Querying Object-Oriented Databases

Michael Kifer*
University of Toronto

Won Kim
UniSQL, Inc., Austin, TX

Yehoshua Sagiv
The Hebrew University of Jerusalem

1 Introduction

Recent years saw several proposals for languages for querying object-oriented databases [2, 3, 4, 5, 13, 6]. None of them, however, captures (or even attempts to deal with) all the aspects of the object-oriented model. In this paper, we present a new query language, henceforth referred to as XSQL, that incorporates features not found in earlier languages. XSQL is easier to use and it has more expressive power than previous languages. It is not our goal here to give the full-fledged syntax and semantics of XSQL. Rather, we use the familiar SQL-like syntax to illustrate certain philosophy in designing object-oriented languages—a philosophy put forward in [12, 10, 13, 8]. Before discussing the novelties found in XSQL, we should point out some of the differences between the object-oriented model and the relational model.

The different features of these two models induce different modes of representing information and querying it. Suppose that a database includes information about engines and their types (e.g., turbo engines, diesel engines, etc.). In a relational database, there would likely be an attribute *EngineType* having the various engine types as its possible values. In an object-oriented database, there would likely be a class *Engines* having the various engine types as subclasses. This is a fundamental difference, because it shifts the information about engine types from the data to the schema. For example, suppose we want to know what are all the engine types. In the relational model, we simply project onto the attribute *EngineType*. In the object-oriented model, we have to interrogate the schema rather than the data (and there is hardly any language for doing that). This shows the need for features not available in relational query languages. In particular, as an object-oriented schema is likely to have much more information than a relational schema, querying the schema (as well as data without a complete knowledge of the schema) becomes an important issue. We also need to deal easily with nested structures.

XSQL provides these (and other) features through *path expressions*. Although the idea of path expressions is not new (it first appeared in [16] and had many incarnations since), our *extended* path expressions have the following features and expressive power not found in earlier incarnations of this idea.

1. Path expressions may have variables that range over classes, attributes, and methods, and hence it is possible to query data without a complete knowledge of the schema. (Earlier query languages for object-oriented databases completely lack any similar feature.) In spite of variables that range over classes and methods, the language is still first-order, since it is based on F-logic [8].
2. Path expressions may have *selectors* that select data or some part of the schema (from which data is to be retrieved).
3. Path expressions treat attributes and methods in a uniform way, more general than just composing methods as function applications (as found in functional query languages).
4. Path expressions "flatten" any nested structure in one sweep—no need to break paths in the schema into several expressions and apply a "collapse" operator to each one.

All the above features increase the expressive power of XSQL and also make it easier to read and write queries. In many cases, queries can be expressed as one simple path expression, while in earlier proposals the same queries could be expressed only by using several path expressions and/or nested subqueries. Path expressions are discussed in Sections 3 and 5.

XSQL has a powerful viewing mechanism discussed in Section 4. When creating new objects, we follow [10] and invent new object identifiers by applying function symbols to existing object identifiers. This approach circumvents the problems with assigning id's to "imaginary objects" discussed in [1]. As in the relational model, views in XSQL are constructed via queries, which is simpler and more uniform than in other proposals.

Typing is a cornerstone of the object-oriented model. Earlier languages, however, hardly discussed the question of when a query is well-typed. The following example may help crystalize some of the problems. Consider a database that has information on the winners of Nobel Prizes. Let there be an attribute *WonNobelPrize* that, for a given object, specifies the area(s) in which that object won the prize. Suppose we want to find all winners of Nobel Prizes. The problem is that winners are not necessarily members of one class. Generally, they could be persons or organizations of various types.[1] It is unlikely that a casual user would know exactly all the classes in the database for which *WonNobelPrize* is defined. Nevertheless, in XSQL one may simply write the query

```
SELECT X   WHERE X.WonNobelPrize
```

*Work supported in part by the NSF grants IRI-8903507 and CCR-9102159. On sabbatical leave from Stony Brook University.

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0393...$1.50

[1] For example, UNICEF (United Nations International Children's Emergency Fund) won the Nobel Peace Prize.

which would return all objects for which *WonNobelPrize* is defined and its value is nonempty. It is not clear, however, whether this query should be considered as well-typed.[2] Obviously, if we allow queries of this form, the expressive power of the language is enhanced considerably. However, too much expressive power may violate the principle behind typing, and might result in unexpected answers (to ill-conceived queries) rather than type errors. In Section 6, we discuss typing and outline a spectrum of approaches between a conservative approach that considers the above query as ill typed, and a liberal approach that considers it as well-typed. The conservative approach does not really permit a query about winners of Nobel Prizes without specifying the classes for which `WonNobelPrize` is defined. This raises the need for querying the schema (rather than the data). Some of this is discussed in Section 3, but more information can be found in [9].

2 Data Model Review

First we review the object-oriented data model used throughout this paper, primarily derived from [8].

Objects and object identity. Objects are abstract or concrete entities in the real world. The programmer refers to objects via *logical object ids*, which are nothing but syntactic terms in the query language. For instance, *_324*, *johnP23*, *secretary(dept77)* are logical object ids. We follow [10, 7, 8] and use explicit id-functions (such as *secretary* above) to get our hands on a sufficient supply of such ids. These will primarily be used in conjunction with the view mechanism. Any logical oid uniquely identifies an object. However, unlike most approaches (that confuse the implementational and conceptual issues) we do not require an object to have a unique id at the logical level. For instance, *_mary65* and *secretary(dept77)* may refer to the same object.

Physical object identity is a purely implementational notion—a surrogate or a pointer to an object. Logical oid's can be implemented as physical oid's, but unlike physical pointers logical oid's may carry certain semantic information. For instance, '20' is a logical id of the abstract object with the usual properties of the number 20. Likewise, "Ford Motor Co." is a logical id of the object with the usual properties of a string of characters 'F', 'o', 'r', 'd', ' ', etc., in that order. We shall be using the words "object identity" or even just "object" to refer to ids at the logical level.

Attributes. Objects are described via attributes. An attribute may be either *defined, undefined,* or *inapplicable* to an object. If an attribute is *defined* for an object then it also has a *value*; otherwise, it has no value. If an attribute is inapplicable to an object then it is also undefined, but undefinedness does not imply inapplicability. Inapplicability captures the idea of *type error*—a situation when an attribute is used in the scope of an object to which it does not apply. Undefinedness is analogous to relational null values. Typing will be formally taken up in Section 6.

Our model does not divide the world into set-objects and tuple-objects—all our objects are tuple-objects. Each entry in a tuple-object is the value of one attribute. If the attribute is *scalar,* then the value is a single object id; if the attribute is *set-valued,* then the value is a set of object id's. Set-objects are described in our model as tuple-objects having a single, set-valued attribute. This approach achieves more uniformity than other proposals and modeling sets of arbitrary nesting depth is easy [10].

Following [7, 8], any logical oid, depending on its syntactic position in a query, may play the role of an attribute or of an object. Theoretical underpinnings of this approach appear in [8]; its practical impact—as we shall see—is that the user can now query the structure of the database in a very natural way, without knowing the system tables that represent the database schema.

[2] It could be argued that since the type of X is not declared, the query is not well-typed.

Classes. Classes organize objects into sets of related entities. However, classes are also objects and thus may have attributes and can be queried just as regular objects. To distinguish ordinary objects from classes, we shall call the former *individual* objects or just *individuals*.

There is a pair of special binary relationships, *instance-of* and *subclass*. *Instance-of* is defined between individuals and classes; it determines which individuals belong to which classes. *Subclass* (also called *IS-A*) is defined between classes and is acyclic. If a class C is a subclass of C', then all instances of C also belong to C'. But the converse is not necessarily true: if at some point the only students registered in the database are TA's, this does not make the class *Student* a subclass of the class *TA*. Treating classes as objects achieves a great deal of uniformity, allows querying the class hierarchy and eliminates the need for metaclasses.

Methods. A method is a pair consisting of a symbol, called the *name* of the method, and a partial function, called the *implementation* of the method. When confusion does not arise, we will use the term "method" to refer either to the name or to the implementation of a method, depending on the context. When invoked in the scope of an object on a tuple of arguments, a method returns an answer and, possibly, changes the internal state of that object (e.g., by changing the value of an attribute). As a function, each method has *arity*—the number of its arguments.

