Querying Object-Oriented Databases*

Michael Kifer†
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4, Canada
kifer@cs.sunysb.edu

Won Kim
UniSQL, Inc.
9390 Research Blvd.
Austin, TX 78759, U.S.A.
execu!sequoia!unisql!kim@cs.utexas.edu

Yehoshua Sagiv
Department of Computer Science
The Hebrew University
Jerusalem 91904, Israel
sagiv@cs.huji.ac.il

Abstract

We present a novel language for querying object-oriented databases. The language is built around the idea of extended path expressions that substantially generalize [ZAN83], and on an adaptation of the first-order formalization of object-oriented languages from [KW89, KLW90, KW92]. The language incorporates features not found in earlier proposals; it is easier to use and has greater expressive power. Some of the salient features of our language are:

- Precise model-theoretic semantics.
- A very expressive form of path expressions that not only can do joins, selections and unnesting, but can also be used to explore the database schema.
- Views can be defined and manipulated in a much more uniform way than in other proposals.
- Database schema can be explored in the very same language that is used to retrieve data. Unlike in relational languages, the user needs not know anything about the system tables that store schema information.
- The notions of a type and type-correctness have precise meaning. It accommodates a wide variety of queries that might be deemed well- or ill-typed under different circumstances. In particular, we show that there is more than one way of settling the issue of type correctness.

For expository purposes and due to space limitation, we chose to make a number of simplifying assumptions and left some features out. A more complete account can be found in [KSK92].

*Appeared in ACM SIGMOD Conference on Management of Data, San Diego, CA, June 1992, pages 393–402
†Work supported in part by the NSF grants IRI-8903507 and CCR-9102159. On sabbatical leave from Stony Brook University.

1 Introduction

In recent years, several papers [BANC90, BEEC88, CLUE89, DLR88, KS90, 1] have proposed query languages for object-oriented databases. However, none of these languages captures (or even attempts to deal) with all the aspects of the object-oriented model. In this paper, we present a new query language that incorporates features not found in earlier languages. The proposed language, henceforth referred to as $XSQL$, is easier to use and has more expressive power than previous languages. It should be emphasized that it is not our goal here to introduce 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 [KIM89b, KS90] and [CW89, KW89, KW92, KLW90]. Before discussing the novelties found in our language, 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. A detailed discussion of these issues is found in [KIM89b]; we will describe some of the important aspects through an example. 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 its 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.[1] 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).

The above example shows the need for features not available in relational query languages. In particular, since an object-oriented schema is likely to have much more information than a relational schema, querying the schema (as well as querying the 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 [ZAN83] and had many incarnations since then), 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 and attributes (and even methods) rather than data, 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. The language of [KLK91] has some similar features, but it was designed for the relational model.) Note that in spite of having variables that range over classes, attributes, and methods, the language is still first order, since it is based on F-logic [KLW90].

2. Path expressions also have selectors that could select either some specific data or some specific part of the schema (from which data is to be retrieved).

3. Path expressions may incorporate both attributes and methods in a uniform way that is more general than just composing methods as function applications (as found in functional query languages).

[1] Actually, this example is rather intricate. One may want to know all the engine types that are currently installed in some vehicles, or one may want to know all the engine types that exist, including those that are currently not installed in any vehicle. The language we propose can handle easily each one of these possibilities.

4. Path expressions "flatten" any nested structure in one sweep, and therefore, there is no need to break a path of the schema into several path 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.

One drawback of earlier languages is that they violate encapsulation. As explained in [BANC90], violating encapsulation means that "objects can be considered as the values they encapsulate." This is not the case in our language. We use the approach of F-logic [KLW90] in order to give precise semantics to $XSQL$ without violating encapsulation.[2] In essence, the language manipulates objects (and not the values they encapsulate), and is capable of creating new objects from existing ones. Therefore, $XSQL$ also provides a powerful viewing mechanism, which is discussed in Section 4. When creating new objects, we adapt the approach of [KW89] that invents 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 [AB91]. As in the relational model, views in our language are constructed via queries, which is simpler and more uniform than in other proposals.

