Z}} ← .book(X,Px),
  .book(Y,Py),.book(Z,Pz),
  Px+Py+Pz ≤ $100
```

We are assuming the existence of evaluable predicates and functions such as +. Note that here we are asking for exactly three books whose cost is at most $100. We can use the argument {...,X,Y,Z} to ask for at most three books, etc. [4].

3.2 Grouping and Negation

In this section we first consider the generalisation of set terms to allow elements to be grouped into sets of arbitrary cardinality and then to allow negation. A grouping rule is a rule of the form

$$.p(\overline{X}, setname\langle\overline{Y}\rangle) \leftarrow body(\overline{X}, \overline{Y})$$

Readers are referred to [4] for a formal description of the semantics of grouping rules. Informally, the operational semantics of such a rule are as follows. The body of the rule is satisfied by a set of substitutions inducing the construction of a relation, called the *body relation*, and for each value $\overline{x}$ of the variables $\overline{X}$ we collect all the Y-values $y_1, \ldots, y_n$ into the set object named **setname**.

Example: Set Grouping. Group all parts by supplier.
$.p(Supplier, parts\langle Part\rangle) \leftarrow$
$.b(Supplier, Part)$

Next we consider the generalisation of set expressions to allow negation. We propose to use stratified negation [8] as in LDL and Datalog$^{\neg}$ with a simple syntactic change. Observe that in LDL a negation is used only in conjunction with a predicate which was defined to be a set of tuples. Recalling our philosophy, that a predicate is a set of tuples, we conclude that the negation is defined only in conjunction with a set. The syntax used in LDL is to prefix the negation symbol $\neg$ to the predicate expression to get the negated expression (i.e.,$\neg emp(...)$) whereas we use $emp\neg(exp)$. We illustrate the usage by the following example.

Example: Consider the following query on the emp relation.

$?.emp(.name = N1, .salary = S1),$
$.emp\neg(.salary\rangle S1, .name \neq N1)$

```
Find the employee with a salary such
that no other employee has a larger
salary.''
```
[2]

We emphasise that the semantics of negation in L^1 are identical to that of stratified negation in LDL and Datalog.

3.3 Semantics of L^1

We define a notion of an *admissible* L^1 program (rdb, edb, q) as follows. We start by defining a preference relation between attributes in rules of *rdb*. If there is a rule of the form

$$.p(\ldots) \leftarrow \ldots, .q(\ldots), \ldots$$

then we say that $p \geq q$. If the rule is of the form

$$.p(\ldots) \leftarrow \ldots, .q\neg(\ldots), \ldots$$

or it is a grouping rule of the form

$$.p(\ldots, \langle\ldots\rangle) \leftarrow \ldots, .q(\ldots), \ldots$$

then we say that $p > q$.

We shall say that a program is *admissible* if there does not exist a sequence of attributes

$$.p_1\theta_1.p_2\ldots\theta_{k-1}.p_k\theta_k$$

such that $\theta \in \{>, \geq\}$ and $.p_1 = .p_k$ and there is some θ_j is $>$, $(\forall 1 \leq j \leq k)$.

As in Datalog and LDL the admissibility requirement induces a partitioning on the rules of *rdb* and we may write *rdb* as composed of the disjoint sets of rules $L_0, L_1, \ldots, L_n$.

Given a program (rdb, edb, q) we define the notion of a standard model of a program as follows. Let $L_0, \ldots, L_n$ be a partitioning of $rdb \cup edb$. Then

$$M_0 = T_{L_0} \uparrow \omega(\phi)$$
$$M_1 = T_{L_1} \uparrow \omega(M_0)$$
$$\cdots M_n = T_{L_n} \uparrow \omega(M_{n-1})$$

[2]It is instructive to note that the equivalent Datalog$^{\neg}$ query
$?\ emp(N1, S1), \neg emp(N2, S2), S2\rangle S1, N1 \neq N2.$
is unsafe.

We define the meaning of an admissible L^1 program to be the set M_n. As in LDL and Datalog it can be shown that M_n is a minimal model of the corresponding program.

