0.1 The Query Language OQL

The query language OQL is part of the ODMG Standard [**?**, **?**, **?**]. Although the standard itself is constantly evolving, it seems that the query language OQL has reached a rather fixed state. The language described in this section is based upon the latest edition of the ODMG Standard [**?**].

0.1.1 Basic Principles

The OQL is a declarative query language whose design is based upon a few basic principles. The main and most important design principle is the orthogonality of building expressions. Several basic expressions like constants and named objects are introduced. These basic expressions can be used to build more complex expressions by applying functions. Contrary to SQL-92 [**?**], the way how more complex expressions can be build is restricted by typing rules only: as long as the typing rules are obeyed, the expressions forms a legal query. Thereby, OQL relies on the ODMG Object Model [**?**].

Another intent of the designers is to make OQL as similar as possible to SQL. Ideally, OQL should be an extension of SQL. However, this goal is not easily reached.

OQL is not only an ad-hoc declarative query language but can also be embedded into programming languages for which an ODMG binding is defined. Using the same type system in the data base and in the programming languge enables this feature. For example, the result of a query can be assigned to a variable, if typing rules are obeyed.

0.1.2 Simple Queries

Every constant is a already a query. Hence,

5
"Jeff"

are already perfect queries returning the values *5* and *"Jeff"* respectively. If a named object *Dean* exists, then

Dean

is also a perfect query returning the object referenced by *Dean*.

The attributes of named objects are also directly accessible via queries:

Dean.spouse.name

retrieves the name of the spouse of the dean via a path expression. The query

Dean.subordinates

retrieves the value of the set-valued attribute *subordinates* of the *Dean*. It contains all the faculty staff of the department of the *Dean*.

Not only prefabricated objects can be retrieved from the database. It is also possible to construct objects in a query. For example, the following query creates a new *Student*:

Student(firstname: "Jeff", lastname: "Ullman", major: "Computer Science")

The parameters within the parentheses allow to initialize certain attributes of the newly created object. The result of the above query is a new *Student* object whose first name is *"Jeff*.

For the construction of tuple-structured values, the keyword **struct** is used. For example, we can construct a tuple value consisting of the three fields *year*, *month*, *day* by the following query:

struct(year: 1999, month: 12, day: 31)

More complex queries can be build by applying built-in or user defined functions:

5 + 5 - 10
Dean.age() + 500

The latter query invokes the method *age* on *Dean* which computes the the *Dean*'s age from the attribute *dateOfBirth*. OQL allows to skip parenthesis for methods without parameters. Hence,

Dean.age + 500

is equivalent to the above query.

Assume that the the *spouse* attribute of the *Dean* is not defined, i.e. contains a *nil* value. Then, the path expression

Dean.spouse.name

produces the special value *UNDEFINED*. In general is any property of the *nil* object *UNDEFINED*. Any comparison (e.g. with =, <) produces *false* if at least one of the compared values is *UNDEFINED*. There exists a special function *is_undefined* to check whether some value is undefined. By

is_undefined(Dean.spouse.age)

we could check whether the path expression returns a legal result. Applying any function other than a comparison operator on an undefined value results in run-time error. Hence,

Dean.spouse.age + 5

will result in a run-time error.

Not only simple values and tuple-structured values can be constructed in a query but also collections. The following three queries construct a bag, a set, a list, and an array respectively:

bag(1,1,2,2,3)
set(1,2,3)
list(1,2,3)
array(1,2,3)

There exist some special built-in functions called aggregates. They are *min*, *max*, *count*, *sum*, and *avg*. These functions are all unary functions and they are applied to a collection. Except for *count* which can be applied to any collection, the other aggregate functions can only be applied to collections of numbers. For example, the function *min* computes the minimum of a collection of numbers, the function *avg* computes the average of collection of numbers. The query

count(Student)

returns the number of students for the set *Student*.

