Language Features for Interoperability of Databases with Schematic Discrepancies

Ravi Krishnamurthy Witold Litwin* William Kent
H.P. Labs, Mailstop 3U-4, P.O. Box 10490, Palo Alto, 94303-0969

Abstract

Present relational language capabilities are insufficient to provide interoperability of databases even if they are all relational. In particular, unified multidatabase view definitions cannot reconcile schematic discrepancies, where data in one database correspond to metadata of another. We claim that following new features are necessary:

1. Higher order expressions where variables can range over data and metadata, including database names.
2. Higher order (multidatabase) view definitions, where the number of relations or of attributes defined, is dependent on the state of the database(s).
3. Complete view updatability for the users of multidatabase views.

We propose these features in the context of a Horn clause based language, called Interoperable Database Language, (IDL).

1 Introduction

While databases were traditionally called heterogeneous when they had different data models, semantic heterogeneity exists even if all the databases follow a common model and language [NSF]. Traditionally observed aspects of semantic heterogeneity[K89] include heterogeneous values, data representations, names and decompositions (e.g., different normalizations in relational databases). A less addressed problem is that of schematic discrepancies (SDs) (see examples 21-23 in [K89]), when one database's data (values) correspond to metadata (schema elements) in others. Schematic discrepancies will be frequent, as exemplified by the following abstraction of a real life application.

Example: Consider three stock databases. All contain the closing price for each day of each stock in the stock market. The schemata for the three databases are as follows:

database euter[1]**:**
relation r : {(*date*, *stkCode*, *clsPrice*) ...}

database chwab[1]**:**
relation r : {(*date*, *stk1*, *stk2*, ...) ...}

database ource[1]**:**
relation stk1 : {(*date*, *clsPrice*) ...},
relation stk2 : {(*date*, *clsPrice*) ...},
⋮

The *euter* database consists of a single relation that has a tuple per day per stock with its closing price. The *chwab* database also has a single relation, but with one attribute per stock, and one tuple per day, where the value of the attribute is the closing price of the stock. The *ource* database has, in contrast, one relation per stock that has a tuple per day with its closing price[2]. For now we consider that the *stkCode* values in **euter** are the names of the attributes and relations in the other databases (e.g., *stk1*, *stk2*). This assumption will be relaxed in Section 6 when explicit name mappings are introduced. ■

These schematically disparate databases have similar purposes although they may deal with different stocks, dates, or closing prices. A user of one of these databases may need to work with the other databases also. Our goal is to provide the interoperability of such disparate databases. Typical needs of a multidatabase user, are:

- to formulate queries with the same intention to each database, using the same formal expression in spite of schematic discrepancies; e.g. all stocks (in all three databases) that closed above $200.
- to formulate queries spanning over several databases; e.g., all stocks that are quoted in all the three databases, for the same day.

*Visiting H.P Labs and Stanford University

[1] Any similarity of names of the database (i.e., (R)euter, (S)chwab, and (S)ource) to popularly known names is purely coincidental.

[2] This *ource* schema may seem contrived to a database researcher but lo and behold, this is a popular schema among stock market data vendors.

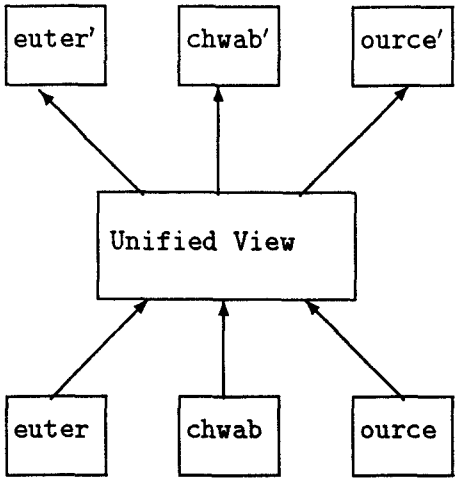

Figure 1: Two level mapping

- to formulate queries about the databases and the information they contain; e.g., list the stocks in `ource` and `chwab` that have the same closing price (which means to list the relation names and attribute names satisfying the join on attribute values).
- to be provided with a unified view of all the databases (*(multiple) database transparency*).
- to see all the databases as if they had the schema (e.g., `chwab`) the user knew before the integration or similar to this schema (e.g., with additional attributes) (*integration transparency*).
- to be able to update all the databases through the individual views or the unified view. (*multidatabase view updatability*).

The relationship between individual views and the unified views is shown at Figure 1 for stock databases. The *unified (multidatabase) view* `U` is defined over all the databases, in general, $D_i, i = 1, \ldots, n$. The customized views $D_i', i = 1, \ldots, n$, are defined using `U` to preserve integration transparency according to original databases $D_i's$.

The above new needs require the capabilities to quantify over data and metadata using higher order variables, and to define higher order multidatabase views. Our goal is therefore to cull the necessary language features. The relational languages, including the SQL dialects of the database industry, cannot suffice as they are first order languages. This is also the case of Horn clause languages such as $\mathcal{LDL}$, Datalog and Prolog. The language for interoperability has to be a higher order language. Even though there were higher order languages proposed in the literature [AG88, KN88, CKW89], none of them addresses the interoperability needs.

In what follows, we propose a language called IDL (read as *ideal*) for *Interoperable Database Language*. IDL provides the higher order capabilities by extending Horn clauses for higher-order logic. IDL is an extension of the language in [KN88] that subsumes $\mathcal{LDL}$, and Datalog, as well as SQL and other relational languages. Its interoperability features subsumes also those of MSQL[Li89]. IDL is nevertheless defined only to demonstrate the new capabilities and not per se to present another language. We define IDL's syntax and semantics and show that it fulfills the interoperability needs through a series of examples. This research is being done in the context of Pegasus project in H.P. Labs.