Typing is one of the cornerstones of the object-oriented model. Earlier languages, however, hardly discussed the question of when a query is well-typed. We discuss this problem in the framework of $XSQL$, and show that there is more than one way of settling the issue. The following example may help crystalize some of the options. Consider a database that has information on the winners of Nobel Prizes. In particular, there is an attribute (or possibly a method) *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, winners could be persons or organizations of various types.[3] 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
```

and the answer would be 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.[4] Obviously, if we allow queries of this form, the expressive power of the language is enhanced considerably. However, too much expressive power might violate the principle behind typing, and might result in users getting unexpected results (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 [KSK92].

[2] Familiarity with F-logic is not necessary for understanding this paper, except for Theorem 3.1 and the expressiveness result mentioned in the concluding section.

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

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

2 Data Model Review

Since conceptual data model is an integral part of any query language, we first review the object-oriented data model used throughout this paper. This model was put together into a coherent logical system in [KLW90], although some of its elements have appeared earlier, in [KW89, CW89, KIM89a, KL89] and other papers and systems.

Objects and object identity. Objects are abstract or concrete entities in the real world. In our model, the programmer refers to objects via their *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 [KW89, KL89, KLW90] and use explicit id-functions (such as *secretary* above) to get our hands on a sufficient supply of such ids. This mechanism will be primarily used to define user views.

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 oids can be implemented as physical object identities, but unlike physical pointers logical oids may carry certain semantic information. For instance, we consider '20' to be 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 consisting of the characters 'F', 'o', 'r', 'd', ' ', etc., in that order.

Since this paper touches upon language issues only, we will 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* for an object *obj*. If an attribute is *defined,* then it also has a *value* for *obj*; otherwise, it has no value. If an attribute is not applicable to *obj*, then it is also undefined for *obj*, but undefinedness does not imply inapplicability. Intuitively, 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. In contrast, undefinedness of an attribute is analogous to the null value in the relational model. The issue of typing will be formally taken up in Section 6.

Unlike some other approaches [LR89], our model does not divide the world into set-objects and tuple-objects. Essentially, 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. As explained in [KW89, CW89], this approach achieves more uniformity than other proposals, and modeling sets of arbitrary nesting depth becomes quite easy.

Following [KL89, KLW90], we do not completely isolate the space of attribute names from the space of other logical oids. In other words, any logical oid, depending on its syntactic position in a query, may play the role of an attribute or that of an object. Theoretical underpinnings of this approach appear in [KLW90]; its practical benefits—as we shall see—are that the user can now ask questions about the structure of the database in a very natural way, without knowing the system tables that represent the database schema. In practice, it is useful to distinguish attribute names from other objects by placing them in a subdomain of the domain of all objects, dedicated specifically to attribute- and method-objects. This can be handily achieved by making the system catalogue part of the class hierarchy. Details are unimportant here, but can be found in [KSK92].

Classes. Classes have the function of organizing objects into sets of related entities. However, classes are also objects. They can have attributes just like regular objects and can be queried as regular objects. To distinguish objects in the regular sense from classes, we will call the former *individual* objects or just *individuals*.

There is a pair of special binary relationships defined on objects. The first one, *instance-of,* is defined between individuals and classes; it determines which individuals belong to which classes. The second relationship, called *IS-A* or *subclass* relationship, is defined between classes and is acyclic. If a class C is a subclass of another class C', then all instances of C must also belong to C'. However, the converse is not necessarily true; for example, if at some point the only students registered in the database are teaching assistants, this does not make the class *Student* a subclass of the class *TA*.

Representing classes as objects achieves a great deal of uniformity, allows to query the class hierarchy (examples in Section 3.1), and eliminates the need for metaclasses (see [KLW90, KSK92]).

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.

Like attributes, methods can be scalar or set-valued, depending on the kind of result they return. Again like attributes, method names are logical oids and therefore can be returned as query answers, which is useful for schema exploration by the user. Furthermore, we do not really distinguish between methods and attributes and simply view the latter as 0-ary methods, i.e., they do not require arguments to be invoked.

We thus see that the space of all objects is divided into three subdomains: individual-objects, class-objects, and method-objects (the latter includes attribute-objects). We assume that the universe of class-objects is disjoint from the other two universes. Classes will be used to classify not just individual objects, but also objects that describe meta-data, such as methods and attributes. We may or may not require the universes of individual-objects and method-objects to be disjoint. If we do, we gain a degree of syntactic safety by imposing stricter rules for syntactic correctness. If we do not, the user has an added flexibility in choosing names for individual-objects, attributes, and methods. As mentioned earlier, this matter concerns the representation of the system catalogue and can be found in [KSK92].

Being a partial function, a method (just like an attribute) may have no value for some arguments. As in the case of an attribute, we distinguish between a method being undefined (i.e., its value is null) and being inapplicable (i.e., a type error). The formal definitions are postponed till Section 6.

Types. In object-oriented languages, the abstract values of interest are objects; types provide one of the important means of classifying objects. Another means of classification is the concept of a class discussed earlier. While types are generally used to classify objects by *structure*, objects are grouped into classes based on *semantic* criteria. Often—if not about always—objects that are instances of the same class also 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 (recall that attributes are 0-ary

methods). The type of a method in a class C is described as a *signature* of the form

$$\texttt{Mthd}: \texttt{Arg}_1, \ldots, \texttt{Arg}_k \Rightarrow \texttt{Result}$$

or

$$\texttt{Mthd}: \texttt{Arg}_1, \ldots, \texttt{Arg}_k \Rrightarrow \texttt{Result}$$

that is attached to the definition of class 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, $\Rrightarrow$, is used for set methods. As explained above, attributes are identified with 0-ary methods and therefore they are covered by the above definition. For aesthetic reasons, we will write their signatures as $attr \Rightarrow class$ (or $attr \Rrightarrow class$) instead of $attr : \Rightarrow class$ (resp., $attr : \Rrightarrow class$).

The above signature is meant to say that when the method $Mthd$ is passed arguments that are instances of classes Arg_1, ... , 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 0th argument. However, the class of the 0th argument is the one for which the signature is defined, and hence it is redundant to include the 0th 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. Polymorphic methods are further discussed in Section 6. However, we do not discuss parametric polymorphism in this paper; the interested reader can consult [KLW90, KW90] for more details. A method can also have different signatures for the *same* type of arguments. For instance, suppose the following signatures are specified for the class *department*:

$$\texttt{workstudy} : \texttt{semester} \Rrightarrow \texttt{student}, \;\; \texttt{workstudy} : \texttt{semester} \Rrightarrow \texttt{employee}$$

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 students, these individuals are also employees of the university. When more than one signature is specified in this way we can save writing by combining them as follows: $workstudy : semester \Rrightarrow \{student, employee\}$. 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: even though an attribute may have no explicitly defined value for a class or on an individual, this attribute is assumed to inherit the value it has in an appropriate superclass.[5] This kind of inheritance is called *behavioral*. There is a potential problem resulting from *multiple inheritance*, i.e., when an object belongs to a pair of superclasses that are incomparable with respect to the IS-A relationship. Another aspect of behavioral inheritance is *overriding* of method definitions. These issues will be touched upon in Section 6.

There is another aspect of inheritance, called *structural* inheritance, which is distinct from behavioral inheritance. As explained above, all objects in a class share some structural commonality. If C

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

is a subclass of C' then all objects in C share the structural commonality pertaining to the objects in C as well as that of C'. Informally, we can say that C *inherits* the common structure of instances of class C'. We deal with structural inheritance in Section 6.

Relations. It has been argued many times in the literature [KW89, AK89, BEER89, KLW90] that objects do not always model real world in the most natural way, and there are situations when the use of relations on a par with objects leads to more natural representation. Relations are more convenient, for example, when a symmetric binary relationship between predicates is called for, or—as it is common with query languages—when query answer is a set of tuples of objects involved in the query. Another argument for having relations in an object-oriented extension of a language like SQL is that it makes upward compatibility with the standard, relational SQL more natural. Although relations can always be encoded as objects, this is not the most natural way of introducing relations and so we prefer to have relations as first-class language constructs.

3 Path Expressions

3.1 Definitions

Figure 1 shows an object-oriented schema.[6] Thick arrows describe the IS-A hierarchy and thin arrows describe the *composition (aggregation) hierarchy.* (Attribute names that end with an asterisk denote set-valued attributes; other attributes are scalar.) Path expressions describe paths along the composition hierarchy, and can be viewed as compositions of methods. For example, the expression

$$\texttt{mary123.Residence.City} \tag{1}$$

describes a path that starts in the object of class *Person* denoted by $mary123$, continues to the residence of $mary123$, and ends in the city of that residence. In (1), "$mary123$" 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

$$\texttt{selector}_0.\texttt{AttEx}_1\{[\texttt{selector}_1]\}.\ \cdots\ .\texttt{AttEx}_\mathrm{m}\{[\texttt{selector}_\mathrm{m}]\} \tag{2}$$

where $m \geq 0$, and braces denote optional terms (i.e., only the first selector 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 [CKW89, KLW90]). Also note that any selector is also a (trivial) path; this follows from the above definition when $m = 0$.

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

[6] Figure 1 appears at the end of the paper.

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 $mary123$ 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 the path expression (1) is ground (i.e., has no variables) and all its attributes are scalar, it is satisfied by at most one database path. In comparison, the path expression

