A Data Model and A Query Language for Object-Oriented Databases

Manojit Sarkar and Steven P. Reiss

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-57
December 1992

A Data Model and A Query Language for Object-Oriented Databases*

Manojit Sarkar
Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912 USA

Abstract

We present an object-oriented data model, and a powerful declarative query language. The data model eliminates the *object-versus-values* dichotomy by representing all entities as objects. This is achieved at no loss of modeling power, or performance. The data model provides for private ownership of objects by other objects. The operational part of the data model is also simple. We present a rule-based query language called OQL (Object Query Language) based on the data model. OQL creates and manipulates objects without explicitly referring to object identifiers. It is statically typed, and capable of creating new objects of arbitrary types. It can express recursive queries, eliminate duplicates, manipulate mixture of tuple-valued, set-valued and list-valued objects, and express queries involving inheritance trees. We present an object-algebra, an assignment operator, and a REPEAT_UNTIL loop construct which are used to implement OQL. Reduction of OQL to algebra, assignment operation, and loop construct, and reduction of algebra, assignment operation, and loop construct to OQL are also discussed.

1 Motivation

We started investigating object-oriented database systems for constructing abstract information about programs, called *program abstractions*, by retrieving information from some form of stored representation of programs, called *program database*. The idea is to generate abstractions by querying the database, the answers to the queries are the target abstractions. These abstractions are subsequently visualized graphically, and our full system is a *program visualization* system [1].

The data model of our full program visualization system is object-oriented, and we wanted a database with a compatible data model. This paper presents a data model, and a query language to be used with our program visualization system.

The query language presented in this paper is named OQL (Object Query Language). OQL is declarative, and highly expressive. We use the query language for both definition and generation of abstractions. A declarative language can make this tasks easier. Our goal is to develop a language that is nearly as easy or more easy to use than SQL. Higher expressive power is desirable because that allows generation of wider range of abstractions.

Our mapping from abstraction objects to their graphical visualization is type-based. All the generated abstractions therefore must have a type. Also, to provide effective program visualization, our system must have the capability to generate abstractions that are not expressed by existing relationships in the database. OQL is therefore designed for creating objects of any arbitrary type, and the users can define a new type at any time.

2 Contributions

The data model presented in this paper represents all entities as objects. The system uses internally generated *object identifiers* for giving objects their unique identity. An object also has a state, and a behavior. Each object belongs to a class. The class defines the structure of the object's state, and its behavior. Classes are organized in inheritance hierarchy. An object of a class τ can be used where an object of τ's superclass is expected. Inheritance polymorphism is provided through late binding of methods based on actual classes of objects.

Our data model is conceptually simple. It avoids the *objects-versus-values* dichotomy found in other data models such as in [11, 12, 3, 8]. This is achieved at no extra cost. We also show that it is possible to satisfy a variety of data modeling requirements such as *values* of O_2, *composite objects* of ORION [4, 10], and *own ref* objects of EXODUS within the object-oriented paradigm. The conceptual simplicity of the data model does not imply lower performance. It is possible to engineer internal optimizations for good performance.

*Support for this research was provided by NSF grants CCR9111507 and CCR9113226, by ARPA order 8225 and by ONR grant N00014-91-J-4052

[1]Readers interested in our program visualization system are referred to [16]

The query language OQL is the most important contribution of this paper. The language creates and manipulates objects without explicitly referring to object identifiers. It is statically typed. It is capable of creating new objects of arbitrary types. It can express recursive queries, and perform duplicate elimination, and manipulate mixture of tuple-valued, set-valed, and list-valued objects. It is also capable of expressing queries involving inheritance trees.

OQL is implemented using an object-algebra, an assignment operator, and a REPEAT_UNTIL loop construct. The algebra provides new operators for operating on mixture of set-valued, and list-valued objects. The algebra and the language OQL are proved equivalent in expressive power.

The data model, and the query language is under implementation. They have been designed with a practical application in mind. This is discussed in the Motivation section. Section 3 presents the data model. The algebraic operators along with the assignment operator and the loop construct are described in Section 4. The language OQL, its syntax, typeing, and examples are in Section 5. Section 6 describes the semantics of the OQL programs, its translation to the object-algebra, and reduction of the algebraic to OQL. Remaining issues and future research are mentioned in Section 7.

3 Data Model

We start a formal description of the data model with the following pairwise disjoint sets:

- a finite set of *basic types* **D**
- a countably infinite set of *oids* **O**
- a finite set of *attribute names* **A**
- a finite set of *method names* **M**
- a finite set of *class names* **C**

3.1 Values

An object may have a *value*. Only *structured values* can be values of objects. Structured values are built from *atomic values*. Each member of the domain associated with a basic type, and each member **O** is an atomic value. This is formalized below.

3.1.1 Atomic Values

There are four basic types. These are `Integer`, `Float`, `Boolean`, and `String`. Hence **D** = {`Integer`, `Float`, `Boolean`, `String`}. The domains of basic types are called basic domains. Basic domains are pairwise disjoint. Each domain consists of a countably infinite set of atomic values. For example, the domain of type `Integer` consists of the set of all integer values. The atomic values in basic domains are also called *basic values* as opposed to oids.

A *type* is either a class or a basic type. Classes are built from basic types and classes using the `Tuple`, `Set`, and `List` constructors. A class built using the `Tuple` constructor is called a *tuple-class*. Similarly a class built using the `Set`, or the `List` constructor is called a *set-class*, or a *list-class* respectively.

O has three special members. These are `nulltuple`, `nullset`, and `nulllist`. They denote undefined objects of all tuple-classes, set-classes, and list-classes respectively. The value of an undefined object is assumed to be unknown, or an empty-set, or an empty-list depending on the the class of the object. Each member of **O** is also an atomic value.

3.1.2 Structured Values

Structured values are constructed from atomic values using the `Tuple`, `Set`, and `List` constructors. There is one special structured value, [ ] which is used to denote the value any tuple with zero attributes. Note that [ ] and `nulltuple` are not the same values. The former denotes a structured value, while the later is an atomic value and the oid of an object whose value is undefined. Structured values are defined formally as follows:

- the value [ ] is a structured value
- each tuple of atomic values $[a_1 : v_1, a_2 : v_2, \ldots, a_n : v_n]$ is a *tuple-value*, where $n > 0$, $a_i \in \mathbf{A}$ and v_i for $i = 1, 2, \ldots, n$ are attribute names, and atomic values respectively
- each set of atomic values $\{v_1, v_2, \ldots, v_n\}$ is a *set-value*, where $n > 0$, and v_i for $i = 1, 2, \ldots, n$ are atomic values
- each list of atomic values $< v_1, v_2, \ldots, v_n >$ is a *list-value* where $n > 0$, and v_i for $i = 1, 2, \ldots, n$ are atomic values

A set-valued object whose value is an empty-set is considered an undefined object. Similarly, a list-valued object whose value is an empty-list is also considered an undefined object. We denote the set of all the structured values by the countably infinite set **V**.