XSQL treats attributes as 0-ary methods. Thus, method names, like attribute names, are logical oid's and therefore can be returned as query answers. We assume that the universe of class-objects is disjoint from the universes of individuals and of method-objects. Like attributes, methods can be scalar or set-valued, depending on the kind of result they return. Methods can also be undefined or inapplicable. Section 6 takes care of the details.

Types. In object-oriented languages, the abstract values of interest are objects; types provide one of the important means of classifying objects. Class hierarchy discussed earlier is another means of classification. While types are generally used to classify objects by *structure*, objects are grouped into classes based on *semantic* criteria. Often—if not about always—instances of the same class share common structural features. Thus, grouping objects into classes implies typing but not the other way around. This suggests that the concept of a class should be the primary means of classification in object-oriented languages.

The type of a class is determined by the types of its methods/attributes. The type of a method in a class C is described as a *signature* of the form

$$\texttt{Mthd} : \texttt{Arg}_1, \ldots, \texttt{Arg}_k \Rightarrow \texttt{Result} \quad \text{or}$$
$$\texttt{Mthd} : \texttt{Arg}_1, \ldots, \texttt{Arg}_k \Rightarrow\!\!> \texttt{Result}$$

that is attached to the definition of C, where Arg_i and *Result* are class names. The single arrow, $\Rightarrow$, is used in the declarations of scalar methods, while the double arrow, $\Rightarrow\!\!>$, is used for set methods. Since attributes are 0-ary methods, they are covered by the above definition. For aesthetic reasons, we write their signatures as $attr \Rightarrow class$ (or $attr \Rightarrow\!\!> class$) instead of $attr :\Rightarrow class$ (resp., $attr :\Rightarrow\!\!> class$).

The above signature is meant to say that when the method *Mthd* is passed arguments that are instances of classes $Arg_1, \ldots, Arg_k$, respectively, the result is expected to be an instance or a set of instances of the class *Result*, depending on whether *Mthd* is scalar or set-valued. Note that there are actually $k+1$ (rather than k) arguments, since the method is invoked in the scope of some object, and that object could be viewed as the 0^{th} argument. However, the class of the 0^{th} argument is the one for which the signature is defined, and hence it is redundant to include the 0^{th} argument in the signature.

A method can have several signatures, each constraining the behavior of the method on different sets of arguments. When this is

the case, the method is said to have *polymorphic* type. A method can also have different signatures for the *same* type of arguments. For instance, suppose both *workstudy* : *semester* $\Rightarrow\!\!>$ *student* and *workstudy* : *semester* $\Rightarrow\!\!>$ *employee* are specified for the class *department*. This states that *workstudy* is a unary method that, when invoked in the scope of an instance of class *department* with an argument of class *semester*, returns a set of this department's work-study students in the given semester. Besides being instances of class *student*, these must also be instances of *employee*. When more than one signature is specified in this way we can save writing by combining them as follows: *workstudy* : *semester* $\Rightarrow\!\!>$ {*student*, *employee*}. Signatures are further discussed in Section 6.

Inheritance. Methods defined in the scope of a class C are *inherited* by each of the subclasses of C and by all of its instances. This means that even though a function may not be explicitly defined on a class-object or an individual object o, it may still be implicitly defined, provided that this function is defined for a superclass of o. The same holds for attributes.[3] This kind of inheritance is called *behavioral*.

Another aspect of inheritance is called *structural* inheritance and is distinct from behavioral inheritance: If C is a subclass of C' then all objects in C share structural commonality pertaining the objects in C and those in C'. We then say that C *inherits* the common structure of objects in C'. See Section 6 for the details.

3 Path Expressions

3.1 Definitions

Figure 1 shows an object-oriented schema.[4] Path expressions describe paths along the composition hierarchy, and can be viewed as compositions of methods. For example,

```
mary123.Residence.City                    (1)
```

describes a path that starts in the object of class *Person* denoted by *mary*123, continues to the residence of *mary*123, and ends in the city of that residence. In (1), "*mary*123" is called a *selector*, and "*Residence*" and "*City*" are called *attribute expressions*.

Path expressions can be more general than the one above. Formally, a *path expression* is of the form

$$\texttt{sel}_0\texttt{.AttEx}_1\{[\texttt{sel}_1]\}.\ \cdots\ \texttt{.AttEx}_m\{[\texttt{sel}_m]\} \qquad (2)$$

where $m \geq 0$, and braces denote optional terms (i.e., only the first selector sel_0 is mandatory). A selector is either *ground* (abbr. *g-selector*) or *variable* (abbr. *v-selector*). A g-selector is just an object id, and a v-selector is an *individual variable* that ranges over id's of individual objects. The attribute expressions $AttEx_1, \ldots, AttEx_m$ in (2) are either attribute names or *attribute variables* that range over attribute names. (We usually omit the classifiers, "individual" or "attribute", of variables when they are clear from the context.) Note that "higher-order" variables do not make the underlying logic second-order (see [8]). Any selector is also a path with $m = 0$.

The formal definition of the meaning of a path expression requires several concepts which will be defined next. A *database path* (or just *path* when confusion does not arise) is any finite sequence of database objects $o_0, o_1, \ldots, o_n$ ($n \geq 0$); the object o_0 is the *head* of the path and o_n is called its *tail*. A *ground instance* of a path expression is obtained by substituting an object id for each v-selector, and an attribute name for each attribute variable. Formally, a path expression E describes a set consisting of all database paths p, such that p *satisfies* some ground instance of E. A path $o_0, o_1, \ldots, o_m$, where the o_i's are objects, *satisfies* the ground instance $sel_0.attr_1\{[sel_1]\}.\ \cdots\ .attr_m\{[sel_m]\}$ if all of the following hold.

- $o_0 = sel_0$.
- For every $j = 1, \ldots, m$, if the selector sel_j is specified in the above path expression (recall that these selectors are optional, by definition) then $o_j = sel_j$.
- For all $i = 1, \ldots, m$, the attribute $attr_i$ must be defined on o_{i-1}. Furthermore, if $attr_i$ is scalar, then o_i must equal the value of $attr_i$ on object o_{i-1}; if $attr_i$ is set-valued then o_i must *belong to* the value of $attr_i$ on o_{i-1}.

The set of database paths satisfying ground instances of the path expression E could be empty. This may happen because of a type error or because the path expression describes an empty set of paths in the current state of the database. For example, if E is the path expression (1) and *mary*123 is not an object of the database, then the set of paths described by E is empty. In contrast, if E is the path expression *mary123.Residence.Salary*, then this is a type error, since the result of *Residence* is an object of class *Address*, but *Salary* is not an attribute of that class.

Since (1) is ground and all its attributes are scalar, it is satisfied on at most one database path. By contrast, *uniSQL.President.FamMembers.Name* is satisfied on the paths that begin with the *Company*-object *uniSQL*, pass through *uniSQL*'s president, a family member of that president, and end in the object representing the name of this family member. If *uniSQL*'s president had several family members, there would be several such paths. A variant of (1) leads to the following query:

```
SELECT Y  FROM Person X
WHERE X.Residence[Y].City['newyork']
```

Now we should consider all ground instances of the path expression in the *WHERE* clause. For each ground instance *x.Residence*[*y*].*City*['*newyork*'], we should first check *consistency* with the *FROM* clause; in this case, consistency means that x should be an oid of a person.[5] If this ground instance is consistent, then y is in the answer provided that this instance is satisfied by at least one database path.[6] Observe that a path expression is used as a Boolean predicate, and a ground instance of a path expression is either true or false depending on whether it is satisfied by some database path or not.

Queries may involve path expressions with intermediate v-selectors, where the purpose of these selectors is to limit intermediate objects in the path to instances of some class.

```
SELECT Z  FROM Employee X, Automobile Y
WHERE X.OwnedVehicles[Y].Drivetrain.Engine[Z]
```

This retrieves all engines in the employee-owned automobiles. The purpose of Y is to restrict the search through employee-owned vehicles to just automobiles.