Section 2 discusses the multidatabase language requirements more in depth. We show the need for higher order query, higher order views and view updatability. Section 3 presents the overview of IDL. Section 4 deals with higher order queries. Section 5 deals with update expressions. Section 6 describes higher order views in IDL. The update through customized views is the topic of section 7. Section 8 concludes the paper.

2 Language Requirements

In order to handle SDs, it is necessary to model each database as a complex object wherein the metadata (i.e., relation names, attribute names, keys, types, etc.) and data are explicitly represented and can be used in the reasoning necessary for reconciling heterogeneity. In this paper, we concentrate on the relation names and attribute names only. It is easy to extend this to other metadata such as keys, types, authorization, etc. One such extension was given in [KN88] which is the basis of the language proposed here.

Higher Order Queries

Consider an `euter` user posing the following queries to all the databases.

```
1) Did any stock ever close above $200?
2) For each day, list the stock with
the highest closing price.
```

Both queries, against `chwab/ource` databases, cannot be posed as a relational query. They require higher order quantification (i.e., iteration over relation/attribute names) and the expressions requiring the join of the domains of these higher order variables. The language feature that is needed is higher order (query) expressions, which are the topic of section 4.

The higher order expressions are used to define a unified view over multiple databases. The view definition is the mapping from the individual databases to the unified view. Such a view definition requires the expression defining the view to allow variables over relation names and attribute names, for the same reasons observed before. Therefore, the use of higher order expressions in the unified view definition is considered necessary.

Higher Order Views

This integration transparency poses a new requirement to the language. Consider the customized view for the

ource users. .The number of relations in the ource view of the unified database will be data dependent; i.e., there will be as many relations in the ource view as there are stocks in the three databases. This necessitates the view definition to be capable of defining a varying number of relations, where the number is data dependent. Traditional view definition facility, in a relational language, allows the specification of a fixed set of named relations for all states of the database. In contrast, the language construct needed for view definition in a multidatabase language is one that is capable of defining a variable set of relations each with its own name, the number of relations and their names being data dependent. This language feature is termed higher order views and is discussed in section 6.

View Updatability

Given that the users are accessing the databases through their individual views then the ability to update via a given view becomes a necessity. View update requires the translation of the update (to the view) into a series of updates to the databases such that the subsequent computation of the view faithfully reflects the view update.[BS81, DB82, K85] The crux of the problem is that the translation is not unique as seen in the update of the following view defined using Datalog syntax.

$$\texttt{empMgr(Name, Mgr)} \leftarrow \texttt{emp(Name, Dno)}, \texttt{dept(Dno, Mgr)}.$$

This view computes the manager of a given employee by joining the **emp** relation with **dept** relation. In order to update the manager of a given employee through this view, either the Dno of the employee can be changed or the Mgr in the **dept** relation can be changed. Either update to the base relation will have the net effect of changing this employee's manager in the view.

In order to resolve this ambiguity, previous researchers have proposed using functional dependency and other semantics to determine the update translation uniquely[BS81, DB82, K85]. In contrast, using the approach taken for integrating multiple databases, we propose to use additional language features to allow the schema administrator to provide the update translation.

Our approach is based on the notion of an update program that is similar to the view definition for queries. Just as view definition is a named, parameterized collection of (query) expressions, an update program is a named parameterized collection of update and query expressions. Parameterized view definitions allowed the use of the named views with varying set of bindings. For example, the above **empMgr** view can be used to query the managers of any employee, employees of any manager, and managers of all employees. The update programs has similar reuse capability and such a reuse enables update programs to be defined, successively on other update programs, in a nonrecursive fashion. This parallels approach taken in $\mathcal{LDL}$[NK88] and extended to allow updates to other structures beyond set (e.g., tuple) as well as integrate it with the higher order expressions.

Using the update program capability, a view update can be defined as a named parameterized definition of the update to the view. In such a definition, the schema administrator who is setting up the program can unambiguously state the necessary update (e.g., either update **emp** or **dept** relation) and the users can then use these updates transparently. Further, the reuse and parameterization capability provides the ability to build view updates using other view updates, especially for views which are defined on other views. Thus, the ability to allow update programs and the availability of update programs set up by the schema administrator(s) provide the view update capability. A similar proposal to provide updatability of views was originally proposed in the context of OSQL in [K86].

In summary, we have outlined the need to use metadata and data to define the unified view as well as the customized user view. We have argued that the view definition should be capable of defining not only the content of the view but the number of views as well. Finally, the language should be capable of providing updates to the views.

3 IDL Overview

The conceptual structure of the language proposed in [KN88] is based on objects and expressions on objects. An object can be classified into one of three[3] categories: a set of objects, a tuple of objects, or atomic objects. Examples of atomic objects are integers, characters, etc. A tuple object is recursively defined as a collection of attribute/objects pairs, which is syntactically denoted 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 unique $attr_j$ attribute of the tuple. We refer to $object_j$ as the $attr_j$ object of the tuple. For example $(name : john, sal : 10K)$ is a tuple object. A set object is a collection of objects. For example, $\{(name : john, sal : 10K), \ldots\}$ is a set object whose elements are tuple objects.

Elements of a tuple can be accessed by reference to the unique attributes, serving to "name" objects, whereas a set is queried by the contents of the object alone. Using this intuition we model a collection of named relational databases as a tuple; the collection of named relations in a database also as a tuple; and each relation as a set of (unnamed) tuples.

Modeling Multiple Relational Databases:

Define the *universe* of databases to be a tuple of relational databases[4] as follows:

$$\begin{aligned}
u = (&db1 : (r_{11} : \{(a_{111} : o_{111}, \ldots) \ldots\}, \\
&\qquad r_{12} : \{(a_{121} : o_{121}, \ldots) \ldots\} \ldots), \\
&db2 : (r_{21} : \{(a_{211} : o_{211}, \ldots) \ldots\}, \\
&\qquad r_{22} : \{(a_{221} : o_{221}, \ldots) \ldots\} \ldots) \\
&\vdots \\
)&
\end{aligned}$$

[3] The original proposal also included the category of a functor object which has been omitted throughout this paper for brevity.