3.2 Classes

Each object belongs to a *class*. The class describes the structure of the object's value, and the object's behavior. The syntax for class definitions is as follows:

<class decl> → Class *<class name>*
→ Class *<class name>* *<class def>*
<class def> → [Inherits *<superclass name>*]
<class struc>
[Methods {*<class method>*}]
<class struc> → Tuple '(' {*<attr def>*} ')'

→ Set '(' <*basic domain*> ')'
→ Set '(' [Own] <*class name*> ')'
→ Set '(' [Own] <*class struc*> ')'
→ List '(' <*basic domain*> ')'
→ List '(' [Own] <*class name*> ')'
→ List '(' [Own] <*class struc*> ')'
<*attr def*> → <*attr name*> ':' <*basic domain*>
→ [Own] <*attr name*> ':' <*class name*>
→ [Own] <*attr name*> ':' <*class struc*>
<*class method*> → <*method sig*> <*method body*>
<*method sig*> → <*method name*>
'(' {<*param def*>} ')'
':' <*result def*>
<*param def*> → <*basic domain*>
→ <*class name*>
→ <*class struc*>
<*result def*> → <*basic domain*>
→ <*class name*>
→ <*class struc*>
<*method body*> → '{' <*code*> '}'

A class may be defined by specifying a name, a superclass, a structure, and methods. Superclass and methods are optional. It is also possible to define classes *implicitly* without any name, superclass, and methods. Example 3.1 below defines five explicitly declared classes `Point`, `Vertex`, `Edge`, `SegmentedEdge`, and `CurvedEdge`. It also implicitly declares some classes with no name, no superclass, no methods. The class associated with the attribute `bendpoints` of class `SegmemtedEdge` is such a class with structure `List(Point)`. It is also possible to define *recursive* classes. The syntax also allows a class name to be declared and used first, and defined later.

Example 3.1: The following are some class declarations:

```
Class Point
  Tuple (x: Float,
         y: Float)
  Methods {distance (Point): Float {···}};
Class Vertex
  Tuple (Own position: Point,
         label: String,
         radius: Float);
Class Edge
  Tuple (source: Vertex,
         dest: Vertex,
         label: String)
  Methods {length (): Float {···}};
Class SegmentedEdge
  Inherits Edge
  Tuple (Own bendpoints: List(Point))
  Methods {length (): Float {···}};
Class CurvedEdge
  Inherits Edge
  Tuple (Own controlpoints: List (Point))
  Methods {length (): Float {···}}; □
```

3.2.1 Structure

All classes must have a *structure*. As shown above, structures are built using the `Tuple`, `Set`, and `List` constructors from basic types, and already declared class names.

Any object can be potentially referenced by many other objects. Sometimes it is necessary to prohibit such sharing, and allow a class of objects to exclusively refer to other objects. This constraint can be specified using the "`Own`" qualifier in attribute definitions. An object held by an attribute qualified as `Own` cannot be referenced by more than one object.

3.2.2 Methods

A class declaration may have zero or more *methods* as a part of the class definition. Methods define the object's behavior. Each method is a function, it has a *signature* and a *body*. The signature is an expression of the form $c : m(\tau_1, \tau_2, \ldots, \tau_n) \rightarrow \tau_r$ where c is the *receiver class*, m is the *method name*, $\tau_1, \tau_2, \ldots, \tau_n$ are the types of the *parameters* for some $n \geq 0$, and τ_r is the type of the *return value*. A method's body is a piece of code written in some programming language, and it implements the intended function.

3.2.3 Inheritance

A class declaration may specify a *superclass* as a part of its class definition. Inheritance [6, 7] allows the user to derive new classes from existing classes. Only single-inheritance is allowed in our data model *i.e.*, a class can have at most one superclass [2].

The inheritance relationship is a partial order on the classes, *i.e.*, it is *reflexive*, *antisymmetric* and *transitive*. Since superclasses are optional, the class hierarchy is potentially a forest of many disjoint trees, and not a single tree [3].

A class inherits the attributes and the methods of its superclass by default. The structure and method suite of the inheriting class therefore include the inherited attributes and methods respectively. The inheriting class can define additional attributes and methods, or redefine inherited attributes and methods. However the resulting structure and method suite of the inheriting class must be compatible to the structure and method suite of its superclass respectively. For definitions of structural and method suite compatibility see [17].

Subclass Relationship: The actual subclass relationship must be specified by the user by naming the super-

[2] We feel single-inheritance is adequate for our modeling goal. We plan to incorporate multiple-inheritance if our later needs justify the the additional complexity.

[3] We therefore do not associate systemwide operators with any top-level class. In our model, they are predefined operators applicable to any objects of compatible classes.

class during the class definition of the subclass. Since implicitly declared classes do not have names, they cannot be declared to be subclass or superclass of any other class. The system therefore can assume that the implicitly declared classes have no subclass, and optimize the representation and access of the objects of such classes to achieve better performance. Subclass relationship is denoted by the symbol "$<$". If c is a subclass of c', then $c < c'$.

Substitutibility: Subclass relationships allow an object of class c to be used in any context expecting an object of class c', where c' is a superclass of c.

Late Binding: Since an object of a certain class can be assigned to a variable of its superclass, given a method call on a variable, sometimes the method to be executed can only be determined at run time based on the object's actual class. This is known as *late binding*.

Overloading: Method names can also be *overloaded* by defining methods with the same name in more than one class not related by subclass-superclass relationships. Such method name overloading can be resolved at compile time based on method signatures.

Class Equivalence: Two classes are *equivalent* if they have the same name. The system gives internally generated names to classes declared implicitly as the class associated with the attribute `bendpoints` in Example 3.1. Implicitly declared classes with same structures are given the same internally generated name, hence they are equivalent.

3.3 Objects

An object's value may or may not be defined. An *object* whose value is defined is a triple (o, v, c) where $o \in \mathbf{O}$ is the object's oid, $v \in \mathbf{V}$ is the object's value, and $c \in \mathbf{C}$ is the object's class.