Since attribute names are also logical oid's, the user can query database schema without having to know the internal representation of the system catalogue:

```
SELECT Y  FROM Person X  WHERE X.Y.City['newyork']     (3)
```

Here *X.Y.City*['*newyork*'] is a legal path expression, since Y, being a variable, is an attribute expression. The answer to (3) is the set of all attributes y, such that for *some* object x of class *Person* the ground instance *x.y.City*['*newyork*'] is true (i.e., is satisfied by some database path). Observe that if the selector ['*newyork*'] were omitted in the *WHERE* clause, (3) could return more attributes y as an answer, since for some databases the ground

[3] It is common to distinguish so-called "default" attributes from the rest. It is only the former who are inherited from superclasses. In this paper we are interested in default attributes only.

[4] Figure 1 appears at the end of the paper. Attributes marked with an asterisk are set-valued; other attributes are scalar.

[5] A priori, consistency does not impose any restriction on y, since Y is not mentioned in the *FROM* clause. However, no database path would satisfy this ground instance unless y is an oid of class *Address*.

[6] In this case, there is at most one database path satisfying the ground instance, since all attributes are scalar.

instance $x.y.City$ could be true even if $x.y.City$['newyork'] were false. For instance, if all people in the database lived in Austin or San Francisco and none in New York, (3) would return no answer; if the selector ['newyork'] were deleted, however, the attribute *Residence* would have been returned.

We could extend our syntax by permitting *path variables* and then replace the path expression in (3) by $X.*Y.City$['newyork'], where $*Y$ can be bound to any sequence of attributes. Then, unlike in (3), the user would not even have to know that there must be precisely one attribute in the path from *Person* to *City*. We will not explore this any further here, due to space limitation.

In the previous examples, we used individual variables to range over the objects representing regular data, such as persons, cities, etc., as well as *meta-data*, such as attributes. As mentioned earlier, we distinguish method-object from individual objects. A variable ranging over method-objects is called a *method variable*, and is prefixed with a double-quote (e.g., "Y). Strictly speaking, the path expression in (3) is syntactically incorrect; the correct version would be $X.$"$Y.City$['newyork'].

Attribute variables in path expressions let us ask questions about attributes and methods that are *defined* for certain objects. Often it is also desirable to ask questions about attributes that are *applicable* to an object. As noted in Section 2, attributes need not always be defined for all objects to which they are applicable, since their value may be a null. To ask queries about applicable attributes one needs *type variables*—an issue discussed in [9].

The next example uses *class variables*, i.e., variables that range over id's of classes. To distinguish such variables we will prefix their name with the "♯"-sign.

```
SELECT ♯X  WHERE TurboEngine  subclassOf  ♯X                (4)
```

The *subclassOf* relation is interpreted as a *strict* relation, i.e., *Cl subclassOf Cl* is always false. This query is evaluated, as before, by considering all assignments of oid's to variables. In this case, we must find all oid's of classes that—when substituted for ♯X—make the predicate in the *WHERE* clause true. Thus, the answer to (4) consists of the class names *FourStrokeEngine*, *PistonEngine*, and *Object* (*Object* is the class containing all individual objects as its instances).

Using the following query template, we can formulate more sophisticated queries that retrieve all classes ♯X of the individuals Y that satisfy certain properties:

```
SELECT ♯X  FROM ♯X Y  WHERE  condition on Y, ♯X
```

To summarize the above discussion, not only did we classify the objects into three different categories—classes, methods, and individual objects, but also the variables can be of the following variety: class-variables, method-variables, and individual-variables.

3.2 Comparing Path Expressions

Path expressions can be compared using the comparators =, !=, >, etc. Since path expressions represent sets, these comparators may have to be modified with the quantifiers *some* or *all*.

We define the *value* of a ground path expression π to be the set of the tail objects of the database paths satisfying π. A comparison involving a pair of ground path expressions is evaluated by comparing the values of these path expressions according to the specified comparator. For example, the comparison *_john13.FamMembers.Age some> 20* is true when some family members of *_john*13 are older than 20. The quantifier *some* is used to say that the expression is to be considered true if just one family member of *john* has the right age. To the right of ">" we have a path expression 20, whose value is the singleton set {20}. Hence, no quantifier is needed. The following finds all employees with a family member who is over 20 years old:

```
SELECT X FROM Employee X WHERE X.FamMembers.Age some> 20
```

An *Employee*-object o is in the answer set if $o.FamMembers.Age$ evaluates to a set with at least one member > 20.

Path-comparisons can be combined using Boolean connectives *and, or, not*. Since path expressions are evaluated to sets they, too, can be compared using standard set-comparators *contains*, *containsEq*, *subset*, *subsetEq*, etc. We can also apply union, intersection, and set-difference to path expressions. The next query finds all automobile companies managed by young presidents who own both blue and red vehicles:

```
SELECT X  FROM Automobile Y
WHERE Y.Manufacturer[X]
  and X.President.OwnedVehicles.Color
                containsEq  {'blue', 'red'}
  and X.President.Age < 30
```

Note that it is not necessary to define the range of X since it can be inferred from the path expression that X is of type *Company*. This query is evaluated similarly to the previous cases: For every *Company*-object x and an *Automobile*-object y that are substituted for X and Y, respectively, check if the value of the ground path expression $y.Manufacturer[x]$ is non-empty; if it is, evaluate the comparisons $x.President.OwnedVehicles.Color$ *containsEq* {'blue', 'red'} and $x.President.Age < 30$. If both are true, place the *Company*-object x in the answer. Other interesting examples of elementary comparisons include:

```
X.Residence.City =all X.FamMembers.Residence.City
    Y.FamMembers.Age all<all X.FamMembers.Age
```

The first one selects *Person*-objects all whose family members reside in the same city; the second finds pairs of persons such that all family members of one person are strictly older than every family member of the other person.

Finally, we remark that path expressions can be arguments to *aggregate* functions, such as *sum*, *count*, or *average*. Thus, the query to find all employees that make less than $35,000 and have family of more than 4 members all of which live in the same house is written as follows:

```
SELECT X  FROM Employee X
WHERE count(X.FamMembers) > 4  and X.Salary < 35000
  and X.Residence =all X.FamMembers.Residence
```

3.3 Constructing and Manipulating Relations

So far, we have dealt with queries that selected objects from one class of the database according to a specified condition. Conditions for selection could be rather complicated, but the *SELECT* clause always had a single variable.

There is no reason to restrict the *SELECT* clause just to a single variable. Moreover, instead of writing a variable in the *SELECT* clause, it is possible to write any number of *scalar* path expressions (that is, expressions that produce single values once variables are bound to specific object id's). For example,

```
SELECT X.Name, W.Salary  FROM Company X                    (5)
WHERE X.Divisions.Employees[W]
```

would return a binary relation. Its first column is a name of a company, and the second is the salary of some employee of a division of that company. Query (5) is evaluated as follows: for each assignment of object id's x and w to variables X and W, respectively, a tuple $<x.Name, w.Salary>$ is added to the result, provided that the condition in the *WHERE* clause, which forces w to be an employee of some division of company x, is satisfied.

Since, in general, the *SELECT* clause can contain any list of scalar path expressions and the *WHERE* clause can be any Boolean combination of conditions, we can specify any join, including the implicit and explicit joins discussed in [12]. An example of an explicit join is the following query:

```
SELECT X, Y  FROM Company X                                (6)
WHERE X.Name =some X.Divisions.Employees[Y].Name
```

This query produces tuples consisting of a company-object and an employee-object such that the employee has the same name as the

company he works in. In [12], a join of this type is called explicit, since it involves a comparison of two attributes that share a common domain, rather than being based solely on the composition hierarchy. Relations computed by queries can be manipulated via usual SQL operators *UNION, MINUS,* etc.

3.4 Semantics

The formal semantics of queries considered thus far can be easily defined. Given a query Q, all substitutions of oid's for variables should be considered, provided that they respect the sorts (individual, class, or method) of the variables. For each substitution that is consistent with the *FROM* clause, all ground path expressions are evaluated. Next, the *WHERE* clause is evaluated as follows. A stand-alone ground path expression is true if its value is non-empty; a comparison is true if the values of the ground path expressions involved in it stand in the specified relationship (such as "=", "*some* >", "= *all*"). The Boolean operators (*and, or,* and *not*) are evaluated in the usual way. If the *WHERE* clause evaluates to *true,* then the scalar ground path expressions in the *SELECT* clause are evaluated. The result of this evaluation is a tuple of oid's that is added to the answer of the query.
The following theorem shows that the semantics of our language is rooted in F-logic [8].

