

# Deductive Queries in ODMG Databases: the DOQL Approach

Pedro R. F. Sampaio and Norman W. Paton  
Department of Computer Science  
University of Manchester  
Oxford Road, Manchester, M13 9PL, UK  
(sampaio,norm)@cs.man.ac.uk

## Abstract

The Deductive Object Query Language (DOQL) is a rule-based query language designed to provide recursion, aggregates, grouping and virtual collections in the context of an ODMG compliant object database system. This paper provides a description of the constructs supported by DOQL and the algebraic operational semantics induced by DOQL's query translation approach to implementation. The translation consists of a logical rewriting step used to normalise DOQL expressions into molecular forms, and a mapping step that transforms the canonical molecular form into algebraic expressions. The paper thus not only describes a deductive language for use with ODMG databases, but indicates how this language can be implemented using conventional query processing techniques.

## 1 Introduction

The ODMG standard is an important step forward due to the provision of a reference architecture for object databases. This architecture encompasses an object model and type system, a set of imperative language bindings and the OQL declarative query language. Perhaps the cornerstone contribution of the ODMG specification is the definition of a standard object model that can be accessed and manipulated by different languages, catering for portability and supporting different paradigms of interaction with object databases (currently, imperative by way of language bindings and functional, by way of OQL).

Besides the programming API supported by the DML of the ODMG imperative language bindings (viz. C++, Java and Smalltalk) and the OQL query language that supports interactive and embedded declarative access modes in a functional style, new modes of interaction with ODMG compliant databases are being investigated in the fields of visual query languages [7, 13] and computationally complete query language extensions [17].

In this paper we report on the use of the deductive paradigm for querying ODMG databases through DOQL. DOQL is a rule-based database query

language that provides a deductive interface to an ODMG compliant object database. DOQL is designed to integrate object-oriented and deductive language constructs, and to support recursion, aggregates, grouping and virtual collections (views).

This paper presents the main features of DOQL, and the algebraic operational semantics of DOQL. The remainder of this paper is structured as follows. An overview of the features supported in DOQL is presented in section 2, followed by a description of the query translation approach defining the operational semantics of DOQL in section 3. Related work is presented in section 4, and the paper concludes in section 5 with a summary of the work and a discussion of future directions.

```
extern const char _boss[] = "boss";
extern const char _subordinates[] = "subordinates";
extern const char _workers[] = "workers";
extern const char _works_for[] = "works_for";

class Skill : public d_Object {
    d_String s_name;
    d_String rating;
};

class Person : public d_Object {
    d_String name;
    d_String surname;
    d_Ref<Person> father;
    d_Ref<Person> mother;
    d_UShort age;
    d_Set<d_Ref<Person>> dependents;
    d_Set<d_Ref<Skill>> skills;
};

class Employee : public Person {
    d_Rel_Set<Employee,_boss> subordinates;
    d_Rel_Ref<Employee,_subordinates> boss;
    d_Rel_Ref<Department,_workers> works_for;
};

class Department : public d_Object {
    d_String name;
    d_Rel_Set <Employee,_works_for> workers;
};
```

Figure 1: Database schema in C++ ODL.

Figure 1 describes the application schema used throughout the paper, specified in C++ ODL [14, 4]. The database conforming to the application schema supports the extents *persons*, *employees* and *departments*. It is assumed that

readers have some familiarity with the ODMG standard, and with deductive database technologies, as described in [4] and [5], respectively.

## 2 An Overview of DOQL

DOQL is designed to exploit language integration [1, 15], where a deductive language is integrated with an imperative programming language in the context of an object model or type system. In this strategy, the resulting system supports a range of standard object-oriented mechanisms for structuring both data and programs, while allowing different and complementary programming paradigms to be used for different tasks, or for different parts of the same task. The idea of integrating deductive and imperative language constructs for different parts of a task was pioneered in the Glue-Nail deductive relational database [9], and is now adapted for ODMG compliant object-oriented databases. The success of this strategy depends on the seamlessness of the integration of the deductive language and the imperative language, which is achieved by adopting ODMG types as the type system underlying the integration.

The architecture adopted for DOQL is illustrated in figure 2. The DOQL compiler and evaluator is a class library that stores and accesses rules from the ODMG database – the rule base is itself represented as database objects. The class library is linked with application programs and the interactive DOQL interface. The interactive interface is itself a form of application program. The system is designed with the goal of serving as a complementary and non-invasive query layer that can be used by application designers without the need to change existing data or programs.