Proposition: L^1 subsumes Datalog$^{\neg}$ and update-free LDL. ∎

3.4 Insertion and deletion in set expressions

The final generalization we consider of set expressions is that to allow elements to be inserted and deleted from set objects. LDL has the capability to insert and delete tuples from a *base relation*. For example the LDL rules

$$\ldots \leftarrow \ldots, +b(X,Y)$$

$$\ldots \leftarrow \ldots, -b(X,Y)$$

have the meaning that the substitution for X and Y that result from the body are inserted and deleted in the '+' and '−' cases respectively. We propose a syntactic change to denote the insertion (resp. deletion) of tuple from the set object b by `b+(:X,:Y)` (resp. `b-(:X,:Y)`). We empahsise that this being a syntactic change, the semantics presented in [9] carry over to this case. In particular, note that the basic maxim in update semantics in LDL was that a query provides bindings for tuples that are to be inserted or deleted, i.e., first compute the set of tuples for an update and then do the update.[3] This maxim and its concommitant update semantics can be carried over into L^1 without change. For the sake of completeness, we present below some examples of updates to base relations, so as to make the presentation self-contained for readers unfamiliar with the update proposal in LDL.

Example [9]:

Fire all managers who make more than their employees.

```
.fireEmp() ←
  .emp(N,X,S1), .mgr(N,M),
  .emp(M,Y,<S1), .emp−(N,M,S1).
```

Note that the set of employees to be deleted is determined by the subgoals preceding `.emp−(...)`.

Give every database employee a 10 percent raise.

```
?.eds(X,db,S), S1=S*1.1,
  eds−(X,db,S), eds+(X,db,S1).
```

[3]The same update maxim was used in Query-by-Example (QBE).

Further note that the procedural constructs presented in LDL [9] may be assumed without any syntactic change. Also the declarative and operational semantics defined for these procedural constructs may be used with minor changes in L^1. As none of our examples in this paper use procedural constructs we leave this out from our presentation.

Proposition: L^1 subsumes LDL with updates. ∎

4 L^2: A higher-order language

Two worlds become much like each other
—T. S. ELIOT, LITTLE GIDDING

In L^1 we restricted attribute names and functor symbols to be values. In this section we remove this restriction and define a notion of higher-order quantification over attribute names and functor symbols. The resulting language is rich enough to express and query meta and schema information. Essentially we generalise L^1 expressions as follows:

$$\begin{array}{ll} \textit{Fname} & \rightarrow \textit{constant} \mid \textit{variable} \\ \textit{Aname} & \rightarrow \textit{constant} \mid \textit{variable} \end{array}$$

Thus, we re-define the notions of functor and tuple expressions A functor expression is an expression of the form $@F(exp_1, \ldots, exp_n)$ where F is either a variable ranging over functor symbols or a functor symbol itself. Further, $exp_j, j = 1, \ldots, n$ is an expression for the j^{th} argument for the n-ary functor reference F.

Similarly, a tuple expression is a conjunct of elementary expressions as defined below:

$$.A_1exp_1, .A_2exp_2, \ldots, .A_kexp_k$$

where each A_i, $i = 1, \ldots, k$, is either a variable for an attribute name or an attribute name itself. Further, each exp_j is an expression on the object associated with the A_j attribute of the tuple.

A variable occurring in a functor symbol or an attribute position in an expression will be referred to as a *higher-order variable*. We define a *higher-order expression* as an expression as defined before, using the new definition for functor and tuple expression and use the word expression to mean possibly a higher-order expression from now on. Informally, the evaluation of an expression is as before, except that now a substitution must also give bindings to the variables standing for attribute names and functor symbols, i.e, a *substitution* is now defined to be a non-empty finite set of ordered pairs $\{X_1/o_1, \ldots, X_n/o_n\}$ such that $(\forall 1 \leq i \leq n)$ X_i is a distinct variable ranging over not only the objects in the universe (as defined earlier) but also over functor and attribute names.

A rule is defined as an expression of the form

$$db.p(exp) \leftarrow e_1exp_1, \ldots, e_nexp_n.$$

where each e_i in the body is now allowed to range over the objects in the universe and each exp_i is a higher-order expression. The body is viewed as a conjunct of expressions on the objects of the universe using, as before, the variables $X_1, \ldots, X_k$. Since the objects usually range over the database, we still omit the usage of db in rules; for all objects other than the database, db, we shall be explicit in referring to that object. Observe that the head predicate is still restricted to be a name in the database object (cf. section 2.3) and not a variable and exp is a list of simple atomic expressions or objects. Thus, all variables in the head are still required to be referenced in the body. As a consequence, the set of tuples defining the predicate p is still defined by the substitutions for the variables $X_1, \ldots, X_k$. Needless to say that the definition of a program remains unchanged.