Theorem 3.1 *There exists an effective procedure $\mathcal{P}$ that for any given* XSQL *query ϕ (of the form considered thus far) returns an equivalent first-order query in F-logic $\mathcal{P}(\phi)$.*

4 Creating New Objects

Queries considered so far return relations, i.e., sets of tuples of object id's. The tuples themselves do not have object id's and duplicates are not allowed. In this section, we define queries that return complex objects (rather than just tuples) and show how to assign oid's to the newly created objects.

4.1 Assigning OID's to Query Result

Instead of merely viewing the result of a query as an ordinary relation, we can also view tuples produced by queries as new objects. This necessitates assigning object id's to the new tuples produced by queries. In addition, since attribute names are crucial to the composition of a complex object, we need to extend our syntax to accommodate explicit assignment of values to attributes. Consider the following query:

```
SELECT EmpSalary = W.Salary  FROM Company X
OID FUNCTION OF X,W
WHERE X.Divisions.Employees[W]
```

This query has two new features. First, the *SELECT* clause gives explicit names to attributes of the output relation (in this case, there is a single attribute, called *EmpSalary*). Second, the *OID FUNCTION OF* clause determines an object id for each tuple in the result. Note that a tuple of the result is generated from a pair of object id's, say x and w, that are assigned to variables X and W, respectively. We follow the idea of [10] that the object id of a tuple generated from x and w should be a function of x and w. In other words, associated with the query there is some partial function f, called *id-function*, such that the object id of the tuple generated from x and w is $f(x,w)$. The user does not have to know what the function f is. In fact, it can be any partial function provided that for each pair of oid's x and w, the value of $f(x,w)$ is unique, if defined, and does not occur elsewhere in the database. Also note that the function is not required to have a short mathematical form (such as $2^x 3^w$). In fact, the function can be stored as a table showing explicitly the oid created for each pair of object id's x and w.
Although the answer to the above query has no *Employee*-objects (but rather salaries), the id-function provides a correspondence between employees and the entities of the answer set. As the id-function depends on both x and w, any *Employee*-object o will have more than one corresponding object in the result, if o represents an employee that works for more than one company. If each employee works for only one company, we may write:

```
SELECT EmpSalary = W.Salary  FROM Company X
OID FUNCTION OF W   WHERE X.Divisions.Employees[W]
```

Here, the id-function depends only on w, and so, for each *Employee*-object there will be a unique tuple in the result. One might wonder what would happen if we use the following query:

```
SELECT CompName = X.Name, EmpSalary = W.Salary
FROM Company X   OID FUNCTION OF X
WHERE X.Divisions.Employees[W]
```

In the answer to this query, two tuples corresponding to distinct salaries in the same company will be assigned the same id (since the id-function depends only on the company). Since an object is defined solely by its object id, this is a contradiction. We view this situation as an ill-defined query (a run-time error).
So far, we have seen queries that create objects with only scalar attributes. We can also define objects that have set attributes. Suppose we want to create objects that have a scalar attribute for a company name and a set attribute that returns the set of all employees of that company. This is accomplished thus:

```
SELECT CompName = Y.Name,
       Employees = Y.Divisions.Employees                    (7)
FROM Company Y   OID FUNCTION OF Y
```

The precise meaning of (7) is as follows. For each oid y assigned to Y, a new object is created in the result. The value of the attribute *CompName* in that object is the company name of the object assigned to Y. The value of the attribute *Employees* of the new object is the value of the ground path expression *y.Divisions.Employees*, which is a set of employees.
For another example of the use of set attributes in *SELECT*, suppose that companies maintain rosters of beneficiaries, where a beneficiary of a company is either a retiree or a dependent of an employee. This can be accomplished as follows:[7]

```
SELECT CompName = Y.Name,  Beneficiaries = {W}
FROM Company Y   OID FUNCTION OF Y                          (8)
WHERE Y.Retirees[W]
  or  Y.Divisions.Employees.Dependents[W]
```

The braces in *SELECT* indicate that the value of the attribute *Beneficiaries* is the set of all w that, when substituted for W, satisfy the *WHERE* clause, given an assignment y for Y. It is seen clearly from this example that the clause *OID FUNCTION OF* can play the role of the *GROUP BY* clause of SQL. Notice the ease with which the value of *Beneficiaries* is specified. Other similar proposals (e.g., O_2 [2]) would require a nested *SELECT*-clause in order to specify the value of *Beneficiaries*.

4.2 Views

A complete discussion of views in object-oriented databases is beyond the scope of this paper. In this section we illustrate a few salient aspects of querying and updating views that are not available in previous proposals for object-oriented query languages. First, consider the following view definition.

```
CREATE VIEW CompSal AS SUBCLASS OF Object
SIGNATURE CompName ⇒ String,
          DivName ⇒ String, Salary ⇒ Numeral
SELECT CompName = X.Name, DivName = Y.Name,
       Salary = W.Salary
FROM Company X  OID FUNCTION OF X,W
WHERE X.Divisions[Y].Employees[W]
```

It declares a new view, *CompSal*, as a subclass of the class *Object*, which we will take to mean the class of all individual objects. The *SIGNATURE* clause specifies the type of each attribute of

[7] The attributes *Retirees* and *Dependents* of the classes *Company* and *Employee*, respectively, are not shown in Figure 1.

the view. Note that for each employee w of a company x, the view has an object consisting of the name of company x, the name of division y where employee w works, and the salary of w (but no other information about w).[8] Two distinct objects in the view could be equal on all attributes if they correspond to two employees of the same company that have the same salary. Thus, we have a view that provides aggregate information about companies and salaries without containing explicit information about the employees having those salaries (which could be used as a security measure).

The above view can be also used in queries that involve other classes of the database schema. For example, the following query finds all names of automobile companies that have some employees who earn more than $35,000.

```
SELECT X.Manufacturer.Name
FROM Automobile X, Employee W                                    (10)
WHERE CompSal(X.Manufacturer,W).Salary > 35000
```

Here, the expression $CompSal(X.Manufacturer,\ W)$ denotes an object whose id is obtained as a result of an application of the id-function associated with the view *CompSal* to whatever *Automobile* and *Employee* object ids are substituted for X and W, respectively. Whenever the result of the application is defined (recall that *CompSal* is a *partial* function), we reach an employee salary accessible through the attribute *EmpSalary* of the corresponding object in the view, and check that the salary is greater than $35,000. If the comparison is *true*, the name of the company is added to the answer.

The form of the head selector in the path expression in (10) necessitates an extension to our syntax. An *id-term* is either an oid, a variable (class, method, or individual), or has the form $f(t_1, \ldots, t_n)$, where f is a symbol denoting an id-function of n arguments and $t_1, \ldots, t_n$ are id-terms. Now, we allow selectors in path expressions to be id-terms instead of just oid's or variables. Still, the id-term $CompSal(X.Manufacturer, W)$ in (10) does not quite satisfy the given definition of id-terms; this term can be made to satisfy it after replacing it with $CompSal(Y, W)$, where Y is a new variable, and adding the conjunct $X.Manufacturer[Y]$ to the *WHERE* clause.

We conclude this section with an illustration of how the mechanism of assigning object id's can be used to translate view updates to database updates. Consider again the view *CompSal* defined in (9). Assuming that each employee works in just one company, objects in the view stand in the 1-1 correspondence with objects of class *Employee* from which the value of the attribute *Salary* is derived. Thus, an update made through the view on the *Salary* attribute (e.g., increase salaries of employees of UniSQL, Inc. by 10%) can be translated into an update on the database.

In general, a view update can be translated to an update on the database if there is some database class C such that objects of the view are in the 1-1 correspondence with objects of C. We will not define a formal syntax for updating through views, since these details are beyond the scope of this paper. The main point, however, is that due to the explicit correspondence between objects in views and objects in database classes, we have a more powerful mechanism for view update as compared to the relational model and other proposals for object-oriented query languages.