```
uniSQL.President.FamMembers.Name
```

would normally be satisfied on database 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, then there will be several such database paths.

If the expression (1) is slightly modified, it can be utilized in 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.[7] If the ground instance is consistent, then y is in the answer provided that (at least) one database path satisfies the ground instance.[8] 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.

Quite often queries 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. For example, the following query

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

[7] 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*.

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

retrieves all engines installed in the automobiles owned by employees. Here, the purpose of the variable Y is to restrict the search through employee-owned vehicles to just automobiles.

As explained in Section 2, attribute names are also logical object ids. This allows us to use variables for querying database schema without having to know the internal representation of the system catalogue. For instance, in

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

$X.Y.City$['*newyork*'] is a legal path expression, since Y, being a variable, is an attribute expression. The answer to this query 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, the above query would have (potentially) returned more attributes y as an answer, since (for some databases) the ground 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, the above query 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*. We could then replace the path expression in (3) by $X. * Y.City$['*newyork*'], where $*Y$ can be bound to any sequence of attributes. The result would be that, unlike in (3), the user will not even have to know that there must be precisely one attribute in the path from *Person* to *City*. Details of this extension are easy and we will not pursue this issue any further.

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. Although considering attributes as objects is convenient for browsing database schemas, it is nevertheless clear that an attribute (or a method name) is a special kind of an object, henceforth called a *method-object.* Therefore, a variable ranging over method-objects is called a *method variable*, and is prefixed with a double-quote (e.g., "Y). Strictly speaking, this would make the path expression in (3) 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. In order to ask queries about applicable attributes one needs *type variables*—an issue discussed in [KSK92].

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                                                        (4)
WHERE TurboEngine  subclassOf  ♯X
```