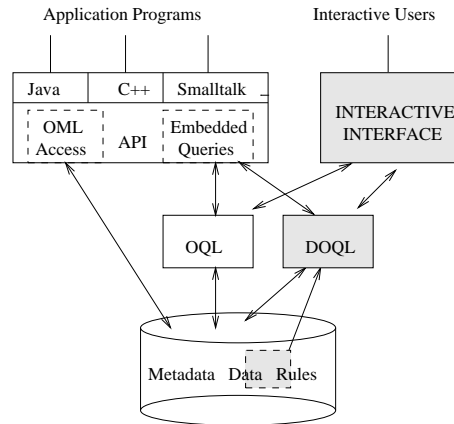


Figure 2: Architecture diagram showing location of DOQL compiler/evaluator.

## 2.1 DOQL Methods and Rules

Rules are used to define virtual collections and deductive methods over ODMG databases. A rule is a clause of the form:  $H \leftarrow L_1, L_2, L_3, \dots, L_n$ , where  $H$  is the *Head* of the rule and the body is a sequence of literals denoting a formula. The syntactic form of the head depends on whether the rule is a *regular clause* or a *method clause*.

### 2.1.1 Regular Clauses

Regular clauses are rules that specify a virtual collection (intensional database predicate) over the stored database. Virtual collections can be used to define new types that can model associations, aggregations and specializations involving stored objects. In regular clauses, the head of the rule is of the form:

`< rulename > (arg1, ..., argn)`

where each  $arg_i$  is a variable, an atomic constant, a compound constant, or a grouping expression applied to a term. Rules can be recursive, can range over collections, can traverse ODMG relationships (collection valued relationships using the operator  $\Rightarrow$ , and single valued using  $\rightarrow$ ), and combine positional (access to arguments of rule heads) and named approaches (access to properties and operations of objects) to attribute denotation. The following rule defines a virtual collection formed by the names of same aged siblings of a person object.

```
sameAgeSiblings(N1,N2) :- persons[dependents=>P1,dependents=>P2],
                          P1 != P2, P1.age = P2.age,
                          P1.name = N1, P2.name = N2.
```

*Grouping* is the process of grouping elements into a collection by defining properties that must be satisfied by the elements. Grouping is restricted to rules that contain a single clause and to a single argument position in the head of the clause. The grouping construct is expressed using the symbols { Var\_Name } for set groupings, < Var\_Name > to group elements as bags and [ Var\_Name | a<sub>i</sub> : o<sub>i</sub>, ... a<sub>n</sub> : o<sub>n</sub> ] to group as lists, sorting the list according to the attributes a<sub>i</sub>, ..., a<sub>n</sub>, each attribute defining a letter (a=ascending, d=descending) for the ordering field o<sub>i</sub> that defines the order of the elements of the list according to that attribute. The following example shows the grouping of the relatives of a person as a list sorted in ascending order of the age attribute.

```
ancestor(X,Y) :- persons(X), X.father->Y.
ancestor(X,Y) :- persons(X), X.mother->Y.
ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).

relatives_of(X,[Y|age:a]) :- ancestor(X,Y).
```

### 2.1.2 Method Clauses

Method clauses are rules that specify deductive methods attached to classes. Method clauses extend the interface of the recipient object type and can be

used to define derived properties associated with classes. In method clauses, the head of the rule is of the form:

$$\langle \textit{recipient} \rangle :: \langle \textit{method} - \textit{name} \rangle (\textit{arg}_1, \dots, \textit{arg}_n)$$

where *recipient* is a variable, and *arg<sub>i</sub>* are as for regular clauses. For example, the following method rule reinterprets `ancestor` as a deductive method on `person`. In the deductive method definition, the typing information for variables is obtained from the class to which the deductive method is attached, instead of explicitly stating the extent over which the variable ranges, as was done in the example describing regular clauses.

```
P::ancestor(R) :- P.father->R.  
P::ancestor(R) :- P.mother->R.  
P::ancestor(R) :- P::ancestor(Z), Z::ancestor(R).
```

Deductive methods can be overridden along the sub-typing hierarchy, specializing the behaviour of a rule. The approach to overriding and late binding follows that of ROCK & ROLL [10]. In essence, methods can be overridden, and the definition that is used is the most specialised one defined for the object fulfilling the role of the message recipient.

## 2.2 Rule Management

Rules can be created either transiently, or in rule bases that are stored persistently in the ODMG compliant database. Rule bases are created using the `create_rulebase` operation, which takes as parameter the name of the rulebase to be created. The inverse of `create_rulebase` is `delete_rulebase`. These commands can be run from the operating system command line, from the interactive DOQL system, or from executing programs.