0.1.3 Queries with Select-From-Where-Blocks

As SQL, OQL provides a select-from-where-block. The following query retrieve all first year students:

select s
from Student s
where s.year = 1

The variable *s* is bound successively to all students contained in the set *Student*. For each student *s*, the predicate contained in the **where** clause is checked. Those students whose *year* property returns 1 are selected. The result of the query is a bag of students. The set *Student* is called the domain of the (iterator) variable *s*. We call a query or subquery consisting of a **select**, **from**, and **where** clause a SFW-block.

In order to retrieve a set of students, the keyword **distinct** must be applied:

select distinct s
from Student s
where s.year = 1

Whenever **distinct** is applied, possible duplicates are eliminated and the result is a set. In the above case, the result is a set of students.

In OQL there exist three alternative possibilities to specify in the **from** clause that a variable ranges over a collection:

select distinct s
from Student s
where s.year = 1

select distinct s
from s **in** Student
where s.year = 1

select distinct s
from Student **as** s
where s.year = 1

All these queries are equivalent.

As in SQL, an entry in the **from** clause can have more than one entry in order to join collections. With the next query we find all professors and students having the same name:

select struct(prof: p, stud: s)
from p **in** Professor, s **in** Student
where p.name = s.name

Note the usage of *struct* in the **select** clause. It is used to construct the result records. Other than in SQL, it is not allows to have multiple entries in the **select** clause. In SQL, a result tuple is implicitly constructed. In OQL, the result tuple has to be constructed explicitly (see also Sec. 0.1.7).

Quantifiers can be used independent of a SFW-block. They return a boolean value. The first query asks for professors earning more than $100.000, the second ask whether all students passed the database course:

```
exists p in Professor: x.salary() ¿ 100.000
for all s in Student: databaseCourse in s.passedCourses()
```

The special predicate **in** tests for membership and *databaseCourse* is assumed to be a named object referencing the database course.

Nesting of SFW-blocks can occur in many places within quantifiers. Assume that we do not have a named object *databaseCourse* and that we want to know whether all students named "Smith" passsed the database course. Then we might query:

```
for all s in select  s
             from    s in Student
             where   s.name = "Smith":
        for all c in select  c
                     from    c in Course
                     where   c.title = "database")
                 c in s.passedCourses()
```

Quantifiers can also occur within the **where** clause where they play the role of a selection predicate. The following query retrieves all students that passed the database course:

```
select  s
from    s in Student
where   for all c in select  c
                     from    c in Course
                     where   c.title = "database")
                c in s.passedCourses()
```

This query is a little awkward. It could be easier stated if we could directly ask whether a given the database courses are a subset of the passed courses. This would save the universal quantifier. In fact, universal quantifiers can often be replaced by the subset predicate and vice versa. The alternative formulation of the query is:

```
select  s
from    s in Student
where   (select  c
         from    c in Course
         where   c.title = "database")
        <=       c in s.passedCourses()
```

Here, <= denotes the subset predicate. Likewise, >= denotes the superset predicate, = denotes set equality. The comparison operators < and > can be used if we test for strict subsets or strict supersets.

As can be seen from the above queries, SFW-blocks can occur nested within SFW-blocks. In fact, in OQL SFW-blocks can be nested anywhere (in the **select**, **from**, and **where** clause) as long as the typing rules are obeyed. The following three queries demonstrate nesting in different places:

```
select  struct(stud: s, courseNames: select  c.name
                                     from    c Course
                                     where   c in s.passedCourses())
from    s in Student
where   s.name = "Smith"
```

This query retrieves students named “Smith” the names of all courses passed. Note that a reference to the student *s* occurs in the inner block nested in the **select** clause. It occurs in the so-called correlation predicate *c* **in** *s.passedCourses()* that correlates a student with the passed courses.

Often it is necessary to select the best. For example, we would like to query the best students. By definition, the best students are those with the highest *gpa*. The following query retrieves the best students by applying nesting in the **where** clause.

```
select  s
from    s in Student
where   s.gpa = max(select  s.gpa
                    from    s Student)
```

Additionally, the query demonstrates a typical application of an aggregate function (*max*) and shows a block without a **where** clause. The **where** clause is optional and can be omitted, not only in nested blocks.