A brief discussion of our treatment of views compared to [1] is in order. Besides the obvious syntactic differences, the main distinction is that our queries create sets of objects while in [1] they create relations. Therefore, we can use queries to define views, just as in the relational model, while [1] goes outside the query language to convert tuples into objects. Apart from the non-uniformity, this approach faces difficulties when the objects to be created are to have set attributes. Moreover, our explicit use of the clause "*OID FUNCTION OF*" circumvents the problems of [1] related to the assignment of oid's to "imaginary" objects in views. Query (10) above also shows that, due to our taking id-functions seriously, views and non-views can be used in one query, like in the relational model. It is unclear how this can be achieved in [1] without changing the philosophy underlying the language.

Our discussion of views is incomplete, however. In agreement with [1], we believe that an object-oriented view is *not* just a new virtual class as the above discussion may seem to suggest. In general, a view would include a separate class hierarchy (which may share classes with the "official" class hierarchy of the database—see [1] for more discussion). However, as classes are also objects, in our language all work can be done using queries only, while [1] has to work around the limitations of the query language at hand. View hierarchies will be treated in [9].

5 Methods

In the presence of methods, path expressions have format similar to the one used earlier, except that attribute expressions are replaced by more general method expressions.

A k-ary *method expression* is of the form $(Mthd@Arg_1, \ldots, Arg_k)$, where $Mthd$ is a method name of a method variable; $Arg_1, \ldots, Arg_k$ are oid's or variables that play the role of arguments.[9] A method expression is *ground* if it contains no variables. For 0-ary method expressions, i.e., for attribute expressions, we will write $Attr$ instead of $(Attr\ @)$, to save space and to make our extended notation consistent with the old one.

A *path expression* now has the form

$$\mathtt{sel_0.MthdEx_1\{[sel_1]\}.\ \cdots\ .MthdEx_m\{[sel_m]\}} \qquad (11)$$

where $m \geq 0$ and $MthdEx_1, \ldots, MthdEx_m$ are method expressions. Again, braces in (11) surround the optional selectors.

The definition of satisfaction of path expressions by database paths is an obvious modification of the definition in Section 3.1. A database path $o_0, o_1, \ldots, o_m$ *satisfies* a ground path expression

$$\mathtt{sel_0.(mthd_1@a_{1,1}, \ldots, a_{1,k_1})\{[sel_1]\}.\cdots}$$
$$\mathtt{\cdots.(mthd_m@a_{m,1}, \ldots, a_{m,k_m})\{[sel_m]\}}$$

if all of the following hold:

- $o_0 = sel_0$.
- For every $j = 1, \ldots, m$, if sel_j is specified in the above path expression, then $o_j = sel_j$.
- For all $i = 1, \ldots, m$, the method $mthd_i$ is defined on the arguments $a_{i,1}, \ldots, a_{i,k_i}$ in the scope of object o_{i-1}. Furthermore, if $mthd_i$ is scalar, then o_i is the result of this method when it is invoked on the above arguments in the scope of o_{i-1}, i.e., $o_i = mthd_i(o_{i-1}, a_{i,1}, \ldots, a_{i,k_i})$; if $mthd_i$ is set-valued, then $o_i \in mthd_i(o_{i-1}, a_{i,1}, \ldots, a_{i,k_i})$.

The *value* of a ground path expression is, as before, the set of all tails of the database paths satisfying the path expression.

With the above extensions, the semantics of queries carries over from Section 3.4 without change. Theorem 3.1 holds true for this more general case as well.

Methods are defined similarly to queries and views. For instance, (12) defines a new method, *MngrSal*, that is applicable to every *Company*-object. When this method is invoked by a *Company*-object c with an argument d of type *Division*, it returns the salary of the manager of division d in company c. Note that we use *ALTER* to indicate that this method definition extends the original definition of class *Company*. In other words, the following method definition *alters* the definition of class *Company*, and the signature of the newly defined method is *added* to the signatures that are already declared in this class.

[8] It is assumed that an employee cannot work in two distinct divisions of the same company.

[9] In general, a method expression or an argument could even be an id-term; see [9] for a full exposition of this topic.

```
ALTER CLASS Company
ADD SIGNATURE MngrSal :  String ⇒ Numeral
SELECT (MngrSal @ Y.Name) = W
FROM Company X   OID  X                                    (12)
WHERE X.Divisions[Y].Manager.Salary[W]
```

Notice how we used the abbreviated clause "*OID X*" to specify the object (i.e., X) in whose scope the method *MngrSal* is defined. Also notice that the path name *Y.name* is used as an argument of a method expression in the *SELECT* clause, even though—strictly speaking—this is not allowed by the definition. It should be viewed as a shorthand for writing *(MngrSal @ Z)* in the *SELECT* clause and adding the path expression $Y.Name[Z]$ to the *WHERE* clause, where Z is a new variable.

The following query illustrates how methods could be used in path expressions. This query refers to the method defined in (12); it retrieves all vehicles that are manufactured by companies that pay highly to all their division managers.

```
SELECT X FROM Vehicle X
WHERE 200000 <all                                          (13)
   (SELECT W   FROM Division Y
    WHERE X.Manufacturer.(MngrSal @ Y.Name)[W] )
```

Here, for each vehicle x, the nested query is evaluated. If its result is a set containing only numerals > \$200,000, then x is added to the result of the outermost query. The nested query is evaluated thus: For every y in the class *Division*, if $x.Manufacturer.(MngrSal@y.Name)$ evaluates to a nonempty set, add elements of this set to the result.[10]

As mentioned, method arguments can also act as selectors. For instance, using $(MngrSal@'Advertizing')$ in (13) instead of $(MngrSal@Y.Name)$ would direct the system to find vehicles whose manufacturers pay high salaries to their advertizing chiefs. We can also define methods that update the database, e.g., increase salaries of all division managers by a specified percentage:

```
ALTER CLASS Company
ADD SIGNATURE RaiseMngrSal: Numeral⇒ Object
SELECT (RaiseMngrSal @ W) = nil
FROM Company X, Numeral W  OID  X
WHERE  W < 20 and
  (UPDATE CLASS Company
   SET X.Divisions[Y].Manager.Salary =
         (1 + W/100)*X.(MngrSal @ Y.Name) )
```

Notice the special-looking object, *nil*; it expresses the fact that the scalar method *RaiseMngrSal* does not return meaningful values. The purpose of this method is to cause a side-effect through the nested update in the *WHERE* clause. Also, note the use of the method *MngrSal*, defined in (12).

The above definition of *RaiseMngrSal* specifies what needs to be done, should the new method be called in the scope of a *Company*-object with a numeric argument specifying percentage of the raise. Namely, for the given company and percentage, evaluate $W < 20$ (to guard against huge salary increases) and then evaluate the nested *UPDATE* clause. If successful, return *nil*. An *UPDATE* clause evaluates to *true* if and only if the update was successful. We also assume that the conjuncts in the *WHERE* clause are evaluated in the left-to-right manner.

6 Signatures and Typing

6.1 Types and Structural Inheritance

A signature $M : A_1, \ldots, A_n \Rightarrow R$ specified for a class A_0 consists of a method name, M, and a *type expression*

$$A_0, A_1, \ldots, A_n \rightsquigarrow R \qquad (14)$$

where "$\rightsquigarrow$" stands for either "$\Rightarrow$" or "$\Rightarrow\!\!\!\!\!\Rightarrow$", depending on whether M is scalar or set-valued. The type expression says that the method is defined in the scope of class A_0, accepts arguments of types $A_1, \ldots, A_n$, and returns a result of type R.

Note that signatures in the definitions such as (12) do not specify classes in which the corresponding methods are invoked, because these classes are clear from the *ALTER* (or *CREATE*) clauses. However, for the formal treatment of types we indicate these classes explicitly by putting them as the first argument in the corresponding type expressions (cf. (14), (15)). As signatures can be easily confused with type expressions, signatures will always be prefixed with method names (e.g., $M : A, B \Rightarrow C$) while type expressions will be not (cf. (14), (15)).

Consider the following type expression.

$$A'_0, A'_1, \ldots, A'_n \rightsquigarrow R' \qquad (15)$$

We say that (15) is a *supertype* of (14) and (14) is a *subtype* of (15) if each A'_i is a (nonstrict) subclass of A_i, the class R' is a (nonstrict) superclass of R, and both (14) and (15) use the same kind of arrow ("$\Rightarrow$" or "$\Rightarrow\!\!\!\!\!\Rightarrow$"). Note that "supertype" means "superset", that is, the set of functions described by (15) is a superset of the function set described by (14).