Once a rulebase has been created, rules can be added to it using the `extend_rulebase` command, which can also be run from the operating system command line, from the interactive DOQL system, or from executing programs. For example, figure 3 extends the rulebase `employment` with both regular and method rules.

The above environment for creating and storing rules has a role analogous to that of ODL for describing ODMG databases. We do not propose an extension to ODL, as implementation of such an extension requires access to the source of the underlying database.

## 2.3 Queries and Rule Invocation

Queries are evaluated over the current state of the database and rule base, with different query formats depending on the nature of the query. Rules can only be invoked directly from within DOQL queries and rules. The different formats are:

```

extend rulebase employment
{
  // Extend EMPLOYEE with an operation related_subordinate that associates
  // the EMPLOYEE with subordinates who are related to them.
  extend interface EMPLOYEE
  {
    Emp::related_subordinate(R) :-
      Emp::relative(R), Emp::subordinate(R).

    Emp::subordinate(S) :- Emp[subordinates=>S].
    Emp::subordinate(S) :- Emp[subordinates=>Int],
      Int::subordinate(S).

    Emp::relative(R) :- Emp::ancestor(Z), R::ancestor(Z), Emp != R.
  }

  // A dodgy manager is one who is the boss of a relative.

  dodgy_manager(M) :- employees(M), exists(S: M::related_subordinate(S)).
}

```

Figure 3: Extending a rule base.

- Single element queries (SELECT ANY): this form of query returns a single element from the set of elements that results from the evaluation of the goal. The choice of the element to be returned is non-deterministic.
- Set of elements queries (SELECT): this form of query returns all elements that satisfy the goal.
- Boolean queries (VERIFY): this form of query has a boolean type as result. The result is the value yielded by the boolean formula given as a parameter.

The query formats can be summarized according to the following general expressions:

```

SELECT [ANY] Result_Specifier
FROM DOQL Query
[WITH Rule_Statements]
[USING Rulebase]

```

```

VERIFY Boolean_Formula
[WITH Rule_Statements]
[USING Rulebase]

```

The `Result_Specifier` is a comma separated list of arguments that are the same as the arguments allowed in a rule head, and `Rulebase` is a comma separated list of persistent rulebase names. For example, the following query associates each employee with a set of their related subordinates, using rules from the `employment` rulebase.

```

SELECT E,{S}
FROM   E::related_subordinate(S)
USING  employment

```

The verification format requires that the operand of the verify returns a boolean value:

```

VERIFY exists(E:employees(E), E.age < 18)
USING  employment

```

The WITH clause is used to allow the specification of rules that exist solely for the purpose of answering the query (i.e. that are not to be stored in the persistent rule base). For example, the following query retrieves the names of the employees who have more than 5 related subordinates:

```

SELECT Name
FROM   employees(E)[name=Name], num_related_subordinates(E,Num), Num > 5
WITH   num_related_subordinates(E,count({S})) :- E::related_subordinate(S).

```

The above query forms can be used either in the interactive DOQL interface or in embedded DOQL.

## 2.4 Embedded DOQL

DOQL programs can be embedded in host programming languages in a manner similar to that used for OQL [4]. DOQL statements are embedded as parameters of calls to the API function `d_doql_execute` (`d_DOQL_Query Q, Type_Result`)<sup>1</sup>. The function receives two parameters. The first parameter is a query container object formed by the query form described in section 2.3 along with any input arguments to the query. The second parameter is the program variable that will receive the result of the query.

```

d_Ref <Person> adam = ...;
d_Set <d_Ref<Person>> *ancestors = new Set(Person);
d_DOQL_Query
    q1("SELECT Y
        FROM ancestor($1,Y)
        WITH
        ancestor(X,Y) :- persons(X), X.father->Y.
        ancestor(X,Y) :- persons(X), X.mother->Y.
        ancestor(X,Y) :- ancestor(X,Z),
                           ancestor(Z,Y).")
    );
q1 << adam;
d_doql_execute(q1, ancestors);

```

Figure 4: Example embedded DOQL code fragment

---

<sup>1</sup>A general syntax is used to show concepts. The specific syntax varies for each language binding (e.g. for C++, `template<class T> void d_doql_execute(d_DOQL_Query &query, T &result)`).