The next query demonstrates nesting in the **from** clause. It can be used to restrict the domain a variable iterates over.

```
select  s
from    s in select  s
             from    s in Student
             where   s.gpa > 10
where   s.supervisor = dean
```

The nested query retrieves all students whose *gpa* is greater than 10. From these, those are selected whose supervisor is the dean. Obviously, the query could be stated much simpler by applying the boolean connective **and** as done in the next query:

```
select  s
from    s in Student
where   s.supervisor = dean and s.gpa > 10
```

Besides **and** the other boolean connectives **or** and **not** are available in OQL. Common boolean expressions built from these connectives and base predicates can be built and used as selection predicates.

Sometimes it is preferable to express a query with collection operations instead of boolean connectives. The last query can equivalently be stated as follows:

```
    select  s
    from    s in Student
    where   s.supervisor = dean
intersect
```

```
    select  s
    from    s in Student
    where   s.gpa > 10
```

where **intersect** denotes intersection. The other supported collection operations are **union** and **except**. The latter denotes set minus and is applied in the following query:

```
    select  s
    from    s in Student
    where   s.gpa > 10
except
    select  s
    from    s in Student
    where   s.supervisor = dean
```

It retrieves the good students not supervised by the dean.

0.1.4 Grouping and Ordering

Grouping in OQL looks little more complex than in SQL. In general it works as follows. First, the **from** clause and the **where** clause of the SFW-block involved in the query is evaluated. Typically, this can be done by taking the cross product of the result of the expressions in the **from** clause. Then, the selection predicate of the **where** clause is evaluated. Second, this result is split into different partitions. For each partition, there will be one tuple. All but one of the attributes correspond to the properties used for grouping. The last attribute is always called *partition* and is set-valued. Each set contains the result elements from the step that belong to the according group. Third, unwanted groups can be eliminated by a predicate given in the **having** clause.

Let us consider an example. We want to group students into good, mediocre, and bad students according to their *gpa*. The following query does exactly this:

```
    select    *
    from      s in Student
    group by  good:      s.gpa >= 10
              mediocre:  s.gpa < 10 and s.gpa >= 5
              bad:       s.gpa < 5
```

In order to understand this query, it is useful to look at the result type. The result type is

```
    set<struct(good: bool, mediocre: bool, bad: bool,
               partition: bag<struct(s: Student)> )>
```

The result contains three tuples whose values first three values are:

1. struct(good: true, mediocre: false, bad: false, partition: ...)
2. struct(good: false, mediocre: true, bad: false, partition: ...)

3. struct(good: false, mediocre: false, bad: true, partition: ...)

This is due to the fact that for the partition attributes (*good*, *mediocre*, *bad*) three possible value assignments exist: the expressions following the partition attributes are boolean expressions such that for every student exactly one predicate results in true. Of course, such a restriction does not exist in OQL but queries of the above kind are rather typical.

For each combination of values for the partition attributes, the students exhibiting this value combination are collected in the *partition* attribute.

Grouping all students by their advisor results in more than a single value for the partition attribute:

```
select     *
from       s in Student
group by   adv:    s.advisor
```

Restricting the result tuples to those whose advised student group has a good *gpa* is managed by applying a **having** predicate:

```
select     *
from       s in Student
group by   adv:    s.advisor
having     avg(select p.s.gpa from p in partition) >= 10;
```

Results—with or without grouping—can be order by applying the **order by** clause. Assume that in the above query we would like to retrieve the average *gpa* and order the result by the decreasing average gpa. This can be done as follows:

```
select     adv.name, avggpa: avg(select s.gpa from partition),
from       s in Student
group by   adv:    s.advisor
having     avg(select p.s.gpa from p in partition) >= 10;
order by   avggpa desc
```

The annex **desc** states that we want to order by descending average *gpa*. Ordering by an increasing value is specified by **asc** which is also the default if no order specifier is given. In general, a list of expressions can be used as an order specification.