[4] The databases are assumed to be relational for purely convenience of exposition. The reader will readily realize that the proposal allows more general DBMS in the same formalism.

Each relational database *dbi* in the above tuple is a tuple of relations. Each relation r_{ij} in each database is a set of tuples, and each tuple in a relation is a tuple of objects. These objects in a tuple are atomic, as per relational data model.

In short, we have outlined a nested relational model defined recursively using the three categories of objects (i.e., set, tuple and atom). There are couple of salient points of distinction that need to be emphasized here.

- Objects are value based and as defined above does not have a notion of object identity.
- Set can contain heterogeneous objects. Therefore, tuples in the above definition can have varying arity in a given relation. In contrast, tuples in relational model have a fixed arity.
- The above definition assumes that the attributes of each tuple in a relation to be atomic. This is done for ease of exposition and can be easily generalized to allow any nested object.

Using this model of the universe of databases, we develop the language features of interest.

4 Higher Order Queries

We develop the syntax and semantics of the query expression recursively using the three categories of objects. Corresponding to each kind of object we have a query expression (or referred to as expression, if there is no ambiguity) of that type. An expression, evaluated on an object, returns true or false[5]. An atomic (resp. tuple or set) expression evaluates on an atomic (resp. tuple or set) object and returns true only for an atomic (resp. tuple or set) object. If an expression evaluated to true on an object then we say that the object *satisfies* the expression. We define ϵ (i.e., empty string) to be a tautological expression that is satisfied by any set, tuple or atomic object.

In the first subsection we develop the syntax of the query expression. The evaluation of the query expression on an object, (i.e., semantics) is the topic of the next subsection.

4.1 Query Expression on Objects

An *atomic expression* is of the form α`X` where `X` is either a variable (denoted by a word beginning with a capital letter) or a constant (denoted by a word that is not a variable) and $\alpha \in \{<, \leq, >, \geq, =, \neq\}$. For example, $>$ `Cat` and $=$ `cat` are atomic expressions.

A tuple expression is a conjunct of the form: $.\mathtt{a}_1\mathtt{exp}_1, .\mathtt{a}_2\mathtt{exp}_2, \ldots, .\mathtt{a}_k\mathtt{exp}_k$, where each $\mathtt{exp}_j$ is an expression on the $\mathtt{a}_j$-object of the tuple.

[5] In this sense, the term expression is a misnomer. A more appropriate name would be an *operator*. Nevertheless, we define it as it was done in [KN88].

A set expression is defined to be (`exp`) in which `exp` is an expression on an object of the set.

Any expression `exp` can be negated by prefixing the negation operator $\neg$ to get $\neg$`exp`.

The subset of atomic expressions ensuing from restricting $\alpha \in \{=\}$ shall be called *simple atomic expressions*; a *simple expression*, in general, does not use any atomic expressions that is not simple and that does not have any negated expression; a *simple ground expression* is a simple expression that does not have any variable. Such simple expressions can be further qualified as simple tuple expression or simple ground set expression to mean the obvious restriction.

In summary we show below the grammar for expressions, on *constants* and *variables*. Readers familiar with [KN88] will readily see the extensions for negation and the more general ability to have variable representing aggregate objects (i.e., tuple and set).

$$
\begin{array}{ll}
Exp & \rightarrow \neg PExp \mid PExp \\
PExp & \rightarrow Aexp \mid Texp \mid Sexp \mid \epsilon \\
Aexp & \rightarrow Relop\ constant \mid Relop\ variable \\
Texp & \rightarrow .Aname\ Exp \mid .Aname\ Exp, Texp \\
Sexp & \rightarrow (Exp) \\
Aname & \rightarrow constant \\
Relop & \rightarrow < \mid \leq \mid = \mid \neq \mid > \mid \geq
\end{array}
$$

We define a query to be an expression of the form, '`?exp`' where `exp` is an expression on the universe tuple as defined above.

We have so far presented the syntax of an expression. The evaluation of an expression (on an object) provides the semantics of the expression.

4.2 Expression Evaluation

A *substitution* is defined, in the usual way, as a non-empty finite set of ordered pairs $\{\mathtt{X}_1/\mathtt{o}_1, \ldots, \mathtt{X}_n/\mathtt{o}_n\}$ such that $(\forall_{1 \leq i \leq n})\ \mathtt{X}_i$ is a distinct variable, $\mathtt{o}_i$ is an object. We view a substitution as a mapping on variables that is the identity almost everywhere. Thus, if σ is a substitution and `X` a variable the result of *applying* σ to `X` is defined as `o` if $(\mathtt{X}/\mathtt{o}) \in \sigma$ and `X` otherwise. We extend this mapping to expressions in a manner consistent with the above definition. The idea behind an application of σ to an expression `e` is to replace the free occurrences of $\mathtt{X}_1, \ldots, \mathtt{X}_n$ in `e` by the objects $\mathtt{o}_1, \ldots, \mathtt{o}_n$. A substitution σ is said to be a *grounding* substitution for an expression `e` if `e`σ is ground; i.e., all variables are substituted in the expression.

The evaluation of the expression is a recursive algorithm with the base case being the evaluation of an atomic expression.

A ground (i.e., free of variables) atomic expression, say α`c`, evaluated on an atomic object, say `o`, returns true if the comparison `o`α`c` is true. An atomic expression, α`c`, evaluated on an atomic object `o` returns true, if there exists a substitution σ such that `o`α`c`σ is true.