Figure 4 is a code fragment that shows the embedded form of DOQL for C++. First some variables are declared for the input parameter to the query (`adam`) and the result (`ancestors`). Then the query object `q1` is created with the query as the parameter of the constructor function. It is then indicated that `adam` is the (first and only) parameter of `q1` – this parameter is referred to within the text of the query as `$1`. Finally, the query is executed by `d.doql.execute`, and the results are placed in `ancestors`.

## 2.5 Stratification, Safety and Restrictions

DOQL adopts conventional restrictions on rule construction to guarantee that queries have finite and deterministic results. In particular: each variable appearing in a rule head must also appear in a positive literal in the rule body; each variable occurring as an argument of a built-in predicate or a user-defined ODMG operation must also occur in an ordinary predicate in the same rule body or must be bound by an equality (or a sequence of equalities) to a variable of such an ordinary predicate or to a constant; all rules are stratified; a rule head can have at most one grouping expression; the type of each argument of an overridden DOQL method must be the same as the type of the corresponding argument in the overriding method; all rules are statically type checked, using a type inference system.

## 3 DOQL Translation

The translation of DOQL to the object algebra is done following a two-step approach:

1. *Rewriting*: in this step, method clauses are rewritten as regular clauses and a normalisation process transforms a rule into a canonical form (DOQLc) where some types of nested queries are unnested and DOQL statements are rewritten as molecular expressions.
2. *Mapping*: in this step, DOQLc molecules are translated into an object algebra, capable of dealing with multiple collection types, grouping and aggregation.

The translation approach is highlighted in figure 5, which also shows the overall steps involved in the processing of a DOQL query. The operational semantic underpinned by the algebraic translation approach is useful in the sense that it provides a road to optimization and evaluation based on traditional algebraic query processing techniques.

### 3.1 Rewriting

In deductive object-oriented databases (DOODs), objects, classes and class members (structure and behaviour) play a central role in the organisation of

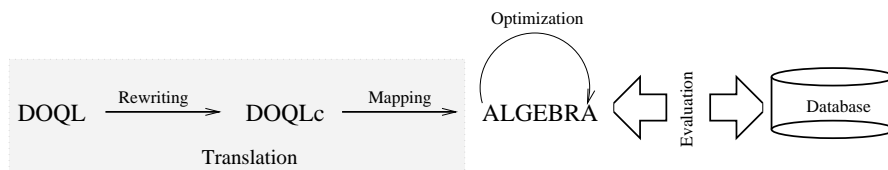


Figure 5: Translation Approach in DOQL Query Processing

information, contrasting with deductive relational databases where tuples, relations and attributes define the main abstractions.

Query processing technology for deductive databases is mainly targeted to the relational data model, which leaves to DOOD languages the options of conceiving new query processing techniques or rewriting the queries in a form that adapts the existing technology. The first approach is still an open research problem, while the latter approach has been employed in several systems and is also adopted by DOQL.

The rewriting process in DOQL is based on bundling the information directly related to collections in terms of molecules. Molecules are used to group formulas regarding properties and operations with the objects that belong to the collection. The molecular form can exploit classical query evaluation methods which are centered on the evaluation of variables and predicates. The following example shows a DOQL rule containing two molecules on the right hand side of the rule:

```
p_Kids(N1,N2) :- persons[name=N1,dependents=>Z],
                Z[age < 2, name=N2].
```

Although it is possible to encode certain deductive object-oriented database languages in terms of Datalog as a preliminary step of rule evaluation, this process can generate complex recursive patterns and drastically increase the number of predicates [18]. In DOQL, instead of completely breaking the object information structure as is done in the process of translating DOOD languages to Datalog, the notion of collection expressions organized in terms of DOQL molecules is kept during the query evaluation process, with method clauses transformed into regular clauses in order to simplify the translation. This latter transformation doesn't change the recursive patterns within the program.

Method clauses in DOQL have the form  $\langle recipient \rangle :: \langle method-name \rangle (arg_1, \dots, arg_n)$ , where  $recipient$  and  $arg_i$  are variables. Method clauses are transformed into regular clauses using the following rewrite rule:

$$\begin{array}{c} \langle recipient \rangle :: \langle method-name \rangle (arg_1, \dots, arg_n) \\ \longmapsto \\ method-name(recipient, arg_1, \dots, arg_n) \end{array}$$

The rewriting process also applies the normalisation operations *Associate* ( $\otimes$ ), *Compose* ( $\circ$ ), and *PullOut* ( $\odot$ ) introduced in [18] to obtain a canonical

form for DOQL. The symbol ( $\rightsquigarrow$ ) is used to denote a formula operator in DOQL (i.e.  $\rightarrow$  or  $\Rightarrow$ ).