If the value of the object is undefined, then its oid must be `nulltuple`, or `nullset`, or `nulllist`. An object whose oid is `nulltuple` is an object of some tuple-class. Similarly, an object with oid `nullset` is an object of some set-class, and an object with oid `nulllist` is an object of some list-class. An object whose value is undefined is called an *undefined object*.

Equality and Copying: The system provides three equality operators and two copy operators. The equality operators test for id-equality, shallow-equality, and deep-equality. The copy operators return shallow-copy and deep-copy of a given object. These operators have their standard meaning as in [15], interested readers may also see [17].

3.4 Class Extensions

A database schema explicitly declares a set of classes $C \in \mathbf{C}$. The system maintains an extension for each class in C. The extension is the set of all objects of its associated class. We define a function π_d, called the *disjoint oid assignment*, from class names to extensions. If for some $n \geq 0$, $\{o_{1_c}, o_{2_c}, \ldots, o_{n_c}\}$ is the extension of class c, then $\pi_d(c) = \{o_{1_c}, o_{2_c}, \ldots, o_{n_c}\}$. The function π_d is called the disjoint oid assignment because, if c and c' are two different classes in C, then $\pi_d(c) \cap \pi_d(c') = \phi$.

Given π_d, the *oid assignment* π (inherited from π_d) is a function mapping each class name to a set of oids such that $\pi(c) = \pi_d(c) \cup \{\pi_d(c') \mid c' \in C, c' < c\}$. In other words, π maps a class name to the set of objects of that class and all of its subclasses. Given π, the interpretation of a class c is defined as follows:

- each basic type `Integer`, `String`, `Float` and `Boolean` has its natural domain
- for each tuple-class c, $domain(c) = \{\texttt{nulltuple}\} \cup \pi(c)$
- for each set-class c, $domain(c) = \{\texttt{nullset}\} \cup \pi(c)$
- for each list-class c, $domain(c) = \{\texttt{nulllist}\} \cup \pi(c)$

3.5 Database

A database consists of a schema S and an instance I. There is a clear separation between the schema and the instance in our data model.

3.5.1 Schema

A database *schema* is a 3-tuple (C, σ, G) where $C \in \mathbf{C}$ is a set of class names, σ is the function mapping class names to class definitions, and G is a set of *global variables* with associated classes.

The sets C and G together act as the entry points to the database. Every object in the extensions of C as well as every object with a global name is *persistent*. Every object that is a part of a persistent object is also persistent.

The function σ maps class names to class definitions. Class hierarchy in C can be constructed from the information available with the class definitions.

3.5.2 Instances

An *instance* of a database schema consists of a finite set of objects and the four functions π_d, π, ν, and γ. The functions π_d, π, ν, and γ are defined as follows:

- the function π_d is the disjoint oid assignment
- the function π is the oid assignment inherited from π_d
- The function ν maps oids to values for all the defined objects in the database instance

- the function γ maps variable names in G to objects which are the values currently assigned to the variables

3.6 Other Data Models

The data model presented in this section eliminates the dichotomy of *object-versus-values* found in other data models [3, 8]. This simplifies the model, and removes any possibility of confusion between objects and values as pointed out in [15]. It is however possible to achieve the functionality and performance of *values* in our data model.

Values: Values can be thought of as special objects which are never shared and have no behavior. Since values are never shared, it is not necessary to refer to them indirectly from multiple objects. It is therefore cheaper to access values. Since values have to behavior, they do not carry any type related information at runtime. It is never necessary to perform a late binding of any method on a *value* based on its actual type. A value of any type can be assigned to a variable of its supertype. The assignment however truncates the value if necessary.

In our data model, a value is an object of an implicitly declared class (which specifies only a structure, but no name, no supertype and no method) and qualified as `Own`. The attribute `bendpoints` of class `SegmentedEdge` in Example 3.1 holds exclusively referenced objects. This provides objects that are never shared and have no behavior. These objects therefore can be stored and accessed like values in O_2 data model. Since the implicitly declared classes have no names, there extensions are also not stored as a part of the database instance. We point out, however, that this is only an internal optimization. To the outside users these special objects are no different than the other objects.

Shared versus Own: Sharing and exclusive access are orthogonal issues to information representation. We think these two issues should be kept orthogonal in data models. Our data model does this by providing a separate `Own` qualifier for attributes.

It is possible to provide *composite object* of ORION [4], as well as *own ref* [8] objects of EXODUS within object-oriented paradigm. To implement composite objects, one has to introduce the concept ownership (not exclusive access) of an object by another object. The *own ref* objects require both exclusive access and ownership, so that when the owner object is deleted, the owned objects are also deleted. In principle, such behavior are achieved by adding additional constraints on creation, manipulation, and deletion of objects in the same basic object graph.

4 Object Algebra

The operators are categorized into six sets based on the types of the arguments they admit.

4.1 Object Operators

CREATE(*class-name*, *value*) $\rightarrow$ *object* **:** Creates an object of the given class and value.

CREATE(*class-name*) $\rightarrow$ *object* **:** Creates an undefined object of the given class.

SH_COPY(*object*) $\rightarrow$ *object* **:** Creates a shallow-copy of the given object.

DEEP_COPY(*object*) $\rightarrow$ *object* **:** Creates a deep-copy of the given object.

INVOKE(*object*, *method-name*, {*object*}) $\rightarrow$ *object* **:** Invokes a method call, and returns the result. The objects following the *method-name* are the actual arguments for the call.

FILTER(*object*, *predicate*) $\rightarrow$ *object* | *null* **:** Returns the given object if the given predicate is satisfied, else it returns *null*.

CLASS_NODE(*class-name*) $\rightarrow$ *set-object* **:** Returns π_d(*class-name*) for a class in the database schema.

CLASS_SUBTREE(*class-name*) $\rightarrow$ *set-object* **:** Returns π(*class-name*) for a class in the database schema.

4.2 Tuple Operator

TUP_CONSTRUCT({*object*}) $\rightarrow$ *tuple-value* **:** Returns a tuple-value constructed from the given objects. The order of the components in the returned value is same as the order of the arguments.

TUP_ATTR(*tuple-object*, *attr-name*) $\rightarrow$ *object* **:** Returns the value of the attribute.

TUP_COMP(*tuple-object*, *position*) $\rightarrow$ *object* **:** Temporary tuple-classes created by the algebra are not given explicit attribute names. Every component of a tuple-value has an associated *position*. This operator is used to extract individual components by position.