A tuple object o satisfies a tuple expression $.a_1exp_1, .a_2exp_2, \ldots, .a_nexp_n$ if there exists a grounding substitution σ such that $(\forall_{1 \le i \le n})$ $.a_iexp_i$, o has an a_i-object that 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 $exp\sigma$.

An object satisfies a negated expression $\neg$exp iff the object does not satisfy the expression exp.

We define the answer to a query to be the set of grounding substitutions satisfying the query. One can extrapolate introducing a structure to the answer. As this is not directly relevant in this paper, we do not elaborate on this aspect of the problem. In the limiting case, when there is no variable in the query, the answer is assumed to be boolean.

Example: Consider the universe (u) of stock databases in example presented in the Introduction. Below we show some query expressions on the euter database.

```
?.euter.r(.stkCode=hp, .clsPrice>60)
```
"Did hp ever close above 60?"

The universe tuple u must satisfy the above tuple expression for the query to be true. For this, the suffix after euter in the query (i.e., .r(...)) must be satisfied by the euter object (i.e., euter tuple) of the universe tuple. That is, the .r-object (i.e., relation r) of the euter tuple must satisfy the set expression (.stkCode=hp, .clsPrice>60), which in turn requires the existence of a tuple in r that satisfies the tuple expression .stkCode=hp,.clsPrice>60.

```
?.euter.r(.stkCode=hp,.clsPrice>60,.date=D),
.euter.r(.stkCode=ibm,.clsPrice>150,.date=D)
```
"List all dates when hp closed above 60 and ibm closed above 150."

This is an example of select, join (i.e., self join of r) and project (i.e., list[6]). The ordering of the attributes is immaterial because the attributes are named.

```
?.euter.r(.stkCode=hp,.clsPrice=P,.date=D),
.euter.r¬(.stkCode=hp, .clsPrice>P)
```
"List the dates/prices when price of hp closed at its all time high."

The highest close price is obtained by negating the existence of a higher price. This is an example of negation and inequality join.

```
?.euter.r(.stkCode=S, .clsPrice>200)
```
"Did any stock ever closed above 200."

This above seemingly simple example will provide the motivation for the higher order expression. ∎

We have showed, in the above examples, that the language has the usual relational algebra capabilities such as join, selection, negation etc. Thus, it has sufficient power to construct any relational expression. All the above queries required the variables to quantify over the data (i.e., first order quantification) in the relation r in euter. This is no more true, if the last query in the above example is to be posed against the other two databases (i.e., chwab and ource). This is because, the variable S needs to be quantified over the stocks which are denoted as attributes and relation names in the two databases. Traditional query languages such as OSQL[Fi89], QBE and SQL and as well as logic languages such as Datalog, $\mathcal{LDL}$ do not allow such higher order quantification.

4.3 Higher Order Expressions

The definition of tuple expression above restricted attribute names to be values. We remove this restriction and define a notion of higher-order quantification over attribute names in order to pose query on meta information as well. We generalize the tuple expressions by allowing variables and constants for attribute names as follows: $.A_1exp_1, A_2exp_2, \ldots, A_kexp_k$, where each A_i, $i = 1, \ldots, k$, is either a variable for an attribute name or an attribute name itself. As a result, each exp_j is an expression on the object associated with the A_j attribute of the tuple.

A variable occurring in an attribute position in an expression will be referred to as a *higher-order variable*. We define a *higher-order expression* as an expression defined as before, using the new definition for tuple expression. The evaluation algorithm remain unchanged; i.e., the grounding substitution provides the binding for all variables then the satisfaction can be checked as before. Intuitively, the semantics of the higher order expression is given by the semantics of the resulting first order expressions obtained through substitution for higher order variables. We explain the semantics of such higher-order queries through the following examples.

Example: Consider the univese of stock databases again.

`?.X`	List the database names in the universe."
`?.ource.Y` or `?.X.Y, X = ource`[7]	List the relation names in the ource database in the universe "
`?.X.Y`	List the database/relation names in all the databases in the universe."
`?.X.hp`	List the names of databases containing a relation named hp."
`?.X.Y(.stkCode)`	List the names of database/relation containing an attribute named stkcode."
`?.chwab.r(.date=D,.S=P), .ource.S(.date=D,.clsPrice=P)`	List the stocks in ource and chwab that have the same closing price."
`?.euter.Y, .chwab.Y, .ource.Y`	List the names of relations that occur in all the databases "

The higher order variables (e.g., X, Y) can be used like

[6] Here 'List' is being used informally to refer to the free variable. We shall revisit the notion of list (i.e., project) in the context of derived views.

[7] Strictly, this is not an expression allowed by the grammar. This is a construct that is used very similar to the use in Datalog, $\mathcal{LDL}$, Prolog and other Horn clause based languages. The reader can extrapolate its meaning in the obvious manner.

any other variables in the expression, in the sense that they can be used to conjunct further selection, join or even projection as we shall see later. It is interesting to note that many of the above queries are very useful in a heterogeneous database environment where all the databases are autonomously administered.

Finally we revisit the last query expression of the previous example in the context of chwab and ource.
`?.chwab.r(.S>200)`
`?.ource.S(.clsPrice > 200)`
"Did any stock ever closed above 200."

The variable S is quantified over names of attributes and relations, respectively in the two databases. ∎

5 Update Expressions

We have so far been concerned with query expressions whose satisfaction can be evaluated for a given object. For example, the query expression
`?.chwab.r(.date=3/3/85,.hp = 50).` can be read as "Is it true that hp closed at $50 on 3/3/85?" In general, a query expression evaluates the truth, á la the expression.