The *Associate* operator binds the collection membership information between objects and collections:

$$\begin{array}{c} p(X), X[l_1 \rightsquigarrow p_1, \dots, l_n \rightsquigarrow p_n] \\ \xrightarrow{\odot} \\ p(X)[l_1 \rightsquigarrow p_1, \dots, l_n \rightsquigarrow p_n] \end{array}$$

The following example illustrates the application of the *Associate* rewrite to a DOQL expression:

```
persons(X), X[name = N, skills =>S[s_name = 'cycling', rating = R]]
```

$$\xrightarrow{\odot}$$

```
persons(X)[name = N, skills =>S[s_name = 'cycling', rating = R]]
```

The *Compose* operator bundles all formulas regarding properties and operations of an object into the same molecule:

$$p[l_1 \rightsquigarrow p_1], \dots, p[l_n \rightsquigarrow p_n] \xrightarrow{\otimes} p[l_1 \rightsquigarrow p_1, \dots, l_n \rightsquigarrow p_n]$$

The following example illustrates the application of the *Compose* rewrite to a DOQL expression:

```
persons(X)[name = N], X.skills => S[s_name = 'cycling', rating = R]
```

$$\xrightarrow{\otimes}$$

```
persons(X)[name = N, skills =>S[s_name = 'cycling', rating = R]]
```

In case of a formula describing relational properties such as equalities, inequalities, greater than, etc., between more than one object (variables of two different molecules appearing in the left and right hand sides of the relational operator), the *Compose* is not applied, as in the formula:

```
persons(X)[age < 21], departments(Y), X.surname = Y.name.
```

In the previous DOQL expression, the formula `X.surname = Y.name` relates to two objects appearing in different molecules. The *Compose* rewrite is not applied in this case. This is explained by the fact that in the mapping stage, formulas involving relational properties between objects of different molecules will appear as predicates of joins between the scans of the two extents corresponding to the molecules, while formulas that involve an object of a single molecule will be translated as filters of scan operations over a single extent. In the latter case, the *Compose* helps to bundle the predicate inside the molecule, simplifying the translation in terms of algebra.

The *PullOut* operation flattens a DOQL rule containing a nested collection expression.

$$\begin{array}{c}
 p[l_1 \rightsquigarrow p_1, \dots, l_i \rightsquigarrow p_i[l \Rightarrow v], \dots, l_n \rightsquigarrow p_n] \\
 \xrightarrow{\alpha} \\
 p[l_1 \rightsquigarrow p_1, \dots, l_i \rightsquigarrow p_i, \dots, l_n \rightsquigarrow p_n], p_i[l \Rightarrow v]
 \end{array}$$

The following example illustrates the application of the *PullOut* rewrite to a DOQL expression:

```
persons(X)[name = N, skills =>S[s_name = 'cycling', rating = R]]
```

$$\xrightarrow{\alpha}$$

```
persons(X)[name = N, skills => S], S[s_name = 'cycling', rating = R]
```

A *canonical* DOQL program is a DOQL program that has been normalised by the rewriting process.

### 3.2 Mapping

The rationale behind the use of an algebraic approach to DOOD query processing stems from the following points:

- A significant part of many DOOD queries involves nesting and path expressions which are mainly covered by research on algebraic approaches to OO query optimization.
- Logical rewriting techniques developed for deductive databases can be applied in the rewriting step and orthogonally complemented by the algebraic optimization stage.
- Where DBMS source code is available, the integrated architecture could benefit from an existing optimizer/evaluator implementation for OQL.

The object algebra underlying the DOQL translation approach is based on the algebras presented in [2, 8, 11]. The bulk operators of the algebra reflect the extent oriented nature of access to ODMG databases and also have a straightforward representation in terms of the algebraic interfaces supported by query optimization tools. The following bulk operators are supported:

**join(A,B,pred,outer)** the join operation joins two collections *A*, *B* using the predicate *pred*. The value of *outer* can be *false*, *left* or *right*, indicating if the join is a standard join or an outer join. The operation returns sets of tuples.

**select(A,a,pred)** the select operation returns the elements *a* of collection *A* such that *pred* holds.

**apply**(*A*, *a*, *f*(*a*)) the apply operation applies the function *f* to the elements *a* of collection *A*.

**unnest**(*E*, *path*, *v*, *pred*, *outer*) the unnest operation unnests the nested collection *E* by retrieving the *path* component into variable *v*. If *outer* is *false*, elements of the input which have a NULL path or that do not satisfy *pred* are not kept in the output.