Declaratively, the satisfaction of a rule r with higher-order database expressions by a collection of objects I can be understood as the question: Does there exist a higher-order substitution σ such that $I \models r\sigma$? Note that under this interpretation and the notion of a higher-order substitution the declarative model of L^2 programs can be defined in a manner similar to the method employed for L^1 programs.[4] We leave this for a formal and fuller presentation.

We discuss the operational model in the following two subsections by first presenting some illuminating examples to impart the essential ideas underlying the operational semantics. First, in the next subsection, we present the language with the restriction that the e_i's in the rules are still restricted to only the database object in the universe (as before), except that the allowed expressions are higher-order expressions. We refer to this class of expressions as higher-order database expressions. In a later section we relax this restriction and allow all objects in the universe.

4.1 Higher-order queries on the database object

Consider the rule: $.p(X) \leftarrow .X$ It bears repetition that the above rule is equivalent to

$$.p(X) \leftarrow db.X$$

[4]The main difference would be in the definition of the universe U.

in our notation as was declared earlier and we have, for notational convenience, been omitting the prefix "db". We shall continue to do so but the readers are reminded to keep this in mind. The predicate p computes the names of all the predicates (both derived as well as base) in the database, db. Informally, the substitution for X ranges over all the predicates. For each substitution, the corresponding tuple in p is added. As the range of X is limited to the predicates in the database object, (i.e., for all other objects of the universe the expression, .X, is false) the computation of the above rule computes the meaning of the rule as per $T \uparrow \omega$.

Operationally, consider a rule with higher-order variables, $X_1, \ldots, X_k$, (i.e., variables referring to a database attribute position). Rewrite the rule for each possible substitution for these variables. The rewritten rules are all rules in L^1 and their meaning is defined as before, as long as the number of substitutions for the variables is finite. The substitutions for the variables have to be finite because, the database predicates have finite attributes. In short, the extension of expressions to higher-order is limited by the range of attributes in the database and thus makes the language decidable. We refer to this process of associating a meaning as *replacement semantics*. Let us consider some more examples.

Example: Consider the database consisting of three relations: systems, ai and hi, each representing the employees in the respective departments. The tuples in all three relations contain `name`, `tel` (i.e., telephone no.), and `salary`.

> `.p(X) ← .Y(.name=X)`
> The predicate p defines all the names of the employees in `systems`, `ai` and `hi` relations.
>
> `.p(X,Y) ←.Y(.name=X)`
> A query, `?.p(john, Y)`, corresponding to the predicate p, computes john's department.
>
> `.p(X,Z) ←.X(.name=Z, .tel=T),`
> `.Y(.name≠ Z, .tel=T)`
> The predicate p computes the names of all employees (and their department name) who share their telephone.
>
> `.p(X,Z) ←.X(.name=Z, .tel=T),`
> `.Y(.name≠ Z, .tel=T), Y≠ X`
> The predicate p computes the names of employees (and the relation name) who share their telephone with an employee in another department.■

In summary, any rule involving higher-order database expressions with variables ranging over attribute names can be associated with a meaning using the above replacement semantics. The approach we have taken for these queries parallels the approach taken in Office-by-Example (OBE) [14] [10], where such higher-order queries were defined using a similar domain calculus but in a very limited context (e.g., OBE did not have recursion,complex objects, etc.). Interestingly, these rules cannot be stated in this manner in Prolog. The notion of quantifying over database predicates was not attempted in Prolog (however, the notion of meta predicates that are allowed in Prolog is addressed in the next subsection).

4.2 Higher-order queries on objects in the universe

Consider the example of the database containing the relation `family(mother,children)` from section 3.1 and the rule

$$.p(X,Y) \leftarrow .family(X,Y), Y\{= bill \ldots\}$$

defining the predicate p to contain all mothers (and the set of her children) having a child named `bill`. Note that the variable Y refers to a set that need not be an attribute of the database. The expression states that the variable Y is substituted for a set that has an element `bill` in it. This expression in the body of the rule is not a higher-order database expression as discussed in the previous subsection. (In the sequel we shall refer to such expressions as higher-order conditions). But the same operational semantics carry over to this case because the values that can be substituted for Y are limited by the finite values for Y in the database. Thus replacement semantics will provide the meaning for this program also.

On the other hand consider the following example:

> `.member(P,X)` $\leftarrow$ `P(X)`
> This defines a member predicate, that tests the membership of X in the set P.