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 $\sharp X$—make the predicate in the *WHERE* clause true. Thus, the answer to this query consists of the following class names: *FourStrokeEngine*, *PistonEngine*, and *Object* (where *Object* is the class containing all individual objects as its instances).

Using the following query template, we can formulate more sophisticated queries that, e.g., retrieve all classes $\sharp X$ of individual objects Y that satisfy certain properties:

```
SELECT ♯X
FROM ♯X Y
WHERE some condition on Y and ♯X
```

To summarize, 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 $=$, $!=$, $>$, and the like. 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 $_john13$ are older than 20. Notice that we used the quantifier *some* 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 $\{20\}$. Therefore, no quantifier is needed.

The query that finds all employees with a family member who is over 20 years old is as follows:

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

Formally, the result of this query is a set of objects from class *Employee*. An object o is in the result if the ground instance *o.FamMembers.Age* is evaluated to a set containing at least one number greater than 20.

Clearly, comparisons can be combined using Boolean connectives (e.g., *and, or, not*). In addition, since path expressions are evaluated to sets they can be compared using such standard set-comparators as *contains*, *containsEq*, *subset*, *subsetEq*, etc. We can also apply union, intersection, and set-difference to path expressions. For example, 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
```

which can be used to select *Person*-objects all whose family members reside in the same city; and

```
Y.FamMembers.Age all<all X.FamMembers.Age
```

which is handy for finding 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 it also makes perfect sense to allow passing path expressions as arguments to *aggregate* functions, such as *sum, count, average*, and use the result in comparisons. 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.Residence =all X.FamMembers.Residence
  and X.Salary < 35000
```

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 results of the queries were always sets of object id's from the database (that is, 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, consider the following query:

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

The result is a relation with two columns. The first column is a name of a company, and the second is the salary of some employee of a division of that company. The above query 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 essentially have the ability to specify any join, including the implicit and explicit joins discussed in the query model of [KIM89b]. 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 [KIM89b], 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.

As usual in SQL, relations computed by queries can be manipulated by relational algebra operators, e.g., *UNION, MINUS,* etc.

3.4 Semantics of Queries with Path Expressions

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 langauge is rooted in F-logic [KLW90].

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 Object Id'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 [KW89] 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 result of the above query does not contain oid's of class *Employee* (it only contains salaries of employees), the id-function provides a correspondence between employees and objects of the result. Since the id-function depends on both x and w, any object o of class *Employee* 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, then we may as well write the following query.

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

In this query, the id-function depends only on w, and therefore, for each object of class *Employee*, 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. Therefore, the two tuples with distinct salaries in the same company are two conflicting descriptions of the same object. 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. As an example, suppose we want to create objects that have a scalar attribute for a company name and a set attribute whose value is the set of all employees of that company. This can be accomplished as follows:

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

The precise meaning of this query is as follows. For each object id y assigned to Y, a new object is created in the result. The value of the attribute *CompName* of the new 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 the target list of a query, suppose that companies need to maintain rosters of beneficiaries, where a beneficiary of a company is either a retiree or a dependent of an employee. This can be accomplished via the following query:[9]

```
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 the *SELECT* clause 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 can be 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 [BANC90]) 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 CompSalaries 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, *CompSalaries*, 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 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 in which employee w works, and the salary of w (but no other information about employee w).[10] 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 could provide aggregate information about companies and salaries without containing explicit information about the employees having those salaries (and, obviously, it 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.

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

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

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

Here, the expression *CompSalaries(X.Manufacturer, W)* denotes an object whose id is obtained as a result of an application of the id-function associated with the view *CompSalaries* to whatever *Automobile* and *Employee* object ids are substituted for X and W, respectively. Whenever the result of the application is defined (recall that *CompSalaries* 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,* then 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 of the syntax of such selectors. An *id-term* [KW89] is either an oid, a variable (class, method, or individual), or an expression of 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. Note that the id-term $CompSalaries(X.Manufacturer, W)$ in (10) does not quite satisfy the given definition of id-terms, but can be made to satisfy it after replacing it with $CompSalaries(Y, W)$, 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 to objects in the views can be used to translate view updates to database updates. Consider again the view *CompSalaries* defined in (9). If we assume that each employee works in just one company, then objects in the view stand in the one-to-one 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 (for example, increase salaries of employees of UniSQL, Inc. by 10%) can be translated into an update on the database.

In general, an update on a view can be translated to an update on the database if there is some class C of the database, such that objects of the view are in the one-to-one correspondence with objects of class C. We will not define a formal syntax for updating through a view, 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 [AB91] is in order. Besides the obvious syntactic differences, the main distinction is that our queries create sets of objects while in [AB91] they create relations. Therefore, we can use queries to define views, just as in the relational model, while [AB91] 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 [AB91] 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 [AB91] without changing the philosophy underlying the language.

Our discussion of views is incomplete, however. In agreement with [AB91], we believe that an object-oriented view is *not* just a new virtual class as the discussion in this section 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 [AB91] for more discussion). However, since classes

are also objects, in our language all work can be done using queries only, while [AB91] has to work around the limitations of the query language at hand. View hierarchies will be treated in [KSK92].

5 Methods

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

A k-ary *method expression* is a statement 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.[11] 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

$$\mathtt{selector_0.MthdEx_1\{[selector_1]\}.\ \cdots\ .MthdEx_m\{[selector_m]\}} \tag{11}$$

where $m \geq 0$ and $MthdEx_1, \ldots, MthdEx_m$ are method expressions. Again, braces in (11) are used to single out 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

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

if all of the following hold:

- $o_0 = sel_0$.
- For every $j = 1, \ldots, m$, if the selector 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 objects occurring as the tails of the database paths satisfying the path expression.

With the above extension of the syntax and the meaning of path expressions, 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 can be defined similarly to queries and views. For instance, the following query defines a new method, *MngrSalary,* 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. Not that we use the *ALTER* clause, since we consider the method definition to be an extension of 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.

[11] In general, a method expression or an argument could even be an id-term; see [KSK92] for a full exposition of this topic.

```
ALTER CLASS Company
ADD SIGNATURE MngrSalary :  String ⇒ Numeral
SELECT (MngrSalary @ Y.Name) = W
FROM Company X                                                          (12)
OID  X
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 *MngrSalary* 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 *(MngrSalary @ 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  ( SELECT W                                           (13)
                     FROM Division Y
                     WHERE X.Manufacturer.(MngrSalary @ Y.Name)[W] )
```

This query has the following meaning. For each vehicle x, the nested query is evaluated. If its result is a set that contains only numerals greater than \$200,000, then x is added to the result of the outermost query. The nested query is evaluated thus: For every instance y of the class *Division*, if the path expression $x.Manufacturer.(MngrSalary@y.Name)$ evaluates to a nonempty set, add the only element of this set to the result.[12]

As mentioned, method arguments in path expressions can also be used as selectors. For instance, using ($MngrSalary$@ *'Advertizing'*) in (13) instead of ($MngrSalary$@$Y.Name$) will direct the system to retrieve those vehicles whose manufacturers pay high salaries to their advertizing chiefs.

We can also define methods that update the database, e.g., increase the salaries of all division managers by a specified percentage:

```
ALTER CLASS Company
ADD SIGNATURE RaiseMngrSalary :  Numeral ⇒ Object
SELECT (RaiseMngrSalary @ 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.(MngrSalary @ Y.Name) )
```

Notice the special-looking object, *nil*; it expresses the fact that the scalar method $RaiseMngrSalary$ 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 *MngrSalary*, defined in (12).

The above definition of *RaiseMngrSalary* 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

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

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*

$$\mathtt{A_0, A_1, \ldots, A_n \leadsto R} \tag{14}$$

where "$\leadsto$" 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 $A_1, \ldots, A_n$ (in that order), and returns a result of type R.

Note that signatures in method definitions, such as (12) above, 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 must indicate these classes explicitly, which we do by putting them as the first argument in the corresponding type expressions (cf. (14), (15)). Since 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.

$$\mathtt{A'_0, A'_1, \ldots, A'_n \leadsto R'} \tag{15}$$

We say that (15) is a *supertype* of (14) and (14) is a *subtype* of (15) if each A'_i is a (possibly nonstrict) subclass of A_i, the class R' is a (possibly 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 set of functions described by (14).

Recall that a method M may have several definitions, and consequently, may have multiple signatures (this is known as *polymorphism*). In addition, definitions of methods (as well as signatures) are inherited. We distinguish between *behavioral inheritance* and *structural inheritance*. Behavioral inheritance means that definitions of methods are inherited and also overwritten. Specifically, 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 overwritten. 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 multiple inheritance with respect to behavioral inheritance is much more complex. Suppose that C' is a subclass of both C_1 and C_2 (but neither C_1 nor C_2 is a subclass of the other), and M is defined in both C_1 and C_2. In this case, it is not clear 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 ([ER83, TOU86, THT87, HTT87, BRE87, KK89, KLW90] is just a tip of the iceberg). We adapt the approach of [MEY88], and require the user to resolve inheritance conflicts explicitly (i.e., the user should state which definition of M is inherited in C' as part of the schema definition). However, as explained above, regardless of which definition of M is inherited (and even in case M is redefined in C'), structural inheritance implies that C' inherits all signatures that M has in C_1 and in C_2.

Suppose that (14) is a type expression occurring in the declaration of a method M, and (15) is a supertype of (14). As mentioned earlier, the set of functions defined by (15) contains the set defined by (14), and thus M must also belong to the former set (since by its declaration, it belongs to the latter). Therefore, we have the following definition: If a signature of M has the type expression (14), then 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 (and this closure reflects the effect of structural inheritance). When a method M possesses type (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

In order 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 in the role of 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[13]).

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

$$\mathtt{Sel_0.(mthd_1@A_{1,1},\ \ldots\ ,\ A_{1,k_1})[Sel_1].\cdots.(mthd_m@A_{m,1},\ \ldots\ ,\ A_{m,k_m})[Sel_m]} \qquad (16)$$

[13] The modification is done as follows. Suppose that $\pi_1 \theta \pi_2$ is 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 we 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 we add a v-selector W at the end of π_i, where W is a new distinct variable.

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[14] 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, we define the *range* of X with respect to $\mathcal{A}$, denoted $\mathcal{A}(X)$, as the set consisting of

- *Object* (i.e., each individual variable is automatically restricted to be of type *Object*),[15]
- 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 $\mathcal{A}(X)$. We say that $\mathcal{A}(X)$ 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) above, 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.)

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

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

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 imposes a stricter notion of well-typing, which will be 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.

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[16] $\mathcal{A}(mthd_i) = (T_0, \ldots, T_{k_i} \rightsquigarrow R)$. 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 [KSK92].

[16] $\mathcal{A}(mthd_i)$ 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 $\mathcal{A}$.

To illustrate the notion of a coherent type assignment, consider the following simple query fragment (more examples later):

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

There is only one execution plan[17]—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 well-typed query and a strictly well-typed query was illustrated in the introduction. The query on Nobel Prizes is liberally well-typed 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 [KSK92] we show how to find coherent pairs $(\mathcal{A}, P)$. The following theorem (whose proof appears in [KSK92]) 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 oids 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.

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

1. *Evaluating* $\mathbf{Q}$ *with respect to any one of these plans yields the same result.*
2. *In the evaluation of* $\mathbf{Q}$ *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* $\mathbf{Q}$.

While the Nobel Prize example shows that strict well-typing may be too strict in some cases, liberal well-typing is too permissive, since 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 can define *well-typing with exemptions*. Namely, whenever desired, we can exempt arguments of certain method occurrences from the second test in the definition of strict well-typing. For the Nobel Prize example, we can exempt the 0-th argument of *WonNobelPrize,* which will make 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 the notion of strict well-typing, consider the following query fragment:

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

```
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 $\mathcal{A}$:

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

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: it is easy to verify that $\mathcal{A}$ 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 $Association, Numeral \Rrightarrow Organization$ 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 of which have while others have no coherent type assignments. The only plan that has a coherent type assignment, call it $\mathcal{A}_1$,

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

is the plan containing the arcs from the third to the second and from the second to the first path expressions. This is because 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 and Related Work

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, [ZAN83, BEEC88, CLUE89, DLR88]) of the dot notation for nested structures. We extended path expressions with the concept of selector, accommodated methods, and "higher-order" variables

that range over method names and classes. The use of higher-order variables endows our language with truly novel capabilities that allow the user to browse database schema in a very intuitive way.

The proposed language has a rigorously defined notion of well-typed queries (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.

Views in our language are constructed via queries, in line with the relational model. This is simpler and more uniform than the construction of views in [AB91], and circumvents certain problems that have to be dealt with there. Furthermore, views and non-views can be referred to in the same query.

The 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 [KLW90] (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 the work of Zaniolo [ZAN83] 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 [KS90] and later modified based on the ideas in [CW89, KW89]. The use of first-order variables for schema browsing originates in [KL89, KLW90]. The authors would also like to express their gratitude to Mariano Consens, Georg Lausen and Rodney Topor for their insightful comments on the ideas presented herein.

References

[AB91] Abiteboul, S., A. Bonner, "Objects and Views," *Proc. ACM SIGMOD Conf. on Management of Data*, 1991.