**nest**(*E*, *group\_var*, *v*, *pred*) the nest operation creates a nested collection from *E*, by grouping each element in *E* according to the value of the variable appearing in *group\_var*. The groups formed relative to each value of *group\_var* are accessible through *v*. The variable in *group\_var* must have been previously defined in *E*.

**fixpoint**( $\Delta$ ) the fixpoint operator computes the fixpoint of a set  $\Delta$  of algebraic expressions.

**union**(*C1*, *C2*) the union operator merges the elements of two union compatible collections *C1* and *C2*.

The overall mapping process works by a left to right translation of the body of a DOQL canonical rule where each rule literal is mapped into a corresponding algebraic expression. The algebraic equivalent of a literal can be used as a parameter in algebraic expressions resulting from the translation of other DOQL literals. The last literal translated is the head of a rule.

Pre-processing steps to the mapping of a rule involve changing constants and replicated variables (with the generation of equalities) on the left-hand side of a rule, and also the generation of equality conditions due to the presence of replicated variables on the right-hand side of the rule. This process transforms the notion of repeated occurrences of the same variable (common in deductive languages) in terms of predicates that can serve as parameters to algebraic operations.

As a first example of the translation process, the following DOQL rule involves a join, retrieving the employees with a surname that coincides with the name of a department.

```
odd_surname(X) :- employees[surname=X], departments[name=X].
```

After the mapping step, the following algebraic expression represents the translation of the rule:

```
apply
(join (select(employees,v1,and(eq(project(v1,surname),x1))),
      select(departments,v2,and(eq(project(v2,name),x2))),
      and(eq(x1,x2)),
      false
    ),
a,
tuple(bind(!X,project(a,surname))
)
```

In the next example, consider the DOQL rule that groups the female dependents under 18 of the employees with surname “Smith”:

```
empDeps(X,<D>) :- employees(X)[surname='Smith'],
                 X.dependents=>D,
                 D.age < 18, D.sex = 'Female'.
```

After the normalisation step, the equivalent canonical form is as follows:

```
empDeps(X,<D>) :- employees(X)[surname='Smith', dependents=>D],
                 D[age < 18, sex = 'Female'].
```

After the mapping step, the following algebraic expression represents the translation of the rule:

```
apply
(nest
 (unnest
  (select(employees,x,and(eq(project(x,surname),'Smith'))),
   project(x,dependents),
   d,
   and(lt(project(d,age),18),eq(project(d,sex),'Female'))),
  true),
 x,
 v,
 and()
),
e,
tuple(bind(!X,x),bind(!D,v)),
)
```

The following example involves the translation of a recursive predicate that computes all direct and indirect supervisors of an employee.

```
super(X,Y) :- employees(X)[boss->Y].
super(X,Y) :- employees(X)[boss->K], super(K,Y).
```

After the mapping step, the following set of algebraic expressions represents the translation of the rule:

```
apply
(fixpoint
 (union(
  apply (select(employees,x,and(refchase(boss,y))),
   e,
   tuple(bind(X1,x),bind(Y1,y))),
  apply (join
   (select(employees,x,and(refchase(boss,k))),
    select(super,v1,and(project(v1,@1),project(v1,@2))),
    and(eq(k,@1)),
    false),
   E,
   tuple(bind(X1,x),bind(Y1,y)))
 ),
 E,
 tuple(bind(!X,X1),bind(!Y,Y1))
)
```

**Negation:** prior to the translation of a set of rules in a DOQL program, it is necessary to order the set of rules that form the program to reflect the stratification conditions. This affects the order of translation such that rules containing atoms in higher strata levels are translated after rules with atoms in lower strata levels.

Solutions for negated atoms are defined when all unbound goal arguments of the atom are of type object, meaning that their domain is finite. In such a case, the result for the negated atom can be obtained by subtracting the result of the non-negated atom from the cross-product of the unbound arguments' domains.