In contrast, an update expression is a decree that proclaims the truth hence forth. For example, if a tuple for hp for 3/3/85 is inserted with a closing price of $50 then such an update can be viewed as a decree that the above query expression will be true hence forth. Similarly, if a tuple is deleted then the decree is the falsehood of the query expression hence forth. These insert and delete expressions are syntactically stated as **+exp1** and **-exp2** respectively. These expressions are to be read as *make* **exp1** *true hence forth* and *make* **exp2** *false hence forth.*

As with the query expressions, we present the syntax of the update expressions and then describe the semantics by specifying the evaluation algorithm.

5.1 Update Expression on Objects

Recall that a simple atomic expression is of the form `=constant` or `=Variable`; and a simple expression is one that does not use any atomic expression that is not simple. A *simple ground expression* is defined to be a simple expression that does not have any variable and that does not have any negated expression (i.e., ¬**exp**).

The syntax of an update expression is either + (for insert) or - (for delete) followed by atomic, tuple or set expression that is simple and ground. The grammar is

$$\begin{aligned} UExp &\rightarrow +SGexp \mid -SGexp \\ SGexp &\rightarrow a\ simple\ ground\ expression. \end{aligned}$$

Some examples of update expressions are as follows:

- *atomic update expressions:* `+=5`, `-=9`
- *tuple update expressions:* `+.`a_i $\mathbf{exp}_i$, `-.`a_j $\mathbf{exp}_j$
- *set update expressions:* `+(`$\mathbf{exp}_k$`)`, `-(`$\mathbf{exp}_l$`)`

where $\mathbf{exp}_i$, $\mathbf{exp}_j$, $\mathbf{exp}_k$, $\mathbf{exp}_l$ are all simple ground (query) expressions.

We refer to expressions of the form **+exp / -exp** as a *plus/minus expression* respectively. These may be further qualified as atomic, tuple or set to be specific. In keeping with the left to right precedence of operations, the following two expressions are *not* equivalent:
$+ .a_1\mathbf{exp}_1, .a_2\mathbf{exp}_2, \ldots, .a_k\mathbf{ex}_k$ and
$+.a_1\mathbf{exp}_1, +.a_2\mathbf{exp}_2, \ldots, +.a_k\mathbf{exp}_k$.
Therefore, in the following discussion when we refer to making **exp** true (or false), and if **exp** is a conjuncted tuple expression, we mean the second expression, even though for brevity we refer to it syntactically as **+exp** (or **-exp**).

An update request is of the form $?\mathbf{exp}_1, \mathbf{exp}_2, \ldots, \mathbf{exp}_k$, where $\mathbf{exp}_i$ is either an update or query expression.

5.2 Update Expression Evaluation

As before, an atomic (similarly, tuple or set) update expression is defined only on an atomic (similarly tuple or set) object. For all other cases, the expression is in error and the results are undefined. For the purpose of describing the evaluation semantics of the update expression, we define two special objects: *null atomic object* and *empty object.* Null atomic object is the null value. For ease of exposition, we make a simplifying assumption that such a null value evaluates to false for all atomic expressions. An empty object behaves as an empty set, empty tuple or a null atomic object depending on the context. Thus, all update expressions are valid on an empty object.

The atomic plus expression, `+=c` is evaluated on an atomic object by replacing the object with the value c and thus making the atomic expression (i.e., `=c`) true hence forth. The atomic minus expression, `-=c`, is evaluated on an atomic object by replacing the value with null, if the atomic object satisfies the expression `=c`, making the atomic expression false hence forth; otherwise unchanged.

A tuple plus expression, `+.`a_i $\mathbf{exp}_i$, is evaluated on a tuple object as follows: First, if the attribute a_i doesn't already exists, then create the attribute a_i in that tuple. Second, associate with that attribute an empty object and thereby implicitly deleting any existing object associated with that attribute. Last, recursively evaluate the update expression $+\mathbf{exp}_i$ on the a_i object (i.e., the empty object). A tuple minus expression, $-.a_i\mathbf{exp}_i$, is evaluated on a tuple object by deleting the attribute $.a_i$ as well as the associated object from the tuple if the object satisfies the expression $\mathbf{exp}_i$. Thus, any query expression of the form `.`a_i **exp** will evaluate to false hence forth for that tuple.

A set plus expression, **+(exp)** is evaluated on a set, **s** object by first creating a new empty object and recursively evaluating **+exp** on that empty object. Then adding the resulting object to the set, **s**. A set minus expression, **-(exp)** is evaluated on a set object by deleting all elements of the set that satisfies the (query) expression **exp**.

We exemplify the above evaluation semantics. Consider the following two set update expressions.
`?.euter.r+(.date=3/3/85,.stkCode=hp,.clsPrice=50).`

```
?.euter.r-(.date=3/3/85,.stkCode=hp).
```
The first expression inserts the tuple in the set r and the second expression deletes all tuples in r for hp with 3/3/85 date. Both these expression can be made query dependent as seen in the following equivalent expression to the above delete request.

```
?.euter.r(.date=3/3/85,.stkCode=hp,.clsPrice=C),
.euter.r-(.date=3/3/85,.stkCode=hp,.clsPrice=C).
```
The above use of variables in the set minus expression is not a contradiction to the requirement that the expression be simple and ground. This delete can be viewed as a series of delete expression, one for each value of closing price in the set r. This approach was also taken in QBE[Z77] and $\mathcal{LDL}$[NK88] and the formal semantics can be given in a similar manner.

Another use of variables in the expression can be seen in the following delete request in the chwab database, exemplifying the use of atomic minus expression.
```
?.chwab.r(.date=3/3/85, .hp=C),
   .chwab.r.(.date=3/3/85, .hp-=C).
?.chwab.r(.date=3/3/85, .hp=C),
    .chwab.r.(.date=3/3/85, -.hp=C).
```
The both the expressions deletes the closing price for hp on 3/3/85 from the chwab database but the second expression also deletes the attribute. Based on the null value semantics assumed, all query expression on the hp attribute for that tuple will not be satisfied after both the updates. In this sense, they behave identically.