[AK89] Abiteboul, S., P. C. Kanellakis, "Object Identity as a Query Language Primitive," *Proc. ACM SIGMOD Conf. on Management of Data*, 1989, pp. 143–153.

[BANC90] Bancilhon, F., S. Cluet, and C. Delobel, "The O_2 Query Language Syntax and Semantics," Technical Report 45-90, GIP Altair, May 1990.

[BEEC88] Beech, D., "A Foundation for the Evolution from Relational to Object Databases," *Proc. EDBT Conf.,* Venice, Italy, 1988, pp. 251–270.

[BEER89] Beeri, C., "Formal Models for Object-Oriented Databases," *Proc. First Int. Conf. on Deductive and Object-Oriented Databases,* Kyoto, Japan, Dec. 1989, pp. 370–395.

[BERT89] Bertino, E., and W. Kim, "Indexing Techniques for Queries on Nested Objects," *IEEE Trans. on Knowledge and Data Engineering,* Dec. 1989.

[BRE87] Brewka, G., "The Logic of Inheritance in Frame Systems," *Proc. Intl. Joint Conf. on Artificial Intelligence*, pp. 483–488, 1987.

[CKW89] Chen, W., M. Kifer, D. S. Warren, "HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs," Proc. of North-American Conf. on Logic Programming, October 1989, Cleveland, Ohio.

[CW89] Chen, W., D. S. Warren, "C-logic for Complex Objects," *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems,* 1989, pp. 369–378.

[CLUE89] Cluet, S., C. Delobel, C. Lécluse, and P. Richard, "Reloop, an Algebra Based Query Language for an Object-Oriented Database System," *Proc. First Int. Conf. on Deductive and Object-Oriented Databases,* Kyoto, Japan, Dec. 1989, pp. 294–313.

[DLR88] 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.

[ER83] Etherington, D. W., R. Reiter, "On Inheritance Hierarchies with Exceptions," *Proc. National Conf. on Artificial Intelligence*, pp. 104–108, Washington, D.C., 1983.

[1] 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.

[HTT87] Horty, J.F., R.H. Thomason, D.S. Touretzky, "A Skeptical Theory of Inheritance in Nonmonotonic Semantic Nets," *National Conference on Artificial Intelligence*, 1987, pp. 358–363.

[KL89] Kifer, M., G. Lausen, "F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Schema," *Proc. SIGMOD Conf. on Management of Data,* 1989, pp. 134–146.

[KLW90] Kifer, M., G. Lausen, J. Wu, "Logical Foundations of Object-Oriented and Frame-Based Languages," Technical Report #90/14, Department of Computer Science, SUNY at Stony Brook, August 1990. to appear in J. of ACM.

[KSK92] Kifer, M., Y. Sagiv, W. Kim, "A First-Order Query Language for Object-Oriented Databases," UniSQL Tech. Report, 1992. in preparation.

[KW89] Kifer, M., J. Wu, "A Logic for Object-Oriented Logic Programming (Maier's O-Logic Revisited)," *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems,* 1989, pp. 379–393.

[KW90] Kifer, M., J. Wu, "A First-Order Theory of Types and Polymorphism in Logic Programming," Technical Report #90/23, Department of Computer Science, SUNY at Stony Brook, July 1990. Also in Intl. Symp. on Logic in Computer Science (LICS-91), Amsterdam, July 1991.

[KW92] Kifer, M., J. Wu, "A Logic for Programming with Complex Objects," Journal of Computer and System Science, 1992. to appear.

[KIM89a] 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, pp. 251–282.

[KIM89b] Kim, W., "A Model of Queries for Object-Oriented Databases," *Proc. Intl. Conf. on Very Large Data Bases,* August 1989, Amsterdam, the Netherlands, pp. 423–432.

[KK89] Krishnaprasad, T., M. Kifer, "An Evidence-Based Framework for a Theory of Inheritance," *Proc. Intl. Joint Conf. on Artificial Intelligence*, 1989.

[KLK91] Krishnamurthy, R., W. Litwin, and W. Kent, "Language Features for Interoperability of Databases with Schematic Discrepancies," *Proc. ACM SIGMOD Conf. on Management of Data*, 1991, pp. 40–49.

[KS90] Kim, W., Y. Sagiv, "A Query Language for Object-Oriented Databases," MCC Report ACT-OODS-087-90, February 1990.

[LR89] Lecluse, C., P. Richard, "The O_2 Database Programming Language," *Proc. Conf. on Very Large Data Bases,* Amsterdam, The Netherlands, Aug. 1989.

[MEY88] Meyer, B., "Object-Oriented Software Construction," Prentice Hall, 1988.

[MBW80] Mylopoulos, J., P.A. Bernstein and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications", *ACM Transactions on Database Systems*, 5:2, 1980, 185–207.