---

**Algorithm 1** Translation of Canonical Rule Body

---

**Require:** Canonical Rule Body (*CRB*):  $L_1, \dots, L_n$

$Result \leftarrow \emptyset$

**for all** Molecular Literals  $L_i \in CRB$  **do**

**if**  $L_i$  is a collection molecule **then**

**if** NotGivenVarInMolecule( $L_i$ ) **then**

      CreateVar( $L_i$ )

$\{L_i \text{ has the form } \text{ext\_name}(Var_i)[exp_1, \dots, exp_k, colexp_1, \dots, colexp_j]\}$

**end if**

$A_i \leftarrow \text{select}(\text{ext\_name}, Var_i, pred_i = \text{and}(exp_1, \dots, exp_k))$

**for all** nested collection expressions  $colexp_j \in L_i$ , of the form  $path_j \Rightarrow$

$var_j$  **do**

$B_k \leftarrow \text{unnest}(A_i, \text{project}(Var_i, path_j), Var_j, pred_j)$

$\{pred_j \text{ is formed by the conjunction of the expressions appearing in the molecule } Var_j \dots\}$

**end for**

**else if**  $L_i$  is an object molecule **then**

**for all** nested collection expressions  $colexp_j \in L_i$ , of the form  $path_j \Rightarrow$

$var_j$  **do**

$B_x^y \leftarrow \text{unnest}(B_k, \text{project}(Var_i, path_j), Var_j, pred_j)$

$\{pred_j \text{ is formed by the conjunction of the expressions appearing in the molecule } Var_j \dots\}$

**end for**

**end if**

$Result \leftarrow \text{generate joins between all } B_x^y, \text{ and all } B_x \text{ not appearing as a parameter of a } B_x^y$

**end for**

**for all** Non Molecular Literals  $L_i \in CRB$  **do**

  Place the formulas defined by the literals as parameters of the join predicates between the algebraic expressions resulting from the translation of the two molecules involved in the literal

**end for**

---

The general steps of the algorithm that maps the body of a canonical DOQL rule into a nested relational algebra expression are described in algorithm 1.

If there are no grouping constructs, the rule head is translated into an *apply* operator, otherwise, the rule head is translated into a *nest* operator followed by an *apply*.

In the mapping process, the following operators supporting path expressions and built-in predicates are used:

**project(*a,path*)** the project operator retrieves the *path* component of the element *a*. The *path* component can be a position (n) of a field of the element belonging to a virtual collection (positional approach akin to Datalog) or the name of a property defined for an element belonging to a stored collection (named approach to attribute denotation).

**eq(*a,b*)** the value equality operator returns true if the values *a* and *b* are equal.

**lt(*a,b*)** the less than operator returns true if the value *a* is less than *b*.

Other operators regarding object equality, relational built-in predicates, aggregations, and arithmetic predicates are also supported in the underlying target implementation algebra, defining the fully-fledged algebraic interface to the database engine.

## 4 Related Work

This section outlines a range of features that can be used to compare the query components of proposals that integrate declarative query languages and imperative OO programming languages in the context of OODBs.

**Bidirectionality of calls:** relates to the flexibility of the integration approach. In unidirectional systems, one of the languages can call the other, but not vice-versa.

**Restructuring:** relates to the operations available in the declarative language that can be used to reorganize the queried data elements to yield more desirable output formats.

**Type system:** relates to the underlying type system used across the integrated languages.

**Query power:** relates to the query classes supported by the query language (FOLQ = first-order queries, FIXP = fixpoint queries, HOQ = higher-order queries).

Table 1 shows a comparison between the query facilities supported by DOQL and the query components of Chimera [6], Coral++ [16], OQL [4], OQLC++ [3], Noodle [12] and ROCK & ROLL [1].

Some important aspects of our design are:

Language	Criteria			
	Bidirection of Calls	Restruct. Operators	Type System	Query Power
Chimera	no	unnest	Chimera object model	FOLQ, FIXP
Coral++	no	set group., unnest	C++	FOLQ, FIXP
DOQL	yes	(set,bag,list) group., unnest	ODMG object model	FOLQ, FIXP
Noodle	yes	(set,bag) group., unnest	Sword object model	FOLQ, FIXP, HOQ
OQL	yes	(set,bag,list) group., unnest	ODMG object model	FOLQ
OQLC++	yes	unnest	C++	FOLQ
ROCK & ROLL	yes	unnest	Semantic object model	FOLQ, FIXP

Table 1: Query languages for object databases

- language integration is done using a standard object-model as the underlying data model of the deductive system.
- the approach to integration complies with the call level interface defined by the ODMG standard, reusing existing compiler technology for other ODMG compliant languages and providing portable deduction within the imperative languages without the need for changes in syntax or to the underlying DBMS.
- bidirectionality of calls between the integrated languages.
- powerful aggregate and grouping operators that can be used for restructuring data in applications that require summaries, classifications and data dredging.
- updating and control features are confined within the imperative language

Language integration proposals can also be compared based on the seamlessness of the language integration (see [1]) or based on the object-oriented and deductive capabilities supported (see [15]). Using the criteria of [1], DOQL can be seen to support evaluation strategy compatibility, (reasonable) type system uniformity and bidirectionality, but to lack type checker capability and syntactic consistency. The lack of type checker capability means that embedded DOQL programs can cause type errors at runtime – obtaining compile time type checking of embedded DOQL would require changes to be made to the host language compiler. The lack of syntactic consistency is unavoidable, as DOQL can be embedded in any of a range of host languages, and thus cannot hope to have a syntax that is consistent with all of them.