TUP_ADDR(*tuple-object*, *attr-name*) $\rightarrow$ *l-value* **:** It returns the l-value of the given attribute. It is used to assign values to the attribute using the ASSIGN operator.

4.3 Set Operators

A set is a collection of objects without duplicates.

SET_CONSTRUCT({*object*}) $\rightarrow$ *set-value* **:** Returns a set-value costructed from the given objects.

SET_UNION(*set-object*, *set-object*) $\rightarrow$ *set-object* **:** Returns the union of the given objects.

SET_DIFF(*set-object*, *set-object*) $\rightarrow$ *set-object* **:** Returns the difference of the given objects.

SET_PRODUCT(*set-object*, *set-object*) → *set-object* : Returns the cartesian product of the given objects.

SET_DE(*set-object*, *equality-op*) → *set-object* : This operator is used to eliminate duplicates. Although each object is unique, some objects may be shallow-equal, or deep-equal to each other.

SET_COLLAPSE(*set-object*) → *list-object* : This operator takes a set-valued object whose value consists of a set of set-valued objects. It produces list-object whose value is the concatenation of the values of those set-valued objects.

SET_APPLY(*set-object*, *op-sequence*) → *set-object* : It applies a given operator sequence on the elements of the value of a set-valued object to produce a set-valued object.

SET_TO_LIST(*set-object*) → *list-object* : Returns a list-valued object created from the objects in the set-value.

4.4 List Operators

A list is a sequence of elements of variable length and may have duplicate elements.

LIST_CONSTRUCT({*object*}) → *list-value* : Returns a list-value constructed from the given objects. The order of elements in the list is same as the order of the arguments.

LIST_CAT(*list-object*, *list-object*) → *list-object* : Returns a list-valued object whose value is created by concatenating the second list to the first list.

LIST_DIFF(*list-object*, *list-object*) → *list-object* : This operator returns a list which has all the elements of the first list except the elements of the second list. The order of the remaining elements in the returned list in same as the order of the elements in the first list.

Example 4.1: Suppose we have two lists L_1 and L_2 with values <1 3 2 1 2 3 5 5 7> and <3 5 3 6 6> then LIST_DIFF(L_1, L_2) is a list with value <1 2 1 2 7>. Since 3 and 5 are elements of L_2, they cannot be elements of the resulting list. The order of the remaining elements is preserved.

LIST_PRODUCT(*list-object*, *list-object*) → *list-object* : Returns a list which is an *ordered cartesian product* of the given lists. This is not a commutative operation.

Example 4.2: Suppose we have two lists L_1 and L_2 with values <4 2 5> and <7 9 6>, then LIST_PRODUCT(L_1, L_2) has value $<o_1, o_2, \ldots, o_9>$ where o_i for $i = 1, 2, \ldots, 9$ are the oids of tuples with values [4 7], [4 9], [4 6], [2 7], [2 9], [2 6], [5 7], [5 9], and [5 6] respectively.

LIST_DE(*list-object*, *eq-op*) → *list-object* : Eliminates duplicates from lists. The first object is retained.

LIST_COLLAPSE(*list-object*) → *list-object* : Returns a list by concatenating all the lists within the input list in the same order.

LIST_APPLY(*list-object*, *op-seq*) → *list-object* : Returns a list of results of application of operator sequence to list elements.

LIST_TO_SET(*list-object*) → *set-object* : Produces a set from the elements of the list.

4.5 Mixed Operators

SET_LIST_COLLAPSE(*set-object*) → *list-object* : Returns a list, created by concatenating the lists in the given set in any arbitrary order.

LIST_SET_COLLAPSE*list-object*) → *list-object* : Returns a list by converting the sets to lists and concatenating them in the order of the given list.

4.6 Other Operations

ASSIGN(*variable*, *object*) : Assigns a value to a variable. It operates by creating side effect. It does not return any result.

REPEAT_UNTIL(*expr-sequence*, *predicate*) : Evaluates the expressions repeatedly till the predicate is satisfied. This operator is used to evaluate recursive queries.

5 Query Language OQL

The Object Query Language OQL is rule-based. It allows stratified negation. It is statically typed. The language allows one to manipulate mixture of tuple-valued, set-valued and list-valued objects obeying certain typeing restrictions. It also includes mechanisms to express queries involving inheritance relationships of classes. Finally, the language provides mechanisms for creating objects of arbitrary types.

5.1 Syntax

The syntax for an OQL program is given below. Detailed semantics are described in Section 6. A *program* consists of a *sequence* of statements. Each *statement* is either an *assignment* or a *rule*.

An assignment assigns an *r-value* to an *l-value* if the *qualifiers* are satisfied. The *object expression* in the assignment statement provides an r-value, and the *path expression* provides the l-value. The symbol ":=" is the assignment operator.

A rule has a *head* and a *body*. A head is a special type of literal with a path expression, an optional equality operator, and an object expression. There are three system defined equality operators for id-equality, shallow-equality, and deep-equality.

```
<prog> → {<statement>}
<statement> → <assign>
            → <rule>
<assign> → <path-expr> ':=' <obj-expr> {<qual>}
<rule> → <head> '←' <body>
<head> → <path-expr> '(' [<eq-op>] <obj-expr> ')'
<body> → {<literal>}
<literal> → <gen>
          → <qual>
<gen> → [¬] <class-expr> '(' <var-name> ')'
      → [¬]<obj-expr> '(' <var-name> ')'
<qual> → [¬] <obj-expr> <op> <obj-expr>
<path-expr> → <var-name>
            → <path expr> '.' <attr name>
<obj-expr> → <basic-value>
           → <var-name>
           → <obj-expr> '.' <attr-name>
           → <obj-expr> '.' <meth-call>
           → 'new' <class-name> '(' <value> ')'
           → <copy-op> <obj-expr>
<class-expr> → <class-name>
             → <class-name> '*'
<value> → <empty>
        → '[' {<attr-name> ':' <obj-expr>} ']'
        → '{' {<obj-expr>} '}'
        → '<' {<obj-expr>} '>'
<meth-call> → <meth-name> '(' {<obj-expr>} ')'
```

Each body consists of a set of *literals*. Each literal is either a *generator* or a *qualifier*. Our data model has no *relations*. There are only set-valued and list-valued objects. Generators therefore have one *variable* within parenthesis. This variables are called *range variables* of the generators. The symbol "¬" denotes *negation*. All the range variables of non-negated generators of a single rule body must be different. A qualifier specifies a condition.

