An Equational Object-Oriented Data Model and its Data-Parallel Query Language

Susumu Nishimura[*]
nisimura@kurims.kyoto-u.ac.jp
Research Institute for Mathematical Sciences
Kyoto University

Atsushi Ohori[†]
ohori@kurims.kyoto-u.ac.jp
Research Institute for Mathematical Sciences
Kyoto University

Keishi Tajima
tajima@in4wolf.in.kobe-u.ac.jp
Department of Computer and Systems Engineering
Kobe University

Abstract

This paper presents an equational formulation of an object-oriented data model. In this model, a database is represented as a *system of equations* over a set of oid's, and a database query is a transformation of a system of equations into another system of equations. During the query processing, our model maintains an *equivalence relation* over oid's that relates oid's corresponding to the same "real-world entity." By this mechanism, the model achieves a declarative set-based query language and views for objects with identity. Moreover, the query primitives are designed so that queries including object traversal can be evaluated in a data-parallel fashion.

1 Introduction

A major advantage of object-oriented databases over traditional relational databases is that they directly support complex objects with complicated object sharing relations through the mechanism of *object identity*. Unfortunately, however, this mechanism makes it difficult to develop a set-based declarative query language and a view mechanism. It also conflicts with parallel processing of database queries. The purpose of this work is to develop a simple yet powerful formulation of an object-oriented data model that overcomes these two difficulties and define a data-parallel declarative query language based on the formalism.

Let us first examine the problem of developing a declarative query language for objects with identity. In the relational data model, the unit of data manipulation is a relation, i.e. a set of tuples, and a database query is simply a relation transformation constructed from a small set of primitive operations on relations. This property provides a clean declarative semantics to database queries and is the basis of query decompositions and optimization. This structure also supports database views: since the result of a query is another relation, a view can simply be a query expression which is to be evaluated lazily when needed. This is in sharp contrast with object-oriented data models. An object-oriented database consists of sets of mutually dependent objects interconnected by object identifiers, and queries crucially depend on navigation, i.e. traversing object-identifiers. This structure does not immediately yield a set-based declarative query language. To see the problem, consider the following simple object types and sets of objects:

type *Empl*=[Name:*string*, Salary:*int*,
    Department:*Dept*, Boss:*Empl*]
type *Dept* = [Name:*string*, Manager:*Empl*]
Empl : $\{Empl\}$
Dept : $\{Dept\}$

where $\{T\}$ denotes the set type whose element type is T and $[\ell_1{:}\tau_1,\ldots,\ell_n{:}\tau_n]$ denotes a type of object containing the set of fields $\ell_1{:}\tau_1,\ldots,\ell_n{:}\tau_n$. One typical

[*]Partly supported by International Information Science Foundation, Tokyo Japan.

[†]Partly supported by the Japanese Ministry of Education Grant-in-Aid for Scientific Research on Priority Area: "Advanced databases," area no. 275.

query against this database would be to transform the above two sets of objects into the following different representation.

type *Empl'* = [Name:*string*, Department:*Dept'*,
Boss:*Empl'*, Colleague:$\{$*Empl'*$\}$]
type *Dept'* = [Name:*string*, Manager:*Empl'*]
Empl' : $\{$*Empl'*$\}$
Dept' : $\{$*Dept'*$\}$

In order for this transformation to be a database query on objects, the results should at least be sets of objects preserving the mutual dependency of the original sets of objects. Furthermore, we also expect that an object in Empl' (Dept') and corresponding object in Empl (Dept, respectively) denoting the same "real-world entity" share the same identity, so that we should be able to perform various other queries on objects through those sets such as identity test for two objects taken from each of the two sets Empl and Empl', or computing the intersection of two subsets of the sets.