Note that the deletion, of the attribute hp from the tuple, has the effect only in the tuple for the date 3/3/85. This is allowed in this language because a set can contain heterogeneous elements, which is a marked contrast to most relational database systems. Such an update would obviously pose implementation problems.

As a short hand notation, we can use the following grammatically illegal delete expression to mean the same as the delete above.

```
?.chwab.r.(.date = 3/3/85,  .hp- = C).
```

The updates in this language can be viewed as the composition of delete followed by insert.
```
?.chwab.r-(.date=3/3/85,.hp=C),
   .chwab.r+(.date=3/3/85,.hp=C+10).
```
This updates the closing price of hp on 3/3/85 to be \$10 more than the previous value[8]. Note that the ordering of these two update requests is relevant as the reverse ordering would not result in the same semantics. This was not the case in the case of query expressions.

In summary, we have provided the capability to update any set, tuple or atomic object such that both the metadata and the data can be updated in the same expression. Such an update capability is considered essential to pose nontrivial updates to heterogeneous databases.

[8] We have assumed the use of arithmetic here even though it was not included in the grammar. The reader can extrapolate the possible uses.

6 Higher Order Views

Derived views are synonymous to derived predicates in Horn clause language. We use the calculus to define views over all the databases in the enterprise. Traditionally, a rule in Horn clause language defines a single derived view or predicate which is named. In this section we extend this notion to allow the definition of many views using a single rule. Such views are termed as *higher order views*. We show the use of higher order views to provide both database and integration transparency.

We define a rule as an implication *head* $\leftarrow$ *body*, in which *head* is a simple tuple expression, expH and *body* is any general tuple expression, expB, both on the universe tuple, such that all variables in the head occur in the body. Intuitively, view the body as an expression on the universe tuple using the variables $X_1, \ldots, X_k$ that occur in both the head and the body expressions. For each grounding substitution, σ satisfying expB, the object expHσ is made true in the universe denoted as +expHσ. This notion of making expH true is recursively defined as follows.

In other words, the derived fact is made true in the universe tuple. Such a rule provides the mechanism to define *derived views*. A *derived database view* for db1 is defined by all the rules with the head .db1 exp. Similarly, A *derived relation view* r in a database db1 is defined by all the rules with the head .db1.r exp. A derived database (similarly relation) view is called derived database (similarly, relation) higher order view if the head (i.e., the expression exp) contains a higher order variable.

We first show the use of the derived views to unify schemata below and then describe the semantics in detail.

Example: Consider the universe of stock databases again.

```
.dbI.p(.stkCd=X,.dat=D,.clsPr=P,.db=euter)
    ← .euter.r(.date=D,.stkCode=X,.clsPrice=P)
.dbI.p(.stkCd=X,.dat=D,.clsPr=P,.db=chwab)
    ← .chwab.r(.date=D,.X=P), X≠date
.dbI.p(.stkCd=X,.dat=D,.clsPr=P,.db=ource)
    ← .ource.X(.date=D,.clsPrice=P)
```

The relation p is placed in a unified database called dbI. This exemplifies the projection of higher order variables and the use of derived views to unify schemata.

```
.dbE.r(.stkCode=X, .clsPrice=P, .date=D)
      ← .dbI.p(.stkCd=X,.dat=D, .clsPr=P)
.dbC.r(.X=P, .date=D)
      ← .dbI.p(.stkCd=X,.dat=D,clsPr=P)
.dbO.X(.clsPrice=P, .date=D)
      ← .dbI.p(.stkCd=X,.dat=D, .clsPr=P)
```

The dbE, dbC and dbO databases provide a database with a compatible schema for euter, chwab and ource users respectively; i.e., provide the users integration transparency. Note, the dbO database has a view definition that defines as many relations as there are stocks

in all three databases. This requires the definition of the view to be stratified. The formal semantics of a program in this language is given in [KLK90].

If there is any value discrepancy amongst the prices for the same stock for any given day, then both prices are in the user's view, as defined above.

```
.dbI.pnew(.stkCd=X,.clsPr=P,.dat=D)
  ← .dbI.p(.stkCd=X,.dat=D,.clsPr=P,.db=euter)
.dbI.pnew(.stkCd=X,.clsPr=P,.dat=D)
  ← .dbI.p¬(.stkCd=X,.dat=D,.db=euter),
    .dbI.p(.stkCd=X,.dat=D,.clsPr=P,.db=chwab)
.dbI.pnew(.stkCd=X,.clsPr=P,.dat=D)
  ← .dbI.p¬(.stkCd=X,.dat=D,.db=euter),
    .dbI.p¬(.stkCd=X,.dat=D,.db=chwab),
    .dbI.p(.stkCd=X,.dat=D,.clsPr=P,.db=ource)
```

Using pnew the individual views can be redefined so that each stock is associated with a unique price. Note that the choice of any such reconciliation is up to the schema administrator. Here, we only provide the language to specify the reconciliation.

As a last example, let us relax the assumption that the stock codes in euter database and the names of attributes/relations in chwab/ource database respectively are from the same domain and have no name conflict. If there is such a discrepancy, then we can define name mappings (i.e., binary relations) mapCE and mapOE from chwab/ource to euter respectively. Using these two name mappings, we can define the unified view as follows:

```
.dbI.p(.stkCd=X,.dat=D,.clsPr=P,.db=euter)
     ←.euter.r(.date=D,.stkCode=X,.clsPrice=P)
.dbI.p(.stkCd=X,.dat=D,.clsPr=P,.db=chwab)
     ←.chwab.r(.date=D,.X1=P), X1≠date,
       .mapCE(.cStk=X1,.eStk=X)
.dbI.p(.stkCd=X,.dat=D,.clsPr=P,.db=ource)
     ←.ource.X1(.date=D,.clsPrice=P),
       .mapOE(.oStk=X1,.eStk=X)
```

Obviously, these two name mapping relations need to be maintained and any updates to the database have to have an appropriate tuple added to the mapping relations. In general, using the power of the Horn clause language, any such reconciliation can be devised. Thus, exemplifying the power of the language. ∎

In summary, p is a unified view of the three databases and by using p the user achieves database transparency. This required the use of higher order variables in the body of the rule. The individual views (e.g., .dbE and .dbO for euter and chwab users) provide the integration transparency by providing a view that is consistent with the ones the users are used to, prior to integration. In order to do this, we used higher order variable in the head of a rule (i.e., defining a higher order view). The definition of pnew showed the use of the powerful language to reconcile any value differences.