A *class expression* stands for a set of objects of that class. If c is a class name, then the expression c stands for the extension of that class, and $c*$ stands for the union of the extension of the class and the extensions of all its subclasses.

The "." operator in the path expression either extracts an attribute from a tuple-value, or invokes a method on an object. A method call needs the method name, and the required number of object expressions for actual arguments.

The "`new`" operator creates a new object with the given class and *value*. A value is a structured value. If the value is empty, the "`new`" operator returns an undefined object of the given class.

5.2 Typing Restrictions

OQL programs are statically typed. There is a type associated with each path expression, object expression, class expression, and basic value.

A object expression of class c can be assigned to a path expression of class c or a superclass of class c only.

In the head of a rule, if τ is the type associated with the object expression, then the type associated with the path expression must be either `Set`(τ') or `List`(τ') where τ' is a supertype of τ.

In a generator literal, if τ is the class name used in the class expression, then the type associated with the class expression is `Set`(τ), and the type associated with its range variable is τ. Similarly, if `Set`(τ) is the type associated with the object expression, then the type of its range variable is τ.

In a qualifier, the equality operators must be applied to object expressions of same type.

In path and object expressions, the attributes and methods must be defined at the appropriate classes, and the arguments for the method calls must be object expressions of required types.

In an object expression, the "`new`" operator must be provided with a value appropriate for the given class.

Finally, if the head of a rule has a path expression of type `List`(τ) for some type τ, then the generators in the rule body may only be object expressions followed by range variables within parenthesis, and the types of the object expressions can only be list-classes. This restriction is necessary in order to define a deterministic order for the elements of the list in the head. Since sets are unordered, allowing sets would make the order of the elements non-deterministic. It is however permissible to use list-valued objects in body, and path expressions of set-class in the head.

5.3 Examples

Example 5.1 Set Filter: Let `V` be a set of vertices. `Vertex` class is declared in Example 3.1. We want to create a set of objects with the `label`, and `position` of vertices of `radius` greater than 10.0. This query is expressed as follows:

```
Class City Tuple(name: String, loc: Point);
C: Set(City);
C(=s new City([name: v.label,
              loc: sh-copy(v.position)]))
  ← V(v), v.radius > 10.0
```

This query declares a new class `City`, and a new variable `C` of the newly declared class. It uses the "`new`" operator and the "`sh-copy`" operator to create new objects. The copy operator is necessary because the `position` attribute of the `Vertex` class is qualified as `Own`. This qualification prevents sharing of the objects held by the position attribute. The program also uses the shallow-equality operator "$=_s$" to eliminate duplicates. □

Example 5.2 Nesting and Unnesting: Let `T` be any arbitrary type in the following declarations.

```
Class C1 Tuple(a1: T, a2: Set(T));
Class C2 Tuple(b1: T, b2: T);
R1, R3: C1;
R2: C2;
```

Assume that initially `R1` holds a set of objects of class `C1`, and other two variables hold undefined objects. We want to unnest `R1` into `R2`, and then nest `R2` into `R3`. The program below unnests `R1` into `R2`:

```
R2(≡ new C2([b1: x.a1, b2: y]))
  ← R1(x), x.a2(y)
```

The following program nests `R2` into `R3`:

```
R3(=s new C1([a1: x.b1, a2: nullset]))
  ← R2(x)
y.a2(≡ sh-copy(x.b2))
  ← R3(y), R2(x), y.a1 =s x.b1
```

Notice the unnesting program uses the id-equality operator, and the nesting program uses the shallow-equality operator in the first rule, and id-equality in the second rule along with a copy operator. The copy operator ensures that nesting and unnesting operation will be inverse operations. □

Example 5.3 Transitive Closure: Consider a set of objects `E` of class `Edge` as in Example 3.1. We want to compute the transitive closure of `E` and create a new set of objects `R` of class `Reachable` as declared below. The following program performs is computation:

```
Class Reachable Tuple(s: Vertex, d: Vertex);
R(=s new Reachable([s: e.source, d: e.dest])
  ← E(e)
R(=s new Reachable([s: r.source, d: e.dest])
  ← R(r), E(e), r.d ≡ e.source
```

The program above uses the shallow-equality operator in the heads. This ensures that `R` does not contain duplicate (*source, destination*) pairs. The program below is another version of the query.

```
R(≡ new Reachable([s: e.source, d: e.dest])
  ← E(e)
R(≡ new Reachable([s: r.source, d: e.dest])
  ← R(r), E(e), r.d ≡ e.source
```

This program uses the id-equality operator in the heads. This program therefore may cause R to contain duplicate (*source, destination*) pairs if there is more than one path from source to destination. This program will also never terminate if there are cycles in the graph. □

Example 5.4 Inheritance Tree Queries: This example uses the `Edge, SegmentedEdge`, and `CurvedEdge` classes defined in Example 3.1. The query below retrieves all the database edges which are neither segmented nor curved, and of length less than 20.0:

```
E1: Edge;
E1(≡ e) ← Edge(e), e.length() < 20.0
```

All database edges of length less than 20.0 are retrieved by the following query:

```
E2: Edge;
E2(≡ e) ← Edge*(e), e.length() < 20.0
```

All database edges of length less than 20.0 except the segmented edges are retrieved by the following query:

```
E3: Edge;
E3(≡ e) ← Edge*(e), ¬ SegmentedEdge(e),
          e.length() < 20.0
```

This example shows the use of class extensions, and the class expressions involving "*" and "¬" operators for dealing with class hierarchies. □

OQL is capable expressing a wide range of queries such *powerset computation* and *grouping*. Interested readers can find more examples in [18].

6 Semantics of OQL

The semantics of OQL is defined by an algorithm that translates OQL programs into algebraic operations, assignment operations and REPEAT_UNTIL loops. In this section we describe an informal and intuitive meaning for OQL programs.

Variables: OQL is similar in spirit to Datalog [20]. OQL programs computes values of database variables whose values are undefined from database variables whose values are defined. These are called *intensional database variables* or IDB variables and *extensional database variables* or EDB variables respectively.

Classes: We also call the classes that are part of the database schema the EDB classes, while the classes defined in an OQL program are IDB classes. IDB classes are not part of the schema, and they do not take part in the inheritance hierarchy, their extensions are not stored in the database, and they cannot be used in the class expressions.