Note that in the above rule, any value from the universe can be substituted for P. Therefore, the number of substitutions is unbounded. As a result, we cannot give replacement semantics to this rule. On the other hand, if the query `?.member(s, jill)` (where `s` is a particular set object) is asked then replacement semantics can be applied to this rule for this query. In short, we observe that such queries have a meaning if proper bindings are supplied from the head; otherwise, they are deemed unsafe.[5] Thus, we can attribute an operational meaning to a rule for a given binding for the attributes of the predicate. A *binding* for a predicate is the bound or unbound pattern of its attributes for which the predicate is to be computed.

[5]Note that unsafe queries in LDL do not have operational semantics.

We give below more examples of such usage.

Example:

> `.intersection(P,Q,`$\langle X \rangle$`)` $\leftarrow$ `P(X),Q(X)`.
> This defines the third attribute of the intersection predicate to be the intersection of the sets, P and Q. Note that the safety requires that P and Q be bound.
>
> `.subset(Q,P)` $\leftarrow$`.intersection(P,Q,P)`.
> P is a subset of Q. From the definition of the intersection predicate we can infer that the subset predicate also requires both P and Q to be bound.
>
> `.diff(P,Q,`$\langle X \rangle$`)` $\leftarrow$
> `.member(P,X),.member`$\neg$`(Q,X)`
> Set difference is defined using negation.
>
> $.r(P, -, \langle X \rangle) \leftarrow P(X)$.
> $.r(-, Q, \langle X \rangle) \leftarrow Q(X)$.
> $.union(P,Q,\langle Y \rangle) \leftarrow .r(P,Q,\{\ldots,Y,\ldots\})$.
> Union is expressed through multiple rules with the same head predicate symbol. ∎

Notice that the lack of such higher-order variables in LDL (and other languages using bottom-up semantics), requires the use of special evaluable predicates for set operations.

The above examples show the need for characterizing admissible rules using higher-order variables as follows. A rule base *rdb* is said to be *ordered* if each rule is associated with an ordering of the conjuncts in the body. An ordered *rdb* is said to be *covered* for higher-order variables with respect to a given query, if the following condition is recursively true. For each rule defining the query, all the higher-order variables are *rule-covered* and the same ordered *rdb* is covered for each predicate occurring in the body (with the implied binding).

Given a rule r in an ordered *rdb* of the form

$$p(\ldots) \leftarrow exp_1, \ldots, exp_n$$

and a binding b for p, the higher-order variables in r are said to be *rule-covered* if for each higher-order variable occurring in exp_i in an attribute position the same variable occurs in some $exp_j, j = 1, \ldots, i-1$ or in a bound argument position of p.

Define $p \leq q$ if there exists a rule of the form

$$.p(\ldots) \leftarrow \ldots, q(\ldots), \ldots$$

and p < q if there exists a rule of the form

$$.p(\ldots) \leftarrow \ldots, q(\ldots, X, \ldots), \ldots$$

where X is a higher-order variable. We say that an ordered *rdb* is *stratifiable* with respect to higher-order variables if there does not exist a sequence of the predicate symbols of *rdb*

$$.p_1\theta_1.p_2\ldots\theta_{k-1}.p_k\theta_k$$

such that $\theta \in \{<, \leq\}$ and $.p_1 = .p_k$ and there is some θ_j is $<$ $(\forall 1 \leq j \leq k)$.

We define a set of rules *rdb* to be *admissible with respect to a query q* if there exists an ordering for *rdb* such that the ordered *rdb* is covered and *rdb* is stratifiable with respect to higher-order variables.

Informally, the admissibility condition has the following effect. The substitutions for every higher-order variable X can be computed independently of the higher-order condition that uses X.

The admissibility condition on *rdb* ensures that replacement semantics will associate only a finite number of objects with higher-order variables. Consequently, replacement semantics will give an L^1 program to which the semantics of section 3 can be attributed.

Observe that all the examples shown above satisfy the admissibility condition. However, this condition is too restrictive as the following example shows. Pump applies an operation to members of a set [2].

Example:

```
.pump(X,Result,Operation,Identity) ←
    .partition(X,X1,X2),X{Y...}
    ,.pump(X1,R1,Operation,Identity)
    ,.pump(X2,R2,Operation,Identity)
    ,.Operation(R1,R2,Result).
.pump({},Result,Operation,Identity)
  ← Result = Identity.