## 5 Summary

DOQL is a deductive query language designed to complement existing query language facilities for ODMG databases that is being implemented on top of the Poet SDK 5.0 C++ ODMG binding [14].

In this paper, the translation approach adopted in the query processing of DOQL has been presented. The paper addresses the rewriting and mapping steps that translate DOQL rules involving conjunctive queries, recursion, negation and grouping into algebraic expressions that are evaluated against the underlying object base. The query transformation technique used in the rewriting step together with the functionality supported in the algebraic operators enable the early application of constraints such as inequalities and equalities, limiting the amount of data generated in the query processing flow.

The choice of a combined approach (logical + algebraic) to the translation of DOQL allows the exploitation of deductive optimization techniques and object-oriented optimization techniques, as well as laying the foundation for the use of query optimization tools like [11] to optimize the algebraic expressions.

**Acknowledgements:** The first author is sponsored by Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Brazil) - Grant 200372/96-3.

## References

- [1] M. L. Barja, N. W. Paton, A. A. Fernandes, M. Howard Williams, and Andrew Dinn. An effective deductive object-oriented database through language integration. In *Proc. of the 20th VLDB Conference*, pages 463–474, 1994.
- [2] C. Beeri and T. Milo. Functional and predicative programming in oodb's. In *Proc. of the ACM Principles of Database Systems Conference (PODS 92)*, pages 176–190, 1992.
- [3] J. Blakeley. OQLC++: Extending C++ with an object query capability. In Won Kim, editor, *Modern Database Systems*, chapter 4, pages 69–88. Addison-Wesley, 1995.
- [4] R. Cattell and Douglas Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufman, 1997.
- [5] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.
- [6] S. Ceri and R. Manthey. Consolidated specification of chimera (cm and cl). Technical Report IDEA.DE.2P.006.1, IDEA - ESPRIT project 6333, 1993.
- [7] M. Chavda and P. Wood. Towards an odmg-compliant visual object query language. In *Proc. of the VLDB Conference*, pages 456–465, 1997.

- [8] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, pages 383–392, 1992.
- [9] M. A. Derr and S. Morishita. Design and implementation of the glue-nail database system. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, pages 147–156, 1993.
- [10] A. Dinn, N. W. Paton, M. Howard Williams, A. A. Fernandes, and M. L. Barja. The implementation of a deductive query language over an OODB. In *Proc. 4th Intl. Conference on Deductive and Object-Oriented Databases*, pages 143–160, 1995.
- [11] Leonidas Fegaras. An experimental optimizer for OQL. Technical Report TR-CSE-97-007, CSE, University of Texas at Arlington, 1997.
- [12] I. S. Mumick and K. A. Ross. Noodle: A language for declarative querying in an object-oriented database. In *Proc. of the Third Intl. Conference on Deductive and Object-Oriented Databases*, volume 760 of *LNCS*, pages 360–378. Springer-Verlag, 1993.
- [13] N. Murray, N. Paton, and C. Goble. Kaleidoquery: A visual query language for object databases. In *Proc. of the Working Conference on Advanced Visual Interfaces - AVI*, 1998.
- [14] Poet Software. *Poet V5.0 ODMG Programmer's Guide*, 1997.
- [15] Pedro R. F. Sampaio and Norman W. Paton. Deductive object-oriented database systems: A survey. In *Proceedings of the 3rd International Workshop on Rules in Database Systems*, volume 1312 of *LNCS*, pages 1–19. Springer-Verlag, 1997.
- [16] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proc. of the 19th VLDB Conference, Dublin, Ireland*, pages 158–170, 1993.
- [17] K. Subieta. Object-oriented standards: can ODMG OQL be extended to a programming language. In *Proc. of the International Symposium on Cooperative Database Systems for Advanced Applications*, pages 546–555, Kyoto, Japan, 1996.
- [18] Z. Xie and J. Han. Normalization and compilation of deductive and object-oriented database programs for efficient query evaluation. In *Proc. of the 4th Intl. Conference on Deductive and Object-Oriented Databases (DOOD'95)*, pages 485–502, 1995.