7 Update Capabilities

The update capability, in a nutshell, is the user's ability to update via their customized view and the appropriate databases updated correctly. In order to provide this view updatability, we present a notion of update programs.

Traditionally, the concept of views is the notion of aggregating a collection of query expressions and giving it a name for subsequent use. This can be extrapolated in the context of updates wherein the collection of selection and update expressions, called update programs can be aggregated for subsequent use. Such aggregations are syntactically quite similar (if not identical, in most cases) to the view definition. This concept of update programs is used to provide the view updatability feature.

7.1 Update Programs

Even though the syntax of update programs and view definitions are quite similar we refer to them as *update programs*, to emphasize the update nature and the restrictions in the use and definitions. Further, the definition of these update programs differ from view definition (i.e., a rule) by the use of a right arrow (i.e., →) instead of a left arrow (i.e., ←) and the body of the definition is any update expression.

Semantically, the arguments in the head are to be viewed as parameters instead of values deduced from the body of the definition. These parameters are passed top–down and therefore a top-down semantics is given for these programs. Further, execution of these programs does not return values except success or failure. As is the case in $\mathcal{LDL}$ updates [NK88], we disallow any recursive call to update program. This enables the use of top-down semantics for update programs without any loss of generality. We present this top down evaluation of the update programs informally through examples.

Consider the following example that aggregates the set of updates to the three databases in one program called delStk.

```
.dbU.delStk(.stk=S, .date=D) →
                    euter.r−(.stkCode=S,.date=D)
.dbU.delStk(.stk=S, .date=D) →
                          .chwab.r(.S-=X,.date=D)
.dbU.delStk(.stk=S, .date=D) →
                                .ource.S−(.date=D)
```

delStk deletes the closing price of a given stock code on a particular date. This program describes the translation of the update to the respective databases. Note that this program has the property that if the stock code is not passed as input then the closing price of all stocks for that date is deleted. If the date is not given as input then the closing price of all the days for that stock are deleted. If both stock code and date are not given then all values are deleted. But the structure of the database is not changed; i.e., chwab database will still contain attribute names called hp, ibm etc.

The following program removes a given stock from all the databases exemplifying the need to construct update programs that not only updates data but also the metadata.

```
.dbU.rmStk(.stk=S) → .euter.r−(.stkCode=S)
.dbU.rmStk(.stk=S) → .chwab.r(−.S )
.dbU.rmStk(.stk=S) → .ource−.S
```

The process of removing stocks from the **euter** database is to remove the tuples from the relation. On the other hand, the removal of a stocks from **chwab** and **ource** database requires the deletion of attributes and relations respectively. The reader should note that the execution of some of these updates might pose nontrivial performance problems, if it can be done at all within the restrictions of the underlying DBMS.

The above two programs were valid even if some (or all) of the arguments were not given as input. This feature is quite useful for increasing the capability to reuse the same update program in different context and still provide a consistent semantics among similar updates (e.g., deleting for a given stock/date and for a given stock). This useful feature cannot be always provided as seen in the following program that inserts the closing price of a given stock on a particular day.

```
.dbU.insStk(.stk=S, .date=D, .cls=P) →
     .euter.r+(.stkCode=S,.date=D, .clsPrice=P)
.dbU.insStk(.stk=S, .date=D, .cls=P) →
                          .chwab.r(.S+=P,.date=D)
.dbU.insStk(.stk=S, .date=D, .cls=P) →
                  .ource.S+(.date=D, .clsPrice=P)
```

Note that if any of the argument is not given then the plus expressions are not defined. This can be used to define the necessary bindings for which a given update program is defined. Such compile time analysis can be used to check the validity of the 'call' to the **insStk** program.

The reader may have already observed that the operations such as **delStk** or **rmStk** may not be expressible in the customized user view using relational language capability. The point that is being made here is that the schema administrator can define such update programs which are then used to provide the updates to the customized views.

In summary, we have allowed for named, parameterized update programs that can be used to construct other update programs in a nonrecursive fashion. Note that the update operations (i.e., + and −) have been allowed only on extensional objects in the universe. In other words, any view defined using a rule cannot be updated using + and −. This requires the view updatability extension of the next subsection.

7.2 View Updatability

The ability to update a given view is similar to the base updates defined earlier in the sense that the update is a decree about the truth regarding the view, hence forth. For example, if **p** in **dbX** is a view defined over the three databases, then **.dbX.p+(exp)** is a decree that the set **p** in **dbX** will satisfy the expression **exp** hence forth. For base objects, such updates can be performed by changing the state of the database (e.g., inserting or deleting tuples from relations). In the case of the derived views, the updates have to be to the base objects, such that subsequent computation based on the updated base objects guarantees the decree.

This is traditionally called the view update problem— the correct translation of updates to the view to the updates on the base relation such that the resulting views based on the updated relation is faithful to the view update semantics. Traditionally, [BS81, DB82, K85] automatic translation of the view updates has met with limited success, even in the context of a single relational database. In this paper, we relax the requirement to translate the view update automatically.