6.1 Expressions

Path Expressions: Path expressions are used to navigate through the object graph. A *path expression* is a variable name followed by a sequence of zero or more attribute names separated by "." operators. It has both l-value, and r-value. The l-value and the r-value of the path expression E, denoted by LVAL(E) and RVAL(E) respectively, are defined as follows:

- if it is a variable name, then LVAL(E) and RVAL(E) are the l-value and the r-value of the variable respectively

- if it is of the form <*path-expr*>`.`<*attr-name*>, then LVAL(E) = TUP_ADDR(RVAL(*path-expr*), *attr-name*), and RVAL(E) = TUP_ATTR(RVAL(*path-expr*), *attr-name*)

Object Expressions: Object expressions have only r-values. The value of an object expression E denoted by RVAL(E) is defined as follows:

- if it is a basic value, the basic value is the value of the object expression
- if it is of the form <*obj-expr*>`.`<*attr-name*> then RVAL(E) = TUP_ATTR(RVAL(*obj-expr*), *attr-name*)
- if it is of the form <*obj-expr*>$_1$`.`<*meth-name*> (<*obj-expr*>$_2$, <*obj-expr*>$_3$, …, <*obj-expr*>$_n$), then RVAL(E) = INVOKE(RVAL<*obj-expr*)>$_1$, <*meth-name*>, RVAL(<*obj-expr*)>$_2$, RVAL(<*obj-expr*)>$_3$, …, RVAL(<*obj-expr*)>$_n$)
- if it is of the form `new` <*class-name*>(<*value*>), then RVAL(E) = CREATE(*class-name*, *value*)
- if it is of the form <*copy-op*><*obj-expr*>, then RVAL(E) = SH_COPY(RVAL(*obj-expr*)) or DEEP_COPY(RVAL(*obj-expr*)) depending on whether <*copy-op*> specifies a shallow-copy or a deep-copy respectively

Class Expressions: Class expressions are used to denote class extensions. They only have r-values. Let c be the name of an EDB class. Then the value of the class expression c, denoted by RVAL(c), is CLASS_NODE(c), and the value of the class expression $c*$, RVAL($c*$) = CLASS_SUBTREE(c)

6.2 Restrictions

Monotonicity: OQL programs do not delete database objects, and do not modify the existing value of any variable of any tuple-class. They may insert objects into sets, and append objects to lists. but they do not remove objects from sets, or lists.

Safety: All the OQL programs must be *safe*. Safety is necessary to ensure finiteness of IDB sets and lists. An OQL program is safe if all the rules in the program are safe. A rule is safe if all the variables appearing in the rule are *limited*. The *limited* variables are defined as follows.

- any EDB variable, IDB variable, or range variable of a *non-negated* generator is limited
- variable `x` is limited if it appears in a qualifier such as $x \equiv E$, or $E \equiv x$, where E is an object expression
- variable x is limited if it appears in a qualifier literal such as $x \equiv y$, or $y \equiv x$, where y is a variable already known to be limited

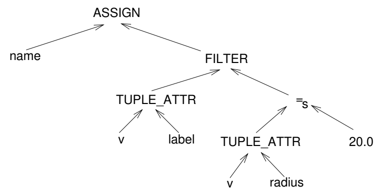

Figure 1: Query tree for a simple assignment

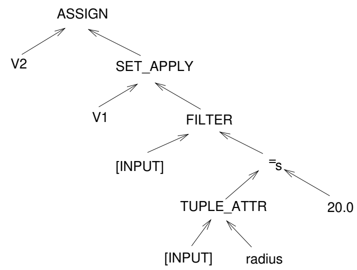

Figure 2: Expression tree for a simple rule

6.3 Simple Statements

An OQL program consists of a *sequence* of statements. Each statement is either an assignment statement or a rule. The following is a simple program with a set of declarations, an assignment statement, and a rule. Assume that `v`, and `V1` are EDB variables of type `Vertex`, and `Set(Vertex)` respectively.

`name`: `String;`
`V2`: `Set(Vertex);`
`name` := `v.label`, `v.radius` $=_s$ `20.0`
`V2(x)` ← `V1(x)`, `x.radius` $=_s$ `20.0`

An assignment statement can only define values of undefined variables. This is required to ensure monotonicity. The first statement above assigns a string to IDB variable `name` if the qualifier `v.radius = 20.0` is satisfied, else the value of `name` remains undefined. This statement is translated to the query tree in Figure 1.

The second statement is a rule. The body of the rule has the generator `V1(x)`, and the qualifier `x = 20.0`. The variable `x` is the *range variable*. The generator can be thought of as the predicate (`x` ∈ `V1`). The qualifier has the obvious interpretation as a predicate. Similarly, the head of the rule `V2(x)` can be interpreted as the predicate (`x` ∈ `V2`). Intuitively, a rule means that whenever the predicates in the body are satisfied, the head predicate must also be satisfied. This accomplished by the expression tree in Figure 2. The nodes denoted by "[INPUT]" are the elements of the set on which the SET_APPLY operator operates.

6.4 Dependency Graph and Recursion

A *dependency graph* of a program describes the way IDB and EDB variables depend on one another. There is an arc from variable P to variable Q if there is rule with a body literal P and with a head literal Q. A program is *recursive* if its dependency graph has at least one cycle. A program with acyclic dependency graph is *recursion-free*.

All variables that are on one more cycles are called *recursive variables*. A variable is non-recursive if it is not part of any cycle. A recursive program may have non-recursive variables.

6.5 Recursion-Free Negation-Free OQL

If the rules are not recursive, we can order the the nodes of the Dependency graph $P_1, P_2, \ldots, P_n$ so that if there is an arc $P_i \longrightarrow P_j$ then $i < j$. We can then compute the values for the variables $P_1, P_2, \ldots, P_n$ in that order, knowing that when we work of P_i the values for all the variables that are required to compute P_i are already known. The computation is done in two steps, i) first compute the values corresponding to the rule bodies, ii) then combine the results from the rules that compute the same IDB variable.

Preventing Loss of Objects: The semantics of the language ensures no "information loss". Coverting a list to a set may cause loss of information due to elimination of duplicates. Converting a set to a list causes no loss of information because the set can be reconstructed from the list. All implicit conversions are therefore done from set to list only.

When sets and lists and lists are mixed in the rule body, sets are implicitly converted to lists for applying the LIST_PRODUCT operator. The operators SET_COLLAPSE, LIST_COLLAPSE, SET_LIST_COLLAPSE, and LIST_SET_COLLAPSE have been designed to return lists for the same reason.

A list is converted to a set only when explicitly asked. OQL does not allow explicit conversion of sets to lists because the ordering of the elements would be unknown.