This sort of transformation is routinely done in the relational model, and is the basis for view definitions. Unfortunately, however, none of existing proposals of object-oriented database query languages allow us to perform such a query processing. The apparent difficulty is the preservation of object identity while performing transformation of structures of objects. In most of object-oriented data models, the identity of an object is determined based on an oid attached to a record representing the object, and therefore a record in Empl and one in Empl' cannot share the same identity. A simple approach to get around this difficulty is to restrict queries to be those that can be constructed by navigation and selection [BKK88], disallowing any query that changes the structures of objects such as the example above, and eliminates the possibility of flexible view definitions. In [Bee95], Beeri discussed several desireble features in a formal model of an object-oriented databases. The query languge sketched in his proposal is, however, based on the approach similar to [BKK88], and does not contain proper query mechanism for identity preserving transformation of object structures. O_2 [LRV88, BCD89] allows queries that construct new structures by distinguishing "values" from objects and insisting that the results of such queries should be values. The drawback of this approach is that a structure transforming query does not preserve properties of objects including sharing relations and object identity. In some languages, so called "object-creating queries" are allowed [AK89, Day89, SZ89, Kim89, Cat94]; the result of an object-creating query is new objects independent of any existing ones. Of course, creation of new objects is sometimes necessary, but this mechanism can not be a substitute for the ability to transform the structures of existing objects while preserving their identity.

In order to obtain a sufficiently expressive query language for objects with identity, we need to develop a general mechanism for writing identity preserving queries. There are some attempts [SS90, AB91, Run92] to support identity preserving queries and views. However, their approach is just to support primitives for dropping or adding methods to the classes to which objects belong, and there seems to be no general mechanism to write identity preserving object transformation such as the example above. We believe that the source of the difficulty is the lack of a proper formalism for manipulating complex objects with identity. As is done in the relational model, one way to solve the problem is to construct a proper semantic domain together with a set of primitive operations to form a sufficiently rich algebra to represent databases and queries.

Let us turn to the second issue of parallel processing of object-oriented database queries. A database consists of a large collection of data elements. One of the most promising paradigms for parallel processing on such a large collection of data is *data-parallelism*, which is based on the mechanism of applying the same operation to each data item simultaneously. This paradigm works well for relational databases in which each relation is represented as a collection of independent tuples, and several parallel databases [DG92, YCWT93] have been developed based on this paradigm. However, object-oriented databases seem to be difficult to be integrated with the paradigm of data-parallelism because of the existence of navigation. For example, suppose a database of the ancestry trees of viruses is defined as a set of objects of the following type.

type *Virus* = [Code:*string*, Ancestor:*Virus*]

The Ancestor field contains an oid for the immediate ancestor. Now consider the problem of transforming the set into the following form

type *Virus* = [Code:*string*, Origin:*Virus*] by finding the origin of each virus. To perform such a transformation, all the ancestors for each virus object must be traversed, until the origin is reached. The current practice in object-oriented database programming is to combine a database query language with a programming language with recursion and to resort to general recursive programming. This rather crude approach loses the benefit of set-based declarative query processing and yields complicated programs. More seriously, recursively traversing oid's (in Ancestor field in this example) sequentializes the oid dereference operations, and requires the time proportional to the maximum height of the ancestry trees. However, as we shall show later, there is a data-parallel algorithm that performs such traversals in logarithmic parallel steps. To achieve an object-oriented query language supporting data-parallel queries, we must develop a mechanism for parallel object transformation including navigation.

The goal of this paper is to develop an object-oriented data model that achieves identity-preserving object transformation and data-parallel query with navigation. There are other important features in object-oriented databases such as *inheritance*. Although the present paper does not consider such features, we believe that the basic framework for representing objects with indetity will be extended with those other features. We will briefly comment on this issue in Section 7.

The basis of our development is to regard an object-oriented database as a system of equations over object identifiers. This general idea is not entirely new. In IQL [AK89], a database is represented by a set of oid's associated with a function that maps oid's to the associated values. A similar structure is used in ILOG [HY90]. The structures used in these models can be seen as representations of system of equations. Indeed, representing an object-oriented database as a system of equations is just a way to represent a set of objects interconnected by oid's, and this alone does not solve the problems mentioned above. Our proposal is based on the following two key observations:

1. A database query is an operation to compute a new system of equations from a given system of equations.

2. An object identity corresponds not to a single oid in a system of equations but to an equivalence class of oid's induced by queries. The oid's belonging to an equivalent class all correspond to the same "real-world entity."

This first feature enables us to recover the desirable closure property of database queries: the result of a query remains the same semantic domain of databases. By combining some of the authors' recent results [NO95], we can also overcome the sequentiality of traversing object identifiers and develop a data-parallel evaluation scheme for database queries. A database can now be regarded as a collection of equations, each equation can be transformed in parallel.

The second feature enables us to develop an identity preserving query language. In most of the existing object-oriented data models, object identity is regarded as an oid itself. We observe that this is the source of the difficulties in developing a query algebra for objects mentioned above. In contrast to those existing ones, object identity in our model is up to an equivalence relation between oid's induced by queries: two objects have the same identity if the oid's of the two objects belong to the same equivalence class. When a query generates a new system of equations over a new set of oid's from a given system of equations, the equivalence relation is extended to relate each new oid in the new system with the corresponding oid in the source system. The general idea of maintaining equality among multiple objects corresponding to one "real-world entity" was also exploited in several proposals in the context of versions of objects [CK86, KSW86], and views or roles of objects [HZ90, ABGO93]. However, in these proposals, the maintenance mechanism of equivalence relation between oid's is designed for specific purposes, and is not an integrated part of a query language. In [KC86], various forms of equality relations among objects are discussed, but none of them can be applied to express the correspondence between different views of the same entity discussed above. To our knowledge, there is no query language that systematically maintains object identity during general query processing.

Based on these two features, we develop an object-oriented data model and its data-parallel query language, which allows us to write data-parallel queries to transform complex (possibly mutually dependent) structures to another structures while maintaining identity of objects. For example, the object set Empl mentioned at the beginning of this paper is represented by a system of equations over a set of oid's. Transformation of this set into a new set Empl' of objects is done by generating a new system of equations from the system of equation over the oid's in Empl by a data-parallel transformation. In this transformation, the equivalence relation is generated to relate each oid in Empl and the corresponding oid in Empl'.

The rest of the paper is organized as follows. In Section 2, we explain the semantic domain of our object-oriented data model. Section 3 develops a language to define and manipulate object-oriented databases. Section 4 demonstrates its usefulness by examples. In particular, we show that our model support identity preserving queries and object-oriented views. Section 5 shows that the query language supports data-parallel query processing involving navigation. In Section 6, we briefly describe both sequential and parallel implementation strategies of the language. Finally, Section 7 concludes this paper.

2 Semantic Structure of the Data Model

This section describes a semantic structure underlying the object-oriented data model proposed in this paper. The operations for writing database queries will be given in the next section by defining a language to create and manipulate the structure described in this section.

The top level semantic structure is a *database*, which is a triple $(\mathcal{S}, \mathcal{O}, \mathcal{C})$ where $\mathcal{S}$ is a *schema* describing the type structure of the database, $\mathcal{O}$ is a *schema instance* representing object structures, and $\mathcal{C}$ is an *oid classification* into classes. In what follows, we explain each of these components.

To define the structure of a schema, we need to define the set of types in the model. Here, we consider the set of types given by the following grammar:

$$\tau ::= t \mid b \mid [\ell : \tau, \ldots, \ell : \tau] \mid \{t\} \mid class(t)$$

t stands for a denumerable set of type variables, which are used for types of oid's. Intuitively, a type variable corresponds to a class name in usual object-oriented systems. b stands for a given set of atomic types for various atomic values such as integers. $[\ell_1 : \tau_1, \ldots, \ell_n : \tau_n]$ is a record type we have already explained. $\{t\}$ is a set type whose element type is type variable t. This implies that sets are restricted to be those of oid's. A set of values of atomic or structured type, however, can be expressed as a set of oid's by using the mechanism of schema and schema instance which will be explained below. $class(t)$ is a *class* type whose element is oid's of type t. A set of oid's of type $\{t\}$ is called a class and has class type $class(t)$, if it satisfies a particular condition which will be given later when we define typing relation. Note that $class(t)$ is a subtype of $\{t\}$ and therefore that a set of type $class(t)$ also has type $\{t\}$.

A schema $\mathcal{S}$ is then defined as a system of type equations of the form:

$$\{t_1 = \tau_1, \ldots, t_n = \tau_n\}$$

with an associated equivalence relation $\cong_{\mathcal{S}}$ on the set of type variables $\{t_1, \ldots, t_n\}$. We require that the set of type equations is *closed*, i.e. $t_1, \ldots, t_n$ are pairwise distinct and all the type variables occurring in $\tau_1, \ldots, \tau_n$ belong to the set $\{t_1, \ldots, t_n\}$. Each equation $t_i = \tau_i$ describes the fact that an oid of type t_i has the associated value of type τ_i, and determines the type structure of $class(t_i)$. Since $t_1, \ldots, t_n$ may appear in any of $\tau_1, \ldots, \tau_n$, the entire schema definition represents the structure of mutually dependent classes of objects. The associated equivalence relation $\cong_{\mathcal{S}}$ denotes the property that if $t_1 \cong_{\mathcal{S}} t_2$ then t_1 and t_2 represent two different classes that correspond to the same set of "real-world entities". How this relation is maintained will be given later when we define query primitives.

To define schema instances, we first define the set of possible object values (ranged over by O) by the following grammar:

$$O ::= o \mid a \mid [\ell = O, \ldots, \ell = O] \mid \{o_1, \ldots, o_n\}$$

where o stands for a denumerable set of oid's and a for a given set of atomic constants. As mentioned above, we restrict sets to be those of oid's. A set of values of atomic or structured type, such as a set of integers, are expressed as a set of oid's $\{o_1, o_2, \ldots, o_n\}$ with the following set of equations over oid's:

$$\{o_1 = 3, o_2 = 5, \ldots, o_n = 38\}$$

The type of this set is expressed as $\{t\}$ with a type equation $t = int$ in $\mathcal{S}$. Representing sets of values in such an indirect way, though it seems to be somewhat inefficient, enables a uniform treatment of ordinary sets of values and sets of objects.

A schema instance $\mathcal{O}$ is defined as a system of closed oid equations of the form:

$$\{o_1 = O_1, \ldots, o_n = O_n\}$$

- Schema

$$\mathcal{S} = \left\{ \begin{array}{lcl} Empl & = & \text{[Name:}\textit{string}\text{, Salary:}\textit{int}\text{, Department:}\textit{Dept}\text{, Boss:}\textit{Empl}\text{],} \\ Dept & = & \text{[Name:}\textit{string}\text{, Manager:}\textit{Empl}\text{]} \\ Empl' & = & \text{[Name:}\textit{string}\text{, Department:}\textit{Dept}'\text{, Boss:}\textit{Empl}'\text{, Colleague:}\{\textit{Empl}'\}\text{],} \\ Dept' & = & \text{[Name:}\textit{string}\text{, Manager:}\textit{Empl}'\text{]} \end{array} \right\}$$

Type equivalence: $Empl \cong_{\mathcal{S}} Empl'$, $Dept \cong_{\mathcal{S}} Dept'$

- Schema instance

$$\mathcal{O} = \left\{ \begin{array}{lcl} mary & = & \text{[Name="Mary", Salary=6850, Department=}\textit{director}\text{, Boss=}\textit{mary}\text{],} \\ john & = & \text{[Name="John", Salary=3770, Department=}\textit{account}\text{, Boss=}\textit{mary}\text{],} \\ judy & = & \text{[Name="Judy", Salary=3120, Department=}\textit{account}\text{, Boss=}\textit{john}\text{],} \\ director & = & \text{[Name="Director", Manager=}\textit{mary}\text{],} \\ account & = & \text{[Name="Account", Manager=}\textit{john}\text{],} \\ & & \\ mary' & = & \text{[Name="Mary", Department=}\textit{director}'\text{, Boss=}\textit{mary}'\text{, Colleague=}\{\}\text{],} \\ john' & = & \text{[Name="John", Department=}\textit{account}'\text{, Boss=}\textit{mary}'\text{, Colleague=}\{judy'\}\text{],} \\ judy' & = & \text{[Name="Judy", Department=}\textit{account}'\text{, Boss=}\textit{john}'\text{, Colleague=}\{john'\}\text{],} \\ director' & = & \text{[Name="Director", Manager=}\textit{mary}'\text{],} \\ account' & = & \text{[Name="Account", Manager=}\textit{john}'\text{],} \end{array} \right\}$$

Oid equivalence: $mary \cong_{\mathcal{O}} mary', john \cong_{\mathcal{O}} john', judy \cong_{\mathcal{O}} judy',$
$director \cong_{\mathcal{O}} director', account \cong_{\mathcal{O}} account',$

- Oid classification:

$$\begin{array}{ll} \mathcal{C}(Empl) = \{mary, john, judy\}, & \mathcal{C}(Dept) = \{director, account\}, \\ \mathcal{C}(Empl') = \{mary', john', judy'\}, & \mathcal{C}(Dept') = \{director', account'\} \end{array}$$

Figure 1: A Well-Typed Database

associated with an equivalence relation $\cong_{\mathcal{O}}$ on the set of oid's $\{o_1, \ldots . o_n\}$. A schema instance $\mathcal{O}$ represents a set of mutually dependent objects. The intended meaning of the equivalence relation $\cong_{\mathcal{O}}$ is that if $o_1 \cong_{\mathcal{O}} o_2$ then o_1 and o_2 are the different views of the same "real-world entity."

An oid classification $\mathcal{C}$ is a finite map from type variables to sets of oid's. For each type variable $t \in domain(\mathcal{C})$, $\mathcal{C}(t)$ indicates the set of all the oid's of type t.

A database, represented by a triple $(\mathcal{S}, \mathcal{O}, \mathcal{C})$ consisting of the above structures, must be type consistent. To define the type consistency, we first define typing relation $O : \tau$ of object values by the following set of rules:

- $a : b$ whenever a is an atomic value of an atomic type b.
- $o : t$ whenever $o \in \mathcal{C}(t)$.
- $[\ell_1 = O_1, \ldots, \ell_n = O_n] : [\ell_1 : \tau_1, \ldots, \ell_n : \tau_n]$ if $O_1 : \tau_1, \ldots, O_n : \tau_n$.
- $\{o_1, \ldots, o_n\} : \{t\}$ if $\{o_1, \ldots, o_n\} \subseteq \mathcal{C}(t)$.
- $\{o_1, \ldots, o_n\}$: *class*(t) if $\{o_1, \ldots, o_n\} = \mathcal{C}(t)$ and $o \cong_{\mathcal{O}} o_i$ for some o_i for all $o \in \mathcal{C}(s)$ s.t. $s \cong_{\mathcal{S}} t$.

The first four express natural typing condition for each structure. The last rule describes the condition for a set of oid's of type t to be a class of type $class(t)$: the set must contain all the oid's of type t and every oid in the set must has an equivalent oid of type s for all s such that $s \cong_{\mathcal{S}} t$.

The well typing condition of a database $(\mathcal{S}, \mathcal{O}, \mathcal{C})$ is now given by the following conditions:

- $t \in domain(\mathcal{C})$ iff $t = \tau \in \mathcal{S}$ for some τ.
- If $o = O \in \mathcal{O}$ then there exists $t = \tau \in \mathcal{S}$ such that $o \in \mathcal{C}(t)$ and $O : \tau$.
- If $\mathcal{C}(t) \cap \mathcal{C}(s) \neq \emptyset$ then $t \equiv s