In keeping with our language oriented approach to providing interoperability of databases, we seek to provide the language capability to allow the schema administrator to state these translation. In particular as update programs. Thus, for each derived view **dbX.p** in the database, the schema administrator can provide named update programs such as

dbX.p + (exp) →

dbX.p − (exp) → .

These update programs can be used to construct other programs and in particular the view updates of other views that are dependent on the the derived predicate **dbX.p**. Note that if more than one (say) plus update to the view is defined, due to different set of parameters to the update program, then some appropriate naming convention needs to be adopted by the administrator so that they can be unambiguously used. In general, a binding signature can be associated to each view update program as well as with the use of these update programs. These signatures can be used to determine the correct definition that corresponds to the use.

In short, the capability of named aggregation of update and query expressions can be used to define the view update and these update programs can be reused to define other view update programs. The update capability that is being provided to the user is limited to the relational language capability. The only departure is that the semantics of such an (relational) update to the the customized views may be stated by update programs that is beyond the capabilities of a relational language.

8 Conclusion

We have presented some language features that are needed to deal with SD's and transparencies. We argued that the metadata and data need to be explicitly represented as complex objects so that both types of data can be used to reason in reconciling the heterogeneity among databases. Extrapolating this observation, it may also be necessary to include other schematic information such as types, keys, referential integrity etc. Such extensions are also possible in such a language.

We argued that higher order expressions wherein variables ranging over data and metadata are needed not only to support the queries but also to define a unified view of all the databases. This unified database provides the database transparency so that the users can formulate queries spanning multiple databases.

We also demonstrated the need for higher order view definition capable of defining varying number of relations depending on the state of the database. This is in contrast to the traditional approach where the view definition facility allows a fixed set of relations for all states of the database.

The necessity to provide view updatability was directly evident from the two-level mapping. The approach of providing this updatability is a marked departure from the traditional approach, in the sense that we provided the language capability to specify the update uniquely. The responsibility of stating the required update in the language is relegated to the schema administrator.

Based on the understanding of these features in the above cryptic Horn clause based language, the next step is to incorporate these features in a language with enough syntactic sugar. In particular, our goal is to incorporate them into OSQL the functional query language for Iris [Fi89] which is the basis for the Pegasus multidatabase system under development in H.P. Labs. An interesting alternative is to view the incorporation of these features in a language that provides extensibility. Such a language has been proposed in [A90], using which many of the features in this paper can be supported.

Acknowledgements: We sincerely thank J. Annevelink, W.Hasan, M. Ketabchi, A. Rafi and J.D. Ullman for their comments and suggestions.

References

[AG88] Abiteboul S., and Grumback S., COL: A Logic based Language for Complex Objects. *Advances in Database Technology—EDBT88*, Venice, Italy, pp271-293 1988.

[A90] Annevelink, J., Database Programming Languages: A Functional Approach, Submitted for Publication, HP Labs Technical Memo HPL-DTD-90-12, Palo Alto, 1990.

[BK86] Bancilhon, F., and Khoshafian, S.: A Calculus for Complex Objects, *ACM PODS Conf.*, 1986.

[BS81] Bancilhon, F., and Spyratos, N.: Update Semantics and Relational Views, *ACM TODS*, Vol. 6, No. 4, 1981.

[CKW89] Chen, Weidong, M. Kifer, and D.S.Warren: Hilog: A first-order Semantics for Higher Order Logic Programming Constructs, *Proc. of 2nd Int. Workshop on Database Programming Languages*, Salishan, OR, 1989

[DB82] Dayal, U., and Bernstein, P. A.: On the Correct Translation of Update Operations on Relational Views, *ACM TODS*, Vo. 7, No. 3, Sept. 1982

[DH84] Dayal, U., and Hwang, H.: View Definition and Generalization for Database Integration in a Multidatabase System, *IEEE Tras. on Soft. Engg.*, SE-10, 6, (Nov.), 1984, pp628-644

[Fi89] Fishman, D. H., et al.: "Overview of Iris DBMS", *Object-Oriented Concepts, Lang., and Appl.*, Edited by W. Kim and F.H. Lochovsky, Addison Wesley Publ. Co., 1989.

[K85] Keller, A. M.: Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections and Joins, *ACM Sym. on PODS*, 1985.

[K86] Kent, W.: Future Work in Iris, HPL Internal Memo, 1986.

[K89] Kent, W.: The Many Forms of a Single Fact, Compcon 89, San Francisco, 1989.

[KLK90] Krishnamurthy, R. Litwin, W., Kent, W.: Language Features for Interoperability of Databases with Schematic Discrepancies, *Technical Memo HPL-DTD-90-14*, 1990.

[KN88] Krishnamurthy, R. and Naqvi, S.: Towards a Real Horn Clause Language, *Proc. of VLDB*, Los Angeles, 1988.

[Li89] Litwin, W.: MSQL: A Multidatabase Language, Elsevier Science Publishing, 1989

[MB81] Motro, A., and Buneman, P.: Constructing Superviews, *Proc. of SIGMOD*, Ann Arbor, 1981.

[NK88] Naqvi, S., and Krishnamurthy, R.: Database Updates in Logic Programming, *ACM Sym. on PODS*, 1988.

[NT89] Naqvi, S. A. and S. Tsur. *A Language for Data and Knowledge Bases*, W.H. Freeman, 1989.

[NSF] Brodie, M.L., et. al: Database Systems: Achievements and Opportunities, NSF Report.

[R*90] Rafii, A., Ahmed, R., DeSmedt, P., Kent, W., Ketabchi, M., Litwin, W., and Shan, M.: Overview of Multidatabase Management in Pegasus, Submitted for publication.

[Z77] Zloof, M.M.: Query-By-Example: a database language, *IBM Systems Journal*, Vol. 16:4.