```

Consider the query `?.pump(X,S,sum,0)` (where `sum` is an evaluable predicate which adds two integers). Note that this query gives a single binding for the higher-order variables `Operation`, and `Identity`. Consequently, replacement semantics correctly provides an L^1 program even though the rules are inadmissible! In fact .pump will compute the result of any commutative and associative operation with an identity. ∎

The above example shows that the stratification condition is too restrictive. We formulate a less restrictive strategy as follows. Consider a rule r which contains one higher-order condition. Let the higher-order variable be X. Compute, disregarding the higher-order condition, the set of all substitutions for X. Note that if this set is finite then replacement semantics will work. The finiteness of substitutions for X can be viewed as the traditional problem of safety in logic programs. In [7] an algorithm employing a sufficient test has been proposed using which the finiteness property can be checked. It is possible to extend this strategy to rules with more than one higher-order condition but for the sake of brevity we leave this to a fuller presentation.

5 Adding a schema facility

The complete consort dancing together
—T. S. ELIOT, LITTLE GIDDING

As mentioned before, most query languages (if not all) provide a separate set of commands for defining the schema of the database. Furthermore, this schema information could not be used to compose a condition on the database. For example, list attributes (in all predicates of the database) that have the data type of DATE and change the values in all these attributes from month/day/year format to day/month/year format. In this section, we present an extension of the language to allow such queries and updates.

We have already observed that the query to list all the predicate names in the database(i.e., ?.X) uses the structure of the database to pose the query. In order to pose a query using schema information, we extend the language to refer to the meta information of the data. Furthermore, using the same syntax, we show that meta information can be defined, thusly, providing a systematic framework for schema definition.

We re-define the notion of an expression to include meta-information as follows. The meta information for any object is a tuple object. Examples of attributes that may be defined in this tuple are *type*, *key*, *cardinality*, etc. Obviously, the list of attributes in this tuple determine the expressive power of the language. As the language provides the capability to define any number of different attributes each of which is an object of arbitrary complexity, we expect the expressive power to be unlimited. In the examples of this section, we shall use only the *type* attribute and also assume only the values, *string*, *integer*, *set*, *tuple*, and *atom*[6] for this attribute.

Observe that any condition on meta information can be expressed as a tuple expression. Thus, we define a *meta-expression*, mexp to be `[texp]exp`, in which the `texp`, enclosed by square brackets, is a tuple expression on the meta information and exp is the expression

[6]Atom is one of the atomic types; e.g., integer, string etc.

on the object. Thus, we have defined meta-expression using the notion of an expression. We re-define an expression on an object, exactly as before, but recursively on meta-expressions.

A meta-expression is said to be satisfied on an object if the meta-information satisfies texp and the object satisfies exp. Needless to say that the evaluation of an expression and the resulting substitutions are extrapolated in the obvious manner. We leave unchanged the definitions of rules, programs, etc.

We also allow the use of + (resp. -) prefixed to an attribute in a tuple expression. This represents the insertion (resp. deletion) of the attribute into the schema for that tuple. Observe the difference between the following two cases:

```
?-.p(X)              ?.p-(X)
```

In the first case the predicate (assuming that it is part of *edb*) is to be deleted from the database, whereas in the second case, all the tuples are to be removed from the relation p.

In summary, we have proposed the following changes to the productions for Exp and Texp:

$$
\begin{array}{ll}
Exp \rightarrow & Exp1 \mid [Texp]Exp1 \\
Exp1 \rightarrow & Aexp \mid Fexp \mid Texp \mid Sexp \\
Texp \rightarrow & Sign\ .Aname\ Exp, Texp \\
 & \mid Sign.AnameExp \mid \epsilon
\end{array}
$$

We use this language for schema definition as well as in posing a queries and updates. This is exemplified in the following two examples. Note that the proposal here attempts to give a sampling of the capabilities of this language as opposed to describing it completely. As mentioned before, the comprehensiveness will depend on the meta-information as well as the capability of the system to use this information.

Example: Insertion or deletion of relations, attributes and types.

```
?+.r[.type=set]{(.name[.type=string],
    .age[.type=integer])...}
```

This defines a predicate *r* in the *edb* to be a set of tuples, each of which contains two attributes, *name* and *age*, which are defined to be of *string* and *integer* type respectively. Similarly, deletion can be done using the "–" operator.

```
?.r{+.salary[.type=integer,
    .default=0] ...}
```

This adds a new attribute, *salary* of type integer to the base predicate *r* and the default value for that attribute to be zero.