Recall that a method may have several definitions, and consequently, may have multiple signatures (this is known as *polymorphism*). In addition, definitions of methods (and signatures) are inherited. We distinguish between *behavioral* and *structural* inheritance. Behavioral inheritance means that definitions of methods are inherited and also overriden: if C' is a subclass of C and M is a method defined for C, then the definition of M is inherited by C'. However, if M is redefined in C', then the new definition overrides the one that would have been inherited from C.

Structural inheritance means that types of methods (but not their definitions) are always inherited and never overriden. Specifically, if C' is a subclass of C, then C' inherits all the signatures of M that exist in C; in addition, class C' may also have new signatures for M (as a result of new declarations of M in C'). Thus, the set of signatures of M in C' consists of all signatures in the ancestors of C' and all signatures in the new definitions of M in C'. Structural inheritance, also called *covariance*, reflects reality in most (if not all) cases, and it is a nearly standard assumption in the works on type theory. A discussion of typing without covariance is beyond the scope of this paper.

It should be clear that if the class hierarchy is a DAG and not just a tree, then C' may have several incomparable superclasses. As explained above, multiple inheritance of types is fairly simple: the set of signatures of M in C' contains all signatures inherited from all superclasses of C'. This means that, as a function, M belongs to the *intersection* of the sets defined by each type expression in these signatures. For instance, the method *earns* may be declared with the signature $earns : project \Rightarrow pay$ in the class *employee* and $earns : course \Rightarrow grade$ in the class *student*. This means that *earns* has two type expressions, $employee, project \Rightarrow pay$ and $student, course \Rightarrow grade$. In particular, in the class *workstudy* which is a subclass of both *student* and *employee*, *earns* returns an object of class *pay* when it is passed an argument of type *project*; if the argument is of the type *course* then the result will be an object of type *grade*.

The issue of behavioral multiple inheritance is much more complex. Suppose that C' is a subclass of both C_1 and C_2 (but neither C_1 nore C_2 is a subclass of the other), and M is defined in both C_1 and C_2. It is then unclear which definition of M is inherited in C'. There is a vast body of work devoted to this issue which we will not discuss here. We adapt the approach of [15], and require the user to resolve inheritance conflicts explicitly (i.e., to state as part of schema definition which declaration of M is to be inherited in C'). However, as explained above, regardless of which definition of M is inherited (and even in case M is re-

[10] Since *Manufacturer* and *MngrSal* are both scalar methods, this set is always either empty or a singleton.

defined in C'), structural inheritance implies that C' inherits all signatures that M has in C_1 and in C_2.
Suppose (14) is a type expression in the declaration of a method M, and (15) is a supertype of (14). We mentioned that the set of functions defined by (15) contains the set defined by (14). Thus M must also belong to the former set (it is in the latter by declaration). Thus, we have the following definition: If a signature of M has the type expression (14), we say that M *possesses* type (15) if (15) is a supertype of (14). By this definition, the set of types possessed by any method is closed under the supertype relationship (which reflects the effect of structural inheritance). When a method M possesses (14), we say that M is *applicable* to arguments of types $A_1, \ldots, A_n$ *in the scope* of the class A_0.

6.2 Well-typed Queries and Type Errors

To simplify the discussion of well-typed queries, we consider only queries in which the *WHERE* clause is a conjunction (i.e., *and* is the only Boolean operator), and the *SELECT* clause is a list of variables. Moreover, we assume that each path expression in the *WHERE* clause has only v-selectors, g-selectors, and method names (in particular, id-terms are not allowed and method variables cannot appear as method expressions). We also assume that if a path expression π appears in a comparison, then either π is just an oid, or π ends in a variable selector (this assumption can always be satisfied by modifying the query[11]).
A *type assignment* $\mathcal{A}$ to a given query is an assignment of at most one type expression to each occurrence of a method name in the *WHERE* clause. Distinct occurrences of the same method name may be assigned different type expressions. A type assignment $\mathcal{A}$ could be either *complete,* in case all occurrences of method names are assigned type expressions, or *partial,* in case only some occurrences are assigned type expressions.
So, consider a type assignment $\mathcal{A}$, and let

$$\mathtt{Sel}_0.(\mathtt{mthd}_1@\mathtt{A}_{1,1}, \ldots, \mathtt{A}_{1,k_1})[\mathtt{Sel}_1].\cdots \cdots.(\mathtt{mthd}_m@\mathtt{A}_{m,1}, \ldots, \mathtt{A}_{m,k_m})[\mathtt{Sel}_m] \quad (16)$$

be a path expression in the *WHERE* clause, where Sel_i and $A_{i,j}$ are object ids or variables, and $mthd_i$ are method names. Note that we assume that *all* selectors Sel_i $(i = 0, \ldots, m)$ appear (this assumption can be easily satisfied by adding new distinct v-selectors wherever selectors are originally missing; it is needed to simplify the following definitions).
If $\mathcal{A}$ assigns a type expression τ to $mthd_i$, then τ must match[12] the number of arguments of $mthd_i$, and must be possessed by $mthd_i$. The type assignment $\mathcal{A}$ forces type assignments to selectors and arguments in (16) as follows. If $mthd_i$ is assigned the type expression $T_{i,0}, T_{i,1}, \ldots, T_{i,k_i} \rightsquigarrow R_i$, then

- $A_{i,j}$ $(1 \leq j \leq k_i)$ is *assigned* the type $T_{i,j}$,
- Sel_{i-1} is *assigned* the type $T_{i,0}$, and
- Sel_i is *assigned* the type R_i.

Note that if both $mthd_i$ and $mthd_{i+1}$ $(1 \leq i < m)$ are assigned type expressions, then Sel_i is assigned two types, R_i and $T_{i+1,0}$, that are not necessarily the same. Since a variable X may have multiple occurrences in the *WHERE* clause, a type assignment $\mathcal{A}$ may assign multiple types to X. Formally, the *range* of X with respect to $\mathcal{A}$, denoted $\mathcal{A}(X)$, is the set consisting of

- *Object* (i.e., each individual variable is automatically restricted to be of type *Object*),[13]
- all the types that $\mathcal{A}$ assigns to occurrences of X in the *WHERE* clause, and
- all the types that are assigned to occurrences of X in the *FROM* clause.

We say that an oid o is *within the range* $\mathcal{A}(X)$ if o is an instance of every class in the range $\mathcal{A}(X)$; $\mathcal{A}(X)$ is *empty* if no oid could ever be in $\mathcal{A}(X)$. For example, if $\mathcal{A}(X)$ contains both *Person* and *Company,* then it is empty. How this is specified is unimportant here. We assume that schema definition provides sufficient information for determining whether $\mathcal{A}(X)$ is empty.
We say that a type assignment $\mathcal{A}$ is *valid* if for each path expression of the form (16), the following holds:

- $\mathcal{A}$ assigns a type expression τ to $mthd_i$ only if τ is possessed by $mthd_i$ and matches the number of arguments of $mthd_i$.
- If Sel_i is an oid and $\mathcal{A}$ assigns a type T to it, then Sel_i is an instance of T.
- If $A_{i,j}$ is an oid and $\mathcal{A}$ assigns a type T to it, then $A_{i,j}$ is an instance of T.
- If $\pi_1 \theta \pi_2$ is a comparison in the *WHERE* clause, where θ is a comparator, then the following is true. The comparison $o_1 \theta o_2$ is well defined for all o_1 and o_2, such that o_i $(i = 1, 2)$ is either π_i, in case π_i is an oid, or o_i is in $A(W)$, in case π_i is a path expression that ends in the v-selector W. (Recall that according to an earlier assumption, a path expression in a comparison is either an oid or ends in a v-selector.)