[RKB87] Roth, M. A., H. F. Korth, and D. S. Batory, "SQL/NF: A Query Language for ¬1NF Relational Databases," *Information Systems,* **12**:1(1987), pp. 99–114.

[SS86] Schek, H.-J., and M. H. Scholl, "An Algebra for the Relational Model with Relation-Valued Attributes," *Information Systems,* **11**:2(1986), pp. 137–147.

[TOU86] Touretzky, D.S., "The Mathematics of Inheritance," Morgan-Kaufmann, Los Altos, CA, 1986.

[THT87] Touretzky, D.S., J.F. Horty, R.H. Thomason, "A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems," *Proc. Intl. Joint Conf. on Artificial Intelligence*, pp. 476–482, 1987.

[ZAN83] Zaniolo, C., "The Database Language GEM," *Proc. ACM SIGMOD Int. Conf. on Management of Data,* 1983, pp. 423–434.

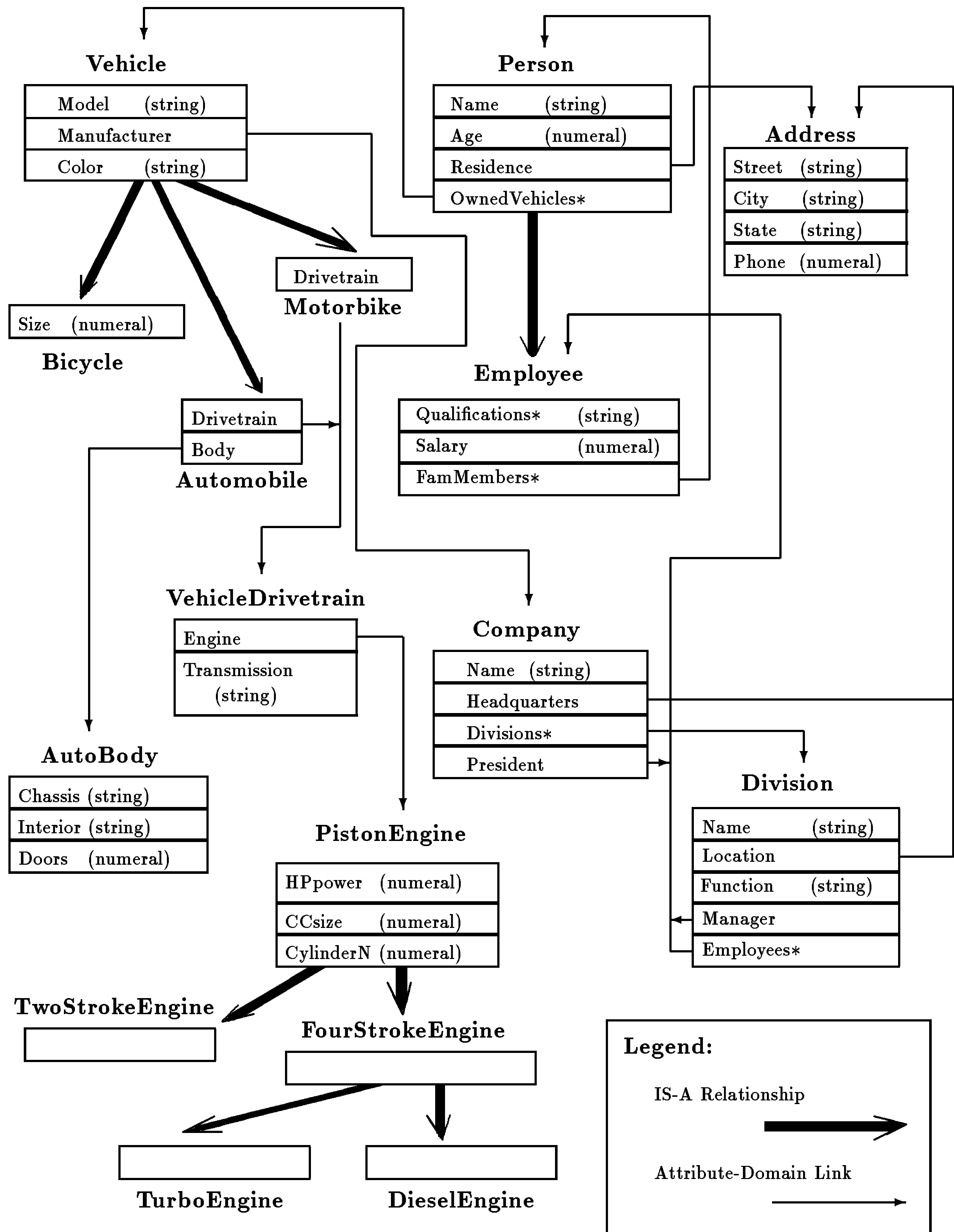

Figure 1: An Object-Oriented Database Schema