```
?.r-(.salary[-.type=integer,
    .default=0]=X, .name=N),
    .i2r(X,Y),.r+(.salary[+.type=real,
     .default=0]=Y, .name=N)
```

This changes the attribute, *salary* in the base predicate *r*, to be of type *real* from type *integer*. Further, all the values are also mapped through an evaluable predicate, *i2r* which coerces integers to reals. ■

Example: Usage of Schema Information.

```
.p1(Y) ←.X(.Y)
.p2(Y) ←.X(.Y[.type=integer])
.p3(Y) ←.X(.Y[.type=integer]),
   Y=[s| Y1]
```

Observe that we are using the Prolog notation of concatenation in this example. An appropriate definition of this evaluable function is assumed here. This example is similar to the predicate to list all the predicate names. The first rule computes all the attribute names in the database. The second rule lists only those attributes that are of type integer. Of those attributes, p3 computes only those that start with s.

```
.p1(Y) ←.X(.Y{...,X,...})
.p2(Y) ←.X(.Y[.type=set])
```

Predicate p1 computes all attribute names in all the predicates of the database that are sets and have at least one set with zero or more elements. In contrast, p2 computes all attribute names in the predicates of the database that are defined to be sets, irrespective of any value associated with that attribute. ■

Note that we have shown that any query (or update) can be used in conjunction with a condition on the meta information. This uniformity in the treatment of the schema information is achieved through the use of the higher-order predicates, as well as the notion of updating these higher-order predicates.

6 Conclusion

And all shall be well
All manner of thing shall be well
—T. S. ELIOT, LITTLE GIDDING

Over the last few months we have been engaged in the exercise of designing a language which has powerful features for deductive databases; and, whose semantics are declaratively specified. In this paper we have concentrated on assimilating higher-order information into a deductive database framework.

The central idea of this paper is that higher-order unification can be replaced by higher-order matching over a finite set of values. We have shown that this claim can be consistently upheld over a variety of powerful features and operations.

Acknowledgements

We thank Hassan Aït-Kaci for setting us straight on many issues and pointing us in right directions and Pat Lincoln for valuable discussions. The set notation we have used stems from a proposal by Carlo Zaniolo. We hope that these friends will continue to help us because "It ain't over till the fat lady sings ..."

References

[1] Apt, K., Emden, M.: Contributions to the Theory of Logic Programming, **J. of the ACM, 29(3)**, 1982.

[2] Bancilhon, F., Briggs, T., Khoshafian, S., Valduriez, P.: FAD, A Simple and Powerful Database Language, *Proc. VLDB*, Brighton, 1987.

[3] Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.: Magic Sets and Other Strange Ways to Implement Logic Programs, *ACM Sym. on PODS*, Boston, 1986.

[4] Beeri, C., Naqvi, S., Shmueli, O. and Tsur, S.: Sets and Negation in a Logic Database Language, *ACM Sym. on PODS*, 1987.

[5] Kuper, G.: Logic Programming with Sets, *ACM Sym. on PODS*, 1987.

[6] Krishnamurthy, R., Naqvi, S.: Non-Deterministic Choice in Datalog, *3rd International Conf. on Data and Knowledge Bases*, Jerusalem, 1988.

[7] Krishnamurthy, R., Ramakrishnan, R., Shmueli, O.: A Framework for Testing Safety and Effective Computability of Extended Datalog, *Proc. SIGMOD*, Chicago, 1988.

[8] Naqvi, S.: A Logic for Negation in Database Systems, *MCC Technical Report* and *Proc. of workshop on Deductive Databases*, Washington, 1986.

[9] Naqvi, S., and Krishnamurthy, R.: Database Updates in Logic Programming, *ACM Sym. on PODS*, 1988.

[10] Whang, K., et al.: Office-by-Example: An Integrated Office System and Database Manager, **ACM Trans. on Office Info. S.**, 1987.

[11] Shmueli, O., Tsur, S., Zaniolo, C.: Rewriting of Rules Containing Set Terms in a Logic Data Language (LDL), *ACM Sym. on PODS*, 1988.

[12] Sacca, D., Zaniolo, C.: Implementation of Recursive Queries for a Data Language Based on Pure Horn Logic, *International Conf. on Logic Programming*, Melbourne, 1987.

[13] Tsur, S., Zaniolo, C.: LDL: A Logic Based Database langauge, *Proc. VLDB*, 1986, Tokyo.

[14] Zloof, M.: Office-by-Example: A Business Language that Unifies Data and Word Processing and Electronic Mail, **IBM Systems Journal 21(3)**, 1982.