We define a query to be *liberally well-typed* if there is (at least) one valid and complete type assignment $\mathcal{A}$, such that for each variable X (of the *WHERE* clause) the range $\mathcal{A}(X)$ is not empty. Type-correctness in a logical language—whether it is Datalog or an SQL derivative—is usually a *metalogical* notion. This means that it does not affect the semantics of queries and any query (well-typed or not) can be evaluated. Therefore, to evaluate a liberally well-typed query we can use the naive process described in Section 3.4 (and extended in Section 5). We might use various optimization strategies, including the type information. For instance, if a preliminary (liberal) type analysis shows that a query is ill-typed then it is guaranteed that this query returns no answers regarding of the database contents.
Quite often queries are evaluated by nested loops; that is, each path expression is evaluated by a sequence of nested loops (corresponding to a traversal of the path from left to right), and different path expressions are evaluated one-by-one (also in a sequence of nested loops). The problem here is that as we evaluate the sequence of nested loops, variables become bound to oid's, and so when we evaluate a specific method occurrence, all its arguments must already be bound to oid's of the appropriate types. This suggests a stricter notion of well-typing defined next.
An *execution plan* for a query is just a partial order on the path expressions in the *WHERE* clause. An execution plan specifies the order of evaluating the path expressions. So, let P be an execution plan and let $\mathcal{A}$ be a type assignment for the given query. Consider a path expression π of the form (16) above. The *restriction* of $\mathcal{A}$ to a method occurrence $mthd_i$ in π is the type assignment $\mathcal{A}'$ defined as follows. $\mathcal{A}'$ is identical to $\mathcal{A}$ for every method occurrence m that either appears in a path expression π', such that π' precedes π in the execution plan P, or m appears in π to the left of $mthd_i$. $\mathcal{A}'$ is undefined (i.e., assigns no type expressions) on all other method occurrences in the *WHERE* clause, including $mthd_i$ itself.

[11] The modification is done as follows. Let $\pi_1 \theta \pi_2$ be a comparison in the *WHERE* clause. If π_i $(i = 1, 2)$ ends in a g-selector o (i.e., o is an oid), then replace π_i with o in the comparison $\pi_1 \theta \pi_2$ and add π_i as an new conjunct to the *WHERE* clause. If π_i does not end in any selector, then add a v-selector W at the end of π_i, where W is a new distinct variable.

[12] Recall that due to polymorphism $mthd_i$ may have different definitions with distinct numbers of arguments.

[13] Recall that *Object* is the class containing all individual objects as its instances.

We define a query to be *strictly well-typed* if there is a valid and complete type assignment $\mathcal{A}$ and an execution plan P, such that the following holds.

1. For each variable X (of the *WHERE* clause), the range $\mathcal{A}(X)$ is not empty.
2. For each path expression π of the form (16) above, and for every $mthd_i$ of π, the following holds. Let $\mathcal{A}'$ be the restriction of $\mathcal{A}$ to $mthd_i$ in π and let[14] $\mathcal{A}(mthd_i) = (T_0, \ldots, T_{k_i} \rightsquigarrow R)$. Then
   (a) If argument $A_{i,j}$ $(j = 1, \ldots, k_i)$ of $mthd_i$ is a variable, then the range $\mathcal{A}'(A_{i,j})$ is a subrange (defined next) of the class T_j (note that T_j is the type that $mthd_i$ expects of $A_{i,j}$ under the type assignment $\mathcal{A}$); and
   (b) If Sel_{i-1} is a variable then the range $\mathcal{A}'(Sel_{i-1})$ is a subrange of T_0 (which, again, is the type $mthd_i$ expects of Sel_{i-1}).

The first condition above is the same as in the definition of well-typing. The second condition simply says that when $mthd_i$ is evaluated, its arguments are bound to oid's of the appropriate types. If a plan and a type assignment satisfy the above conditions we will say that they are *coherent* with each other.

Recall that in the above definition, $\mathcal{A}'(A_{i,j})$ is a range, i.e., a set of classes. We say that the range R is a *subrange* of a class T if every oid belonging to the range R is also an instance of T. Whether R is a subrange of T can be determined from the schema definition [9]. To illustrate the notion of a coherent type assignment, consider the following simple query fragment:

```
FROM Person X   WHERE X.Name
```

There is only one execution plan[15]—the graph containing one node and no arcs. A coherent type assignment would be the one that assigns the expression $Person \Rightarrow string$ to $Name$. An assignment $\mathcal{A}$ such that $\mathcal{A}(Name) = (Employee \Rightarrow string)$ would not be coherent with the plan because if $\mathcal{A}'$ is the restriction of $\mathcal{A}$ to $Name$ then $\mathcal{A}'(X) = \{Person\}$. The latter is not a subrange of $Employee$, the type that $Name$ expects of X according to $\mathcal{A}$.

The difference between a liberally and a strictly well-typed query was illustrated in the introduction. The query on Nobel Prizes is liberally but not strictly well-typed (unless the method *WonNobelPrize* is defined for every class in the database). It is important to understand that the concept of the execution plan is not part of the query semantics and the user does not have to think in terms of these plans when writing queries. So, execution plans do not affect the declarative nature of XSQL.

A strictly well-typed query can be evaluated just as a liberally typed query, using the semantics in Sections 3.4 and 5. However, we can utilize much of the typing information to optimize execution. First we need to find an assignment $\mathcal{A}$ and an execution plan P such that $\mathcal{A}$ is valid, complete, and coherent with P. For any such plan P, it is easy to write a nested loop program that evaluates the answer to the query. In [9] we show how to find coherent pairs $(\mathcal{A}, P)$. The following theorem (whose proof appears in [9]) shows that it suffices to evaluate the query with respect to just *one* such coherent pair. Moreover, for each v-selector X it suffices to limit instantiations to oid's taken from $\mathcal{A}(X)$. This potentially very powerful optimization is not possible with untyped queries and is not always possible even with queries that are liberally (but not strictly) well-typed.

[14] $\mathcal{A}(mthd_i)$ denotes the type expression that $\mathcal{A}$ assigns to the method name $mthd_i$. Recall that if X is a variable (and, hence, not a method name), then $\mathcal{A}(X)$ is the range of X with respect to $\mathcal{A}$.

[15] It is convenient to represent a plan as a DAG in which nodes correspond to path expressions and arcs describe the partial order.

Theorem 6.1 *Let* Q *be a strictly well-typed query and* $\mathcal{A}$ *and* $\mathcal{A}'$ *be a pair of valid and complete type assignments for* Q *that are coherent with the execution plans* P *and* P'*, respectively. Then:*

1. *Evaluating* Q *with respect to any one of these plans yields the same result.*
2. *In the evaluation of* Q *with respect to, say, the plan* P*, it suffices to consider only those instantiations* o *of* X *such that* $o \in \mathcal{A}(X)$*, for every v-selector* X *in* Q.

While the Nobel Prize example suggests that conservative well-typing may sometimes be too strict, liberal well-typing is too permissive, as it does not take into account the fact that path expressions are usually evaluated in nested loops. To strike a better balance between these two notions we define *well-typing with exemptions*. Namely, whenever desired, we exempt arguments of certain method occurrences from the second test in the definition of strict well-typing. For example, exempting the 0-th argument of *WonNobelPrize* makes the path expression *X.WonNobelPrize* type-correct. Note that the liberal and the conservative notions of well-typing are just the two extremes of the notion of well-typing with exemptions: the liberal notion exempts all arguments while the conservative exempts none. To illustrate strict well-typing, consider the following query fragment:

```
FROM Vehicle X
WHERE X.Manufacturer[M]                                    (17)
  and M.President.OwnedVehicles[X]
```

We have two path expressions and three different execution plans. The first plan has no arcs. The second plan has an arc from the first path expression in (17) to the second. The third plan contains an arc going in the opposite direction. Consider the following valid and complete type assignment $\mathcal{A}$:

$$\begin{aligned} &\mathcal{A}(\text{Manufacturer}) = (\text{Vehicle} \Rightarrow \text{Company}), \\ &\mathcal{A}(\text{President}) = (\text{Company} \Rightarrow \text{Person}), \\ &\mathcal{A}(\text{OwnedVehicles}) = (\text{Person} \Rightarrow\!\!\!> \text{Vehicle}). \end{aligned} \tag{18}$$

It does not satisfy the second condition for strict well-typing with respect to the first and the third plans because M does not occur in *FROM*. Indeed, consider the restriction of $\mathcal{A}$ to *President*, call it $\mathcal{A}'$. Then the range $\mathcal{A}'(M)$ that $\mathcal{A}'$ assigns to the second occurrence of M in (17) is $\{Object\}$. On the other hand, the type that *President* expects of M under $\mathcal{A}$ is *Company*. However, $\{Object\}$ is not a subrange of *Company*, contrary to the second condition in the definition of coherence. The situation is different if we consider the second execution plan. Here $\mathcal{A}$ is coherent with that plan and so the query is strictly well-typed.