Duplicate Elimination: Duplicate elimination is expressed by specifying an equality operator in the head literal such as in the rule `L2`($=_d$ `y`) $\longleftarrow$ `L1(y)`. Assume that `L1` is an EDB variable of type `List(T)` for some type `T`. The head of this rule stands for the predicate "there is one object in `L2` that is deep-equal to `y`". This rule is translated to ASSIGN(L2, LIST_DE(L1, $=_d$)).

Join Queries: Join queries are translated to SET_PRODUCT or LIST_PRODUCT operations, and FILTER operation to select resulting tuples.

Example 6.1 Join Queries: Assume that `L1`, `L2` are EDB variables, and `L3` is an IDB variable in the following program.

```
Class C1 Tuple(a: T, b: T);
Class C2 Tuple(a: T, c: T);
Class C3 Tuple(a: T, b: T, c: T);
```

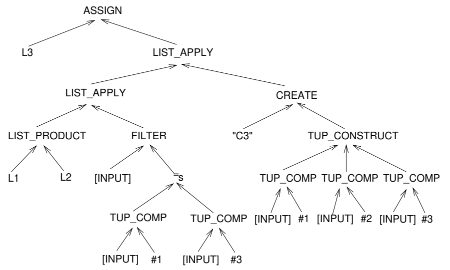

Figure 3: Expression tree for program in Example 6.1

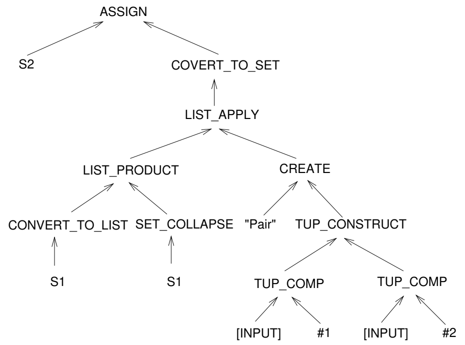

Figure 4: Expression tree for program in Example 6.2

```
L1: List(C1);
L2: List(C2);
C3: List(C3);
L3(new C3([a: x.a, b: x.b, c: y.c]))
  ← L1(x), L2(y), x.a = y.a
```

The expression tree for this program is in figure 3. □

Derived Generators: The path expressions for some generators are constructed from the range variables of other generators. The former type of generators are called *derived generators*. They are said to be derived from the generators whose range variables are used to construct them. Derived generators are translated to expressions involving COLLAPSE operators.

Example 6.2 Derived Generators: Assume that `S1` is an EDB variable of type `Set(List(T)`, and `S2` is the IDB variable in the following program. This program is translated into the expression tree in Figure 4.

```
Class Pair Tuple(a1: List(T), a2: T);
S2: Pair;
S2(new Pair([a1: x, a2: y]) ← S1(x), x(y) □
```

Order of List Elements: The order of the elements in the IDB list of a rule is same as the order of the elements in the list computed by the rule body. If the rule body has multiple generators, the elements in the list computed by the rule body is determined by the left to right ordering of the generators. The generators on the left are more significant in ordering the elements as shown in the following example.

Example 6.3 Order of List Elements: Let L1, and L2 be two lists with values <1 2>, and <1.0 2.0>. The following rule computes L3 from L1 and L2.

```
Class Pair Tuple(a1: Integer, a2: Float);
L3: Set(Pair);
L3(new Pair([a1: x, a2: y])) ← L1(x), L2(y)
```

The above program assigns the list of oids $< o_1, o_2, o_3, o_4 >$ to L3, where value of the objects are [1 1.0], [1 2.0], [2 1.0], [2 2.0]. In the program below the order of the generators in the body has been changed.

```
L3(new Pair([a1: x, a2: y])) ← L2(y), L1(x)
```

This program assigns a list of four oids to L3, but their values are [1 1.0], [2 1.0], [1 2.0], [2 2.0] respectively. □

Combining Results from Multiple Rules: An IDB variable may appear at the head of several rules. If such an IDB variable is a set, its value is computed by taking union of the values computed by the individual rule bodies. If the variable is list, its value is computed by concatenating the values computed by the individual rules. The order of concatenation is same as the order the order of appearance of the rules in the program.

6.6 Recursive Negation-Free OQL

Recursive programs are evaluated by computing *least fixed points* of OQL equations with a REPEAT_UNTIL loop. Assume that a program computes values of m IDB variables $P_1, P_2, \ldots P_m$. Initially all the IDB variables are undefined. We introduce a temporary variable Q_i for each IDB variable P_i. Q_i is assigned the current value of P_i for all $i = 1$ to m at the beginning of the loop. Assume that EXPR(P_i) is the algebraic expression for computing the value of variable P_i from the currently assigned values of Q's, and other EDB variables, basic values, method names, attribute names and class names. EXPR(P_i) is constructed in the same way as for the recursion-free programs. The program below shows the least fixed point computation. Due to monotonicity, IDB variables will only grow in size, hence if the program doesn't specify an infinite computation, a fixed point will be reached.

REPEAT
  ASSIGN(Q_1, P_1);
  ASSIGN(Q_2, P_2);
  ⋮
  ASSIGN(Q_m, P_m);
  ASSIGN(P_1, EXPR(P_1));
  ASSIGN(P_2, EXPR(P_2));
  ⋮
  ASSIGN(P_m, EXPR(P_m));
UNTIL $(P_1 = Q_1) \wedge (P_2 = Q_2) \wedge \ldots \wedge (P_m = Q_m)$

The order of elements in IDB lists is same as for non-recursive programs for each individual iteration of the loop, and each successive iteration appends the newly computed elements at the end. OQL programs never insert elements into lists at arbitrary locations.

6.7 OQL with Negation

OQL programs with negation must be *safe* and *stratified*. Rules are stratified if whenever there is a rule with head IDB variable P and a negated subgoal with predicate Q, there is no path in the dependency graph from P to Q. One can use the algorithm given in [20] to test for and find stratification. A stratified program is evaluated stratum by stratum, starting from the lowest to the highest.

When stratum i is being evaluated, the values for the IDB variables at lower strata have already been computed. Let $\neg Q(x)$ be the negated generator at stratum i. The range variable x must appear in exactly one non-negated generator by the syntax rules and the safety criteria. Let this generator be $P(x)$. Then the value of the negated generator is SET_DIFF(P, Q) or LIST_DIFF(P, Q) depending on whether P and Q are sets or lists respectively.

6.8 Equivalence of OQL and Algebra

Reduction of OQL to algebra : Reduction of OQL to the algebra is proved by the algorithm for translating OQL programs to the algebra. We have already outlined the basic approach. Readers may find a formal algorithm in [18].