0.1.5 Views

OQL supports simple views and views with parameters that behave like functions, often returning a collection. For example, if we are often interested in good students, we might define a view *GoodStudent*

```
define GoodStudent as
select   s
from     s in Student
where    s.gpa >= 10
```

and to refer to them in another query:

select s.name
from s **in** GoodStudent
where s.age = 25

Views are persistent. That is, they are stored permanently in the schema until they are explicitly deleted:

delete definition GoodStudent

In OQL, views are not called views but they are called named queries. A named query can take parameters. An example is a named query that retrieves good students where the measure of what's good and what's not is given as a parameter:

define GoodStudent(goodGPA) **as**
select s
from s **in** Student
where s.gpa >= goodGPA

The syntax for referencing named queries with parameters is the same as the syntax for function calls:

select s.name
from s **in** GoodStudent(10)
where s.age = 25

It is important to note that names for named queries cannot be overloaded. Whenever the same name occurs for a named query, the old definition is overwritten.

0.1.6 Conversion

OQL provides for a couple of conversions. A collection can be turned into a single element by the *element* operator. If the argument of *element* contains more than a single element, *element* raises an exception. For example,

element(**select** s **from** s **in** Student **where** s.name = "Smith")

results in an exception, if there is more than one student named "Smith"; otherwise the single student named "Smith" is returned.

Other conversion operators are concerned with the conversion between different collection types:

listtoset converts a list into a set and

distinct converts a bag into a set.

The special conversion operator *flatten* flattens a collection of collections into a collection by taking the union of the inner collections. In case of lists, the union is the append operator. For example, the result of

flatten(list(list(1,3),list(2,4)))

is

list(1,3,2,4)

0.1.7 Abbrevations

OQL contains a couple of possible abbreviations—or syntactic sugar—that makes OQL look more like SQL. The first important construct allows to omit the explicit construction of tuples in the **select** clause. OQL allows for queries with multiple entries in the **select** clause:

select $p_1, \ldots p_n$
from ...
where ...

where p_i are projections of the form

1. expression$_i$ **as** identifier$_i$

2. identifier$_i$ **:** expression$_i$

3. expression$_i$

Such a query is equivalent to

select struct(identifier$_1$: expression$_1$, ... identifier$_n$: expression$_n$)
from ...
where ...

In the third case, an anonymous identifier is chosen by the system.

Let us consider an example query where we want to select the names and ages of all good students:

select s.name, s.age
from Student s
where s.gpa > 10

This query does not look different from an SQL query. If we want to give names to the projected expressions, we write

select s.name **as** studentName, s.age **as** studentAge
from Student s
where s.gpa > 10

A second kind of syntactic sugar concerns aggregate functions. Instead of writing

aggr(**select** expression
from ...
where ...)

we can write the SQL-like query

select *aggr*(expression)
from ...
where ...)

for any aggregate function *min*, *max*, **avg**, and *sum*.

The query

```
select  count(*)
from    ...
where   ... )
```

translates into

```
count(select  *
      from    ...
      where   ... )
```

The same abbreviations apply to SFW-blocks exhibiting a **distinct**.

SQL allows to compare a single value via a comparison operator (=, <, <=, ...) and a quantifier (**some**, *all*) with a whole set of elements. The same applies to OQL. For example,

```
select  s
from    Student s
where   s.gpa >= all (select  s1.gpa
                      from    Student s1)
```

is a perfect OQL query. It retrieves all students whose *gpa* is greater or equal than all *gpa*'s found for students. This query is equivalent to

```
select  s
from    Student s
where   for all s1 in Student:
               s.gpa >= s1.gpa
```

For a comparison operator $\theta \in \{=, <, >, <=, >=, != \}$, the predicate

expression θ **some** setexpression

is equivalent to

exists x **in** setexpression: expression θ x

and the predicate

expression θ **all** setexpression

is equivalent to

for all x **in** setexpression: expression θ x