Execution plans are not always as simple as the above example may suggest. Suppose the method *Member* has a type expression $Association, Numeral \Rightarrow\!\!\!> Organization$ and let *President* have one more type expression: $Organization \Rightarrow Person$. Consider the following query fragment:

```
FROM Numeral Year
WHERE X.Manufacturer[M]                                    (19)
  and M.President.OwnedVehicles[X]
  and OO_Forum.(Member @ Year)[M].
```

Now there are many execution plans, some with and others without coherent type assignments. The only plan with a coherent type assignment, call it $\mathcal{A}_1$,

$$\begin{aligned} &\mathcal{A}_1(\text{Manufacturer}) = (\text{Vehicle} \Rightarrow \text{Company}), \\ &\mathcal{A}_1(\text{President}) = (\text{Organization} \Rightarrow \text{Person}), \\ &\mathcal{A}_1(\text{OwnedVehicles}) = (\text{Person} \Rightarrow\!\!\!> \text{Vehicle}). \end{aligned} \tag{20}$$

is the plan containing arcs from the third to the second and from the second to the first path expressions. In other execution plans, either the restriction of $\mathcal{A}_1$ to *Manufacturer* assigns X the range $\{Object\}$ or the restriction of $\mathcal{A}_1$ to *President* does so for M. In either case $\{Object\}$ is not a subrange of the types that the *Manufacturer* and *President* expect of X and M (which are *Vehicle* and *Organization*, respectively).

7 Conclusion

We presented some of the salient features of a new language for querying object-oriented databases. The language is capable of expressing sophisticated queries in a very concise way. This is achieved via *extended* path expressions, which are more expressive than any of the previous manifestations (for example, [16, 19, 3, 4, 5]) of the dot notation for nested structures.

The proposed language has a rigorously defined notion of type correctness (which is absent from all previous proposals for object-oriented query languages). In fact, we argued that there must be several such notions available and the user should have the option to choose the one most suitable for the query at hand.

Issues concerning the expressive power are beyond the scope of this paper. Suffices it to mention that we can show that the proposed language has the expressive power of first-order queries in F-logic [8] (which are analogous to queries in Codd's relational calculus, but are built on an object-oriented logic.)

Acknowledgments

Preliminary ideas concerning the use of path expressions for querying object-oriented databases came from [16, 19] and from the work that Won Kim did together with Jay Banerjee, Fausto Rabitti, and Elisa Bertino. Selectors in path expressions were first proposed in [13] and later modified based on the ideas in [10]. The use of first-order variables for schema browsing originates in [7, 8]. The authors would also like to thank Alex Borgida, Mariano Consens and Georg Lausen.

References

[1] Abiteboul, S., A. Bonner, "Objects and Views," *ACM SIGMOD Conf. on Management of Data*, 1991.

[2] Bancilhon, F., S. Cluet, and C. Delobel, "The O_2 Query Language Syntax and Semantics," Technical Report 45-90, GIP Altair, May 1990.

[3] Beech, D., "A Foundation for the Evolution from Relational to Object Databases," *Proc. EDBT Conf.*, Venice, Italy, 1988, 251–270.

[4] Cluet, S., C. Delobel, C. Lécluse, and P. Richard, "Reloop, an Algebra Based Query Language for an Object-Oriented Database System," *1st Intl. Conf. on Deductive and Object-Oriented Databases,* Kyoto, Dec. 1989, 294–313.

[5] Delobel, C., C. Lécluse, P. Richard, "LOOQ: A Query Language for Object-Oriented Databases, Informal Presentation," *Proc. AFCET Conf. on Knowledge and Object-Oriented Database Systems,* Paris, Dec. 1988.

[6] Gardarin G., P. Valduriez, "ESQL2: An Object-Oriented SQL with F-Logic Semantics," *Proc. of IEEE Intl. Conf. on Data Engineering*, Phoenix, AZ, Feb. 1992.

[7] Kifer, M., G. Lausen, "F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Schema," *SIGMOD Conf. on Management of Data,* 1989, 134–146.

[8] Kifer, M., G. Lausen, J. Wu, "Logical Foundations of Object-Oriented and Frame-Based Languages," TR #90/14, Dept. of Computer Science, SUNY at Stony Brook, Aug 1990. to appear in J. of ACM.

[9] Kifer, M., Y. Sagiv, W. Kim, "A First-Order Query Language for Object-Oriented Databases," UniSQL Tech. Report, 1992. in preparation.

[10] Kifer, M., J. Wu, "A Logic for Object-Oriented Logic Programming," *Proc. of PODS,* 1989, 379–393.

[11] Kim, W., et al., "Features of the ORION Object-Oriented Database System," in *Object-Oriented Concepts, Databases, and Applications,* (W. Kim and F. Lochovsky, eds.) May 1989, Addison-Wesley/ACM Press, 251–282.

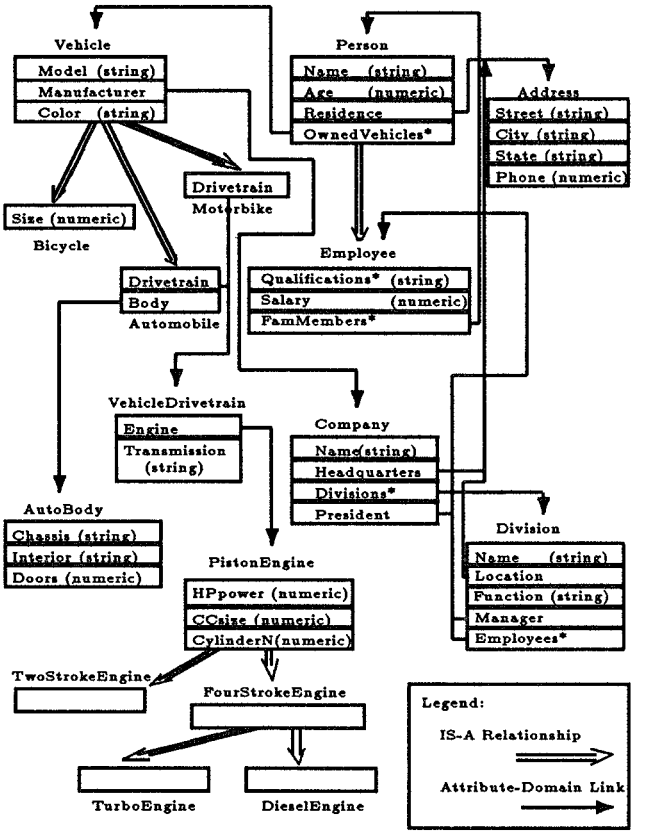

Figure 1: An Object-Oriented Database Schema

[12] Kim, W., "A Model of Queries for Object-Oriented Databases," *Proc. of VLDB,* August 1989, Amsterdam, the Netherlands, 423–432.

[13] Kim, W., Y. Sagiv, "A Query Language for Object-Oriented Databases," MCC TR# ACT-OODS-087-90, Feb. 1990.

[14] Lécluse, C., P. Richard, "The O_2 Database Programming Language," *Proc. of VLDB,* Amsterdam, Aug. 1989.

[15] Meyer, B., "Object-Oriented Software Construction," Prentice Hall, 1988.

[16] Mylopoulos, J., P.A. Bernstein and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications", *ACM TODS*, 5:2, 1980, 185–207.

[17] Roth, M. A., H. F. Korth, and D. S. Batory, "SQL/NF: A Query Language for ¬1NF Relational Databases," *Information Systems,* **12**:1(1987), 99–114.

[18] Schek, H.-J., and M. H. Scholl, "An Algebra for the Relational Model with Relation-Valued Attributes," *Information Systems,* **11**:2(1986), 137–147.

[19] Zaniolo, C., "The Database Language GEM," *SIGMOD Conf. on Management of Data,* 1983, 423–434.