Issues in Query Model Design in Object-Oriented Database Systems

M. Tamer Özsu
Dave D. Straube*
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

The provision of a powerful query system, including a declarative query language and the systems support for efficiently processing queries is essential for the success of object-oriented database technology. Designing a query system involves making a large number of decisions, further complicated by the lack of a universally accepted object-oriented object data model. In this paper, object data model and query model design decisions are enumerated, alternatives are identified, and the tradeoffs are specified.

Keywords: object-oriented database systems, query languages, object calculus, object algebra.

1 Introduction

A feature that the relational systems [?] introduced to database management has been a query system that includes a declarative language and the system support to process queries efficiently. The early research on object-oriented database systems (OODBMS) has avoided this issue, concentrating instead on modeling capabilities. The lack of a universally accepted and formally specified object data model[1], comparable to the relational model, has certainly contributed to the lack of research on query processing. Existing object data models differ in their formalism, support for object identity [?], encapsulation of state and behavior [?], type inheritance [?] and typed collections.

The success of the OODBMS technology will depend, among other things, on the provision of a powerful query system [?]. In this paper, we address precisely this issue. Specifically, we discuss the

*Current Address: Banyan Systems Inc., 115 Flanders Road, Westboro, MA 01581, USA.

[1]In this paper we use the term "object data model" instead of "object model" or "object-oriented data model" in accordance with the terminology used by ANSI/X3/SPARC/DBSSG OODB Task Group.

design of query models, their formalization, and the impact of object data model design decisions on the query model. The contents of this paper is an elaboration of [?]. A more detailed discussion of these issues which also emphasizes their processing can be found in [?].

Figure 1: Query processing methodology

Our research investigates the feasibility of a query processing methodology as depicted in Figure ??, which is an extension of the relational query processing strategy [?]. The steps of the methodology are as follows. Queries are expressed in a declarative language which requires no user knowledge of object implementations, access paths or processing strategies. The calculus expression is first reduced to a normalized form by eliminating duplicates, applying identities and rewriting. The normalized expression is then converted to an equivalent object algebra expression. This form of the query is a nested expression which can be viewed as a tree whose nodes are algebra operators and whose leaves represent the extents of types in the database. The algebra expression is next checked for type consistency to insure that predicates and methods are not applied to objects which do not support the requested function. This is not as simple as type checking in general programming languages since intermediate results, which are sets of objects, may be composed of heterogeneous types. The next step in query processing is the application of equivalence-preserving rewrite rules to the type-consistent algebra expression. Lastly, an execution plan which takes into account object implementations is generated from the optimized algebra expression. Detailed description of the object data model, the query model including an object calculus and an object algebra definition, as well as the rules for algebraic optimization of queries are provided in [?]. Typechecking rules are covered in [?] and the generation of execution plans is in [?].

The unique perspective of this paper is the discussion of design issues and alternatives for object data models (Section ??) and query models (Section ??). We describe our own object data model and in Section ?? demonstrate a consistent set of design decisions. Finally, in Section ??, we provide some concluding remarks.

2 Object Data Model Issues

The power and flexibility of object-oriented systems introduce considerable complexity into their object data models. Even the feasibility of defining an "object-oriented data model" in the same sense as the relational model is in question [?]. A comprehensive discussion and treatment of all

these issues is beyond the scope of this paper. The discussion in this section is restricted to those object data model issues that relate directly to the definition of query models.

The following aspects of the object data model have a direct bearing on the query model and the capabilities that need to be included in a query processor. These issues are not entirely independent of each other. We illustrate how one design decision can affect others.

Nature of an "object". There are different definitions of an "object". Some object data models consider objects simply as complex data structures [?, ?, ?], somewhat similar to the nested relation models that permit relation-valued attributes. This approach is common to those models that are developed to deal with complex object structures as they exist, for example, in engineering applications. Other data models consider objects to be instances of *abstract data types* (ADTs) [?, ?], which encapsulate the representation of the objects together with a set of public methods that can be used to access them. In this case, the type is a template for its instances.

The variation in the level of encapsulation enforced by object data models effects query models in the following sense. The query model must fully describe the visible components of objects which can be accessed by query primitives. For example, if objects are tuple-valued [?], then query expressions can directly access tuple fields by name. Furthermore, the allowable query primitives are dependent upon this decision. In principle, maintaining the data abstraction paradigm would require querying the database based on object behaviors, not their structure. Complete encapsulation, therefore, would require that the comparison operators in the query language be based only on identity ("are two objects the same?") not on structure. There are query models that provide a relaxed form of encapsulation by enabling some sort of structure-dependent equality check.

Strong versus weak typing. If the object data model treats objects as instances of ADTs, then does it require that each object be of a given type? Most object object data models require objects to belong to predefined types, but there are others which permit one-of-a-kind objects (also called *prototypical objects*) that define their own types and are not associated with any predefined type in the system. Some models also permit *variant* objects that belong to a type, but vary from the template defined by the type in some manner. In systems that allow variant and prototypical objects (e.g., [?, ?]), it is no longer possible to specify the full behavior of each object based on its type and this influences the types of optimizations that can be performed.

Furthermore, the definition of the "schema" when prototypical and variant objects are supported

needs to be clearly worked out. When variant objects are allowed in the object data model, the type is no longer a template for its instances as we claimed before; it is only a **minimal** template. Therefore, it only defines the minimal behavior of the objects that belong to this type. This is important because if a query language "takes as input a schema (and a database) and generates as output another schema (and another database)" [?] then the definition of the schema determines part of the query input.

Uniformity of the model. Is everything in the system an object? Some object data models (e.g., OODAPLEX [?], FUGUE [?], FROOM [?]) treat as objects types, methods, and anything else that can be defined in the system. Such models bring uniformity to the treatment of objects. This is in contrast to other models where concepts such as methods and types are treated as meta-information separate from objects.

The uniformity of the object data model affects the query model in various ways. High-uniformity clearly provides more flexibility in querying what is basically meta-information. This could provide possibilities for their manipulation via the query formalism. Furthermore, if, for example, methods are objects, then the query model should be able to handle them (i.e., invoke them, access their bodies, etc). As observed in [?], this requires language capabilities that allow method invocation, supplying parameters to these methods and being able to deal with the returned results. These requirements make the query formalism more powerful, but also more complex.

Single versus multiple types. Objects can either belong to a single type or to multiple types. If two types such as `Employee` and `Student` are defined, then object data models which restrict objects to be of a single type would force a Teaching Associate to belong to one of these types. In other object data models, objects can be of multiple types at once, allowing a Teaching Associate to be both an `Employee` and a `Student` at the same time. This may be a more flexible and, in some sense, more natural representation of the real world, but it is also more difficult to handle in a query model since it is necessary to deal with objects that have multiple behaviors.

We should note that belonging to multiple types in this context does not imply a subtype/supertype relationship. Type A is said to be a *subtype* of type B if the behavior of A (as defined by its methods) is included in the behavior of B. B, in this case, is called the *supertype*. Subtyping establishes an "IS-A" relationship between A and B: "A is a B." Thus, by definition, any object of type A is also an object of type B. The subtype/supertype relationships between types form a hierarchy

(actually a lattice) such that a parent type in the lattice is a supertype of all its children types. This is not what we mean by belonging to multiple sets. The subtype/supertype relationship can be used, however, to model objects which need to be of more than one type. Returning to the example, `Employee` and `Student` types are not in a subtype/supertype relationship. To model a Teaching Associate as both an employee and a student a `TeachingAssociate` type can be created with two supertypes: `Employee` and `Student`.

The difference between an object data model that allows objects to belong to multiple types and one which "simulates" the same effect by creating subtypes is subtle. In the latter, there is a new type in the type lattice which becomes part of the schema; in the former, no such type definition exists in the schema. If we assume the existence of a system-defined `MyType` function that maps an object to its type(s), in the example that we are considering, the result of applying this function to an object which represents a Teaching Associate would be different in the two cases. `MyType` would return {`Employee, Student`} if objects can belong to multiple types, but its result would be {`TeachingAssociate`} if explicit subtypes are created.

Classes versus collections. Some object data models have the concept of a "class" as an *extension* of a type. Therefore, a class represents the set of instances of a type. In these models, the distinction between a class and a type become fuzzy. In other object data models, there is no explicit notion of a class (or class is not the only collection of objects of a given type) and objects are grouped into arbitrary collections. Therefore, queries can be specified on any one (or more) of these collections. The difference from the perspective of query models is the following. Let us assume that there is a type `Employee` with properties `Name, Salary, Department`. In the first case, there is an `Employee` class over which queries are defined. Thus, a query which asks for names of employees in the shoe department will retrieve **all** employees who satisfy the predicate. In the second case, however, no `Employee` class exists. Maybe there are explicit collections such as `BostonEmployees, ParisEmployees, TorontoEmployees` and the query either has to be defined over one of them (resulting in the retrieval of only the employees in that collection) or the user has to explicitly specify the query on all three collections and union the results. Note that this distinction is important in the context of defining what the schema is. If the data model treats these collections uniformly as objects, and a relationship is established between these collections and the type lattice, then the schema can be defined as consisting of the type lattice and an object collection lattice which are connected at the root via a system defined `Object` object. This would

enable the specification of queries on types (as well as on individual collections), which are then executed on all the collections that are of that type.

It has been pointed out that object data models that employ "class as the extent of type" approach make it easier to conceptualize a query [?]. Furthermore, it is possible in these models to exploit the subtype/supertype relationship [?]. If, by definition, an object of type A is also of type B when A is a subtype of B, then the result of a query which asks for all objects of in a class of type B can also include objects of a class of type A. This can be extended to multiple levels of subtype/supertype lattice. Thus, the query result is the union of all objects that are in all the classes of types in the subtree of the type lattice rooted at the type (class) at which the query is posed. We call this the *deep extent* of a class (type) in [?].

Mechanism for sharing. One of the strengths of object-oriented models is that they provide mechanisms for sharing among objects. Two types of sharing are possible: sharing of implementation and sharing of behavior. Behavioral sharing is what we called subtype/supertype relationship above. It is important to differentiate this from implementation sharing. As noted by Nierstrasz [?], "many of the 'problems' with inheritance arise from the discrepancy between these two notions." He goes further and associates subtyping with types and inheritance with classes. This association is certainly valid in models that support "classes as the extent of type" approach. Unfortunately, early object-oriented languages such as Smalltalk [?] bundle these two concepts.

There have been two proposals for implementing sharing, one based on *inheritance* (e.g., Smalltalk) and the other based on *delegation* [?]. In inheritance, the sharing is based on a lattice of types. Thus, type A which is a child of type B in the hierarchy, inherits (behavior or implementation) from B. In delegation, sharing is achieved by an object explicitly delegating its behavior or implementation to another object. By and large, the choice of the sharing mechanism has been tied to the choice of the type system. Those models which allow variant and prototypical objects typically implement delegation as the sharing mechanism, while those that enforce strong typing implement inheritance. There has been some recognition that the two mechanisms are similar [?, ?], but the semantics of an object data model that permits prototypical and variant objects, but implements inheritance as the sharing mechanism is quite complicated. This is especially the case if the object data model does not treat types uniformly as objects. In this case the specification of inheritance between types and the inheritance between instances of those types (which are different due to the existence of variant objects) and the link between the two is bound to be quite

complex. These complications affect query processing directly since the optimization of queries in an object-oriented system can and should take advantage of the semantics of sharing.

3 Query Model Issues

There are many issues to consider in designing query models. In this paper, we cannot discuss all of these issues. We will concentrate on a few of the more important issues and direct the reader to [?] for a more comprehensive discussion, especially of object algebras.

We concentrate on three key trade-offs of OODB query facilities: (1) formal *vs* ad hoc query languages, (2) predicates based upon structure *vs* behavior, and (3) object-preserving *vs* object-generating operations. We also discuss a number of other design issues.

Formal versus ad hoc query languages. Formal query languages [?, ?, ?] have several properties not found in ad hoc query languages [?, ?] making them more suitable for formal analysis. Most importantly, their semantics are well defined which simplifies formal proofs about their properties. Common types of formal query languages are a calculus or an algebra. A calculus allows queries to be specified declaratively without any concern for processing details. Queries expressed in an algebra are procedural in nature but can be optimized. Algebras provide a sound foundation for rule-based transformation systems [?, ?, ?] which allow experimentation with various optimization strategies. A large body of work exists on algebras for other data models (see, for example, [?, ?]). Defining OODB query requirements formally in terms of an algebra facilitates comparisons with these other models.

An important aspect of formal query languages is whether or not they support a calculus definition (in the sense of the relational calculus). If declarative languages are to be provided at the user interface, there is a need to define a formal *object calculus*. We have defined such a calculus in [?], but calculus definitions are typically lacking in object-oriented query research.

Definition of a calculus raises a number of interesting issues. The notion of *completeness* (in the same sense as relational completeness) has to be worked out, since it influences the set of algebraic operators. Completeness requires the calculus and the algebra to be equivalent. *Safety* of calculus expressions is also an issue that needs to be worked out. Safe expressions guarantee that queries retrieve a finite set of objects in finite amount of time [?]. Finally, efficient algorithms need to be developed to translate safe calculus expressions to algebraic ones.

Predicates based upon structure versus behavior. As discussed in the previous section, some object models implement complex objects whose internal structure is visible while others view objects as instances of abstract data types. Access to objects which are instances of an ADT is through a public interface. This interface defines the behavior of the object. Although the two views of objects appear incompatible, the ADT approach can effectively model complex objects by including *get* and *put* methods for each of the components of the internal structure [?]. Thus, a query language which supports predicates based on object behavior is more general while still allowing knowledge of object representations to be introduced in a later stage of query processing.

Object-preserving versus object creating operations. A distinction can be made between object-preserving and object-creating query operations [?]. Object-preserving query languages [?, ?, ?] return objects which exist in the original database. Object-creating languages [?, ?, ?, ?] answer queries by creating new objects from components of other objects. The new objects have a unique identity and some criteria is used to appropriately establish their supertype/subtype properties. In one sense this violates the integrity afforded by objects with identity as objects with no apparent relation to each other can be combined and presented as a new object which presumes to encapsulate some well defined behavior. But the requirement for combining objects into new relationships does exist; either for output purposes or for further processing as in knowledge bases where knowledge is acquired by forming new relationships among existing facts.

Notice that any OODB query language must have a complete object-preserving query facility independent of whether it additionally creates new objects. The ability to retrieve any object in the database utilizing relationships defined by the inheritance lattice or defined by ADT operations on objects is a fundamental requirement. The addition of object-creating operations adds to the power of the language, but also raises a number of issues such as the type of the created objects and the operations that they support.

Closed versus open algebras. One of the strengths of the relational algebra is that it is *closed* so that the output of one operation can become an input to the next. Extension of this concept to object algebras is considered "highly desirable" [?]. Closure is somewhat more complicated in OODBMSs, however. The simplifying factor in relational systems is that the operand(s) as well as the result of any algebraic operation are relations. Thus, all operators have **one type** of input and generate **one type** of output: relation. In object-oriented systems, the schema consists of many

types. Thus, closure property has to be redefined to handle the multiplicity of types. A closed object algebra consists of operators each of which operates on set(s) of objects of a type in the type system and outputs a set of objects that is of an existing type in the type system. As observed in [?], most object-oriented languages are "able to map structured objects into other structured object. However, the objects returned do not necessarily belong to any of the existing [types]."

Note that the existence of object-creating algebra operators, by definition, complicates closure. The provision of heterogeneous collections as outputs of queries is also difficult to reconcile with closure. The issues relate to the determination of the type of objects in the collection which we address next.

Type of the objects in the query result. It is possible for some object algebra operators to produce results that consist of a set of heterogeneous objects (more precisely, a set of identifiers of objects which belong to different types). If the object algebra is closed, then this heterogeneous set may be an input to another operation. Thus, it is important to be able to determine the set of methods that are applicable to **all** the objects in this heterogeneous set. To put it another way, the type that all these objects belong to have to be found. This is only possible by defining a type system which specifies rules for going up and down the inheritance lattice to determine the type conformance of objects.

A related issue is when to do type checking. Static type checking has the advantage of identifying errors early and without the potentially harmful results which could occur at run time. However, it also hampers dynamic binding of objects, which is a commonly advantage of object-oriented languages.

Finally, almost all applications have the requirement that a program variable be iteratively bound to consecutive elements of a query result. The query and application language type checking mechanisms must be compatible for this binding to be performed in a type safe manner [?].

Object algebra operator set. Few object algebras have been defined formally thus far [?, ?, ?, ?]. There is no agreement on the set of operators or their semantics. As we indicated before, disagreements exist on whether object-creating operators should be included, what the proper level of encapsulation should be and so on. It has been suggested that object algebras should extend relational algebra [?], requiring the definition of project and Cartesian product operators. However, these operators, by definition, deal with components of objects, thereby violating strict

encapsulation. As we indicated above, there is probably a need to include these operators in the language, but their exact relationship to encapsulation needs to be worked out.

4 An Example Object Data Model and Query Model

This section presents the object data model and the query model that we have used in our investigation of query processing issues in object-oriented databases. The model indicates how some of the tradeoffs presented in the previous two sections can be resolved. The description in this paper is relatively intuitive. For a rigorous and formal definition of these concepts, the reader is referred to [?].

4.1 Objects

Objects are viewed as instances of abstract data types (ADT) which can only be manipulated via functions defined by the type. Types are organized in an inheritance hierarchy which allows multiple inheritance. Each object has a unique, time invariant identity which is independent of its state. Relations on object identities such as equality and set inclusion provide the basis for query primitives which qualify algebra operators. All other relations among objects are implemented by the ADT interfaces.

4.2 Classes and Methods

In accordance with popular object-oriented terminology, a *class* defines both an ADT interface via *methods* and stands for all the objects which are instances of the type. Methods are named functions whose arguments and result are objects. Each method has a signature of the form $C_1 \times \ldots \times C_n \rightarrow C_{result}$ where $C_1 \ldots C_n$ specify the class of the argument objects and C_{result} specifies the class of the result object. All classes in the database form a lattice where the root node represents the most general class of objects and any individual class may have multiple parents. Subclasses inherit behavior from their parents and may define additional methods. Thus, the class lattice provides *inclusion polymorphism* [?] which allows an object of class C to be used in any context specifying a superclass of C [?].

4.3 Primitive Object Operations

Objects encapsulate a state and a behavior. Methods defined on the class which an object is an instance of define the object's behavior. Behavior is revealed by applying a method to an object. The result of a method application is another object. The dot notation $<o_1 \ldots o_n>.m_1.m_2 \cdots m_m$ is used to denote method application and method composition. Assuming methods m_1 and m_m take three arguments each, and method m_2 takes 2 arguments, then Figure **??** illustrates the processing denoted by this operation. Method m_1 is applied to objects $<o_1, o_2, o_3>$ resulting in object r_1, method m_2 is applied to objects $<r_1, o_4>$ returning object r_2, and so on until the final result object r_m is obtained by applying method m_m to objects $<r_{m-1}, o_{n-1}, o_n>$. $<o_1 \ldots o_n>.mlist$ will be used when the list of method names is unimportant.

Figure 2: Composition of method applications.

An object's state is captured by its value which is distinct from its identity [**?**, **?**]. Object values are either an *atomic value* provided by the database system (int, string, uninterpreted byte sequence [**?**]), a *set value* which is a collection of object identifiers, or a *structural value*. Structural values are visible only to class implementors and can encompass attributes (tuples), discriminated unions, etc. as in [**?**]. Any aspects of structural values which are required by users of a class should be revealed by the implementor via a method.

We define four comparison operators which can be used in queries: $==$, $\in$, $=_{\{\}}$ and $=$ whose semantics are shown in Tables **??** and **??**. The $==$ operator tests for object identity equality; i.e., $o_i == o_j$ evaluates to true when o_i and o_j denote the same object. The $\in$ and $=_{\{\}}$ operators apply

to set valued objects and denote set value inclusion and set value equality respectively. As shown in the tables, one of the operands can denote a value if required. The last operator, =, can only be used to test the value of an atomic object.

Table 1: Semantics of $o_i \theta o_j$ as a function of the object value type.

Table 2: Semantics of $a \theta o_i$ as a function of the object value type.

4.4 Predicate Formation

Atoms are primitive operations of the object data model which return a boolean result. Atoms reference lower case, single letter object variables which range over sets of objects when used in a query. The legal atoms are as follows:

- $o_i \theta o_j$ where:
  - o_i and o_j are object variables or denote an operation of the form $<o_1 \ldots o_n>.mlist$ where $o_1 \ldots o_n$ are object variables.
  - θ is one of the operators $==$, $\in$ or $=_{\{\}}$.
- $a \theta o_i$ where:
  - o_i is an object variable or denotes an operation of the form $<o_1 \ldots o_n>.mlist$ where $o_1 \ldots o_n$ are object variables.
  - a is the textual representation of an atomic value or a set of atomic values.
  - θ is one of the operators $=$, $\in$ or $=_{\{\}}$.

Predicates are formed by connecting atoms with $\wedge$, $\vee$ and $\neg$ as required.

Example 4.1 Let p, q and r be object variables. Then the following are examples of legal atoms and their semantics:

1. $(p == q)$ – Are the objects denoted by p and q the same object?
2. $(p \in <q, r>.mlist)$ – Is the identifier of p contained in the set value of the object obtained by applying the methods in *mlist* to the objects $<q, r>$?

3. ($<p,q>.mlist =_{\{\}} r$) – Is the set value of the object obtained by applying the methods in *mlist* to the objects $<p,q>$ pairwise equal to the set value of the object denoted by r?

4. ("59" $= p$) – Is the atomic value of the object denoted by p "59"?

5. ("59" $\in p$) – Does the set value of the object denoted by p include an identifier for the object whose atomic value is "59"?

6. ("59, 61" $=_{\{\}} <p,q,r>.mlist$) – Does the set value of the object obtained by applying the methods in *mlist* to the objects $<p,q,r>$ contain only two identifiers for objects whose atomic values are "59" and "61"? $\diamond$

4.5 Query Language – An Object Algebra

The object algebra contains both binary and n-ary operators. Let Θ be an operator in the algebra. We will use the notation $P\ \Theta\ \langle Q_1 \ldots Q_k \rangle$ for algebra expressions where P and Q_i denote sets of objects. In the case of a binary operator we will use $P\ \Theta\ Q$ without loss of generality. The algebra defines five object preserving operators: union, difference, select, generate and map. These are fundamental operators; other may be defined (e.g., intersection) for convenience in terms of these.

Union (denoted $P \cup Q$): The union is the set of objects which are in P or Q or both. An equivalent expression for union is $\{\ o \mid P(o) \vee Q(o)\ \}$.

Difference (denoted $P - Q$): The difference is the set of objects which are in P and not in Q. An equivalent expression for difference is $\{\ o \mid P(o) \wedge \neg Q(o)\ \}$. The intersection operator, $P \cap Q$, can be derived by $P - (P - Q)$.

Select (denoted $P\ \sigma_F\ \langle Q_1 \ldots Q_k \rangle$): Select returns the objects denoted by p in each vector $< p, q_1 \ldots q_k > \in P \times Q_1 \times \ldots \times Q_k$ which satisfies the predicate F. An equivalent expression for select is $\{\ p \mid P(p) \wedge Q_1(q_1) \wedge \ldots \wedge Q_k(q_k) \wedge F(p, q_1 \ldots q_k)\ \}$.

The select is similar to, but more powerful than, that of [**?**] which allows only one operand. Multiple operands permit explicit joins as described in [**?**]. An explicit join is a join between arbitrary classes which support (a sequence of) method applications resulting in comparable objects.

Generate (denoted $Q_1\ \gamma_F^t\ \langle Q_2 \ldots Q_k \rangle$): F is a predicate with the condition that it must contain one or more generating atoms for the target variable t, i.e., t does not range over any of

the argument sets. The operation returns the objects denoted by t in F for each vector $<q_1 \ldots q_k> \in Q_1 \times \ldots \times Q_k$ which satisfies the predicate F. An equivalent expression for generate is $\{\ t \mid Q_1(q_1) \wedge \ldots \wedge Q_k(Q_k) \wedge F(t, q_1 \ldots q_k))\ \}$.

Generating atoms are unique in that they generate values for variables which do not range over an input set of the query (Table **??**). They are called generating atoms because they generate objects for x from a constant value (entry 5), from the content of other objects (entries 2,4), or by applying methods to objects (entries 3,4). As an illustration, consider the query $Q\ \gamma^{p}_{(p \in <q,r>.mlist)}\ \langle R \rangle$. Variables q and r range over the argument sets Q and R respectively and thus can be considered 'bound' in the query. However, variable p is not bound to any argument set and the atom $p \in <q, r>.mlist$ will evaluate to true only when p ranges over the objects in the set value of the objects obtained by the method applications. Under these conditions then, the atom generates values for p.

Table 3: Generating atoms for x.

Map (denoted $Q_1 \mapsto_{mlist} \langle Q_2 \ldots Q_k \rangle$): Let *mlist* be a list of method names of the form $m_1 \cdots m_m$. Map applies the sequence of methods in *mlist* to each object $q_1 \in Q_1$ using objects in $\langle Q_2 \ldots Q_k \rangle$ as parameters to the methods in *mlist*. This returns the set of objects resulting from each sequence application. If no method in *mlist* requires any parameters, then $\langle Q_2 \ldots Q_k \rangle$ is the empty sequence $\langle\ \rangle$. Map is a special case of the generate operator whose equivalent is $\{\ t \mid Q_1(q_1) \wedge \ldots \wedge Q_k(q_k) \wedge t == <q_1 \ldots q_k>.mlist\ \}$. This form of the generate operation warrants its own definition as it occurs frequently and supports several useful optimizations.

5 Conclusions

We discussed the object data model and query model design issues in object-oriented database systems. Specifically, we highlighted the alternatives and indicated the tradeoffs. We also presented parts of our work [**?**] to demonstrate a consistent set of design decisions.

It may be too early for an exhaustive discussion of object data model and query model design issues. Work on query models in object-oriented database systems is fairly recent and all of the relevant issues have not yet been uncovered. Nevertheless, the enumeration of the issues and the identification of the tradeoffs is important, especially for standardization efforts. This paper represents and initial attempt along these lines reflecting our experience in designing object-oriented

query models and their processors.

Acknowledgments

This research has been supported in part by the National Sciences and Engineering Research Council (NSERC) of Canada under operating grant OGP-0951.