Reduction of algebra to OQL : Reduction of the algebra to OQL is proved by case-based induction. We omit most of the cases as our goal is to simply give the flavor of the proof. Details can be found in [18]. The proof proceeds by induction on number of operators in an algebraic expression E.

An algebraic expression consists of EDB variables, IDB variables, class names, attribute names, method names, basic values, equality operators, predicates, and one or more algebraic operators. Each expression must have an ASSIGN operator at the root of the expression tree.

Base Case: One ASSIGN operator in E.

- Algebra: ASSIGN(v_1, v_2) or ASSIGN(v_1, b)

  In this case v_1 must be an IDB variable, v_2 must be an EDB variable, and b is a basic value.

  OQL: $v_1 := v_2$ or $v_1 := b$

Inductive Case : Two or more operators in E. The last operation to be performed in a expression is always an assignment. The following are some expressions and their equivalent OQL programs:

- Algebra: ASSIGN(E_1, CREATE(*class-name*, E_2))
  OQL: E_1 := `new` *class-name*(E_2)
- Algebra: ASSIGN(E_1,TUP_ATTR(E_2,*attr-name*))
  OQL: E_1 := E_2.*attr-name*
- Algebra: ASSIGN(E_1, SET_PRODUCT(E_2, E_3))
  OQL: E_1(new TempClass([#1: x, #2: y]))
  $\leftarrow E_2(x), E_3(y)$
- Algebra: ASSIGN(E_1, SET_COLLAPSE(E_2))
  OQL: $E_1(y) \leftarrow E_2(x), x(y)$
- Algebra: ASSIGN(E_1, SET_APPLY(E_2, *op-seq*))
  OQL: E_1(*op-seq*(x)) $\leftarrow E_2(x)$
- Algebra: ASSIGN(E_1, LIST_CAT(E_2, E_3))
  OQL: $E_1(x) \leftarrow E_2(x)$
  $E_1(x) \leftarrow E_3(x)$

6.9 Other Languages and Algebra

OQL is partially influenced by IQL [1]. OQL is however simpler than IQL, because it manipulates only objects, it never makes explicit reference to oids, and it does not manipulate relations. It is also more powerful because it can express duplicate elimination, manipulate lists, handle inheritance trees. IQL is more powerful than languages in [5, 14]. User-level query laguages such as EXCESS [8] and [2] are can be seen as sublanguages of OQL. Shaw and Zdonik [19], and Vadenberg and DeWitt [21] provide object-algebra for object-oriented databases. But neither of them attempt to manipulate mixture of sets and lists.

A few other languages attempt to incorporate object-oriented features into logic programs. But that is not the goal of OQL. OQL uses the logic programming paradigm simply to query database objects with a declarative language.

7 Conclusions

We have presented a data model, a query language, and an algebra. The data model simplifies the existing data models without loss of modeling power or performance. The algebra has new operators for manipulating mixture of sets and lists. The query language OQL is the most significant contribution of this paper. To our knowledge no other query language for object-oriented databases has been able to combine the power and simplicity of OQL.

We are currently in the process of implementing the system. This paper has not discussed query optimization. We are investigating possible rewriting rules for optimizing query trees.

OQL is designed to be an *ad hoc* query language. It is not suitable for performing computation. We are also planning to integrate C++ and OQL to provide a combined language for querying and computation.

The data model and OQL have been designed with a practical application in mind, *i.e.*, program visualization. We use OQL for both definition and construction of program abstractions. The suitability of a database query language for definition of abstractions remains to be seen. Shaping OQL to be an effective abstraction definition language will be our primary research goal in future.

References

[1] S. Abiteboul, and P. Kanellakis. Object identity as a query language primitive. *Proc. of ACM SIGMOD Conf.*, 1989.

[2] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. *Proc. of 2nd DBPL Workshop*, 1990.

[3] F. Bancilhon, C. Delobel and P. Kanellakis. Introduction to the Data Model. *Building an Object-Oriented Database System The Story of O2. Morgan Kaufman*, pp. 61–75, 1992.

[4] J. Banerjee, H-T Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H-J Kim. Data model issues for object-oriented applications. *ACM Transactions on Information Systems*, 5(1), pp. 3–26, January, 1987.

[5] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL1). *Proc. of ACM PODS Symposium*, 1987.

[6] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(1), January, 1988.

[7] L. Cardelli, and P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17(4), pp. 471-522, December, 1985.

[8] M. Carey, D. DeWitt and S. Vandenberg. A data model and query language for Exodus. *Proc. of ACM SIGMOD Conf.*, 1988.

[9] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derret, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An object-oriented database management system. *ACM Transactions on Information Systems*, 5(1), pp. 48–69, Januray, 1987.

[10] W. Kim, J. Banerjee, H-T Chou, J. F. Garza, and D. Woelk. Composite object support in an object-oriented database system. *Proc. of ACM OOPSLA Conf.*, 1987.

[11] C. Lecluse, and P. Richard. Manipulation of structured values in object-oriented databases. *Proc. of 2nd DBPL Workshop*, 1989.

[12] C. Lecluse, and P. Richard. Modeling complex structures in object-oriented databases. *Proc. of ACM PODS Symposium*, 1989.

[13] G. M. Kuper, and M. Y. Verdi. A new approach to database logic. *Proc. of ACM PODS Symposium*, 1984.

[14] G. M. Kuper. The logical data model: A new approach to database logic. *PhD Thesis, Stanford University*, 1985.

[15] O_2 Technology. The O_2 User Manual. June, 1992.

[16] S. Reiss, and M. Sarkar. Generating abstractions for visualization. Technical Report CS-92-35. *Computer Science Department, Brown University*, September, 1992.

[17] M. Sarkar, and S. Reiss. A data model for object-oriented databases. Technical Report CS-92-56. *Computer Science Department, Brown University*, December, 1992.

[18] M. Sarkar, and S. Reiss. A query language for object-oriented databases with tuples, sets and lists. Technical Report CS-92-57. *Computer Science Department, Brown University*, December, 1992.

[19] G. M. Shaw, and S. B. Zdonik. A query algebra for object-oriented databases. *Proc. of Intl. Conf. on Data Engineering*, pp. 152–162, 1990.

[20] J. D. Ullman. Principles of data and knowledge-based systems. *Computer Science Press*. ISBN0-7167-8158-1, 1988.

[21] S. L. Vandenberg, and D. J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. *Proc. of ACM SIGMOD Conf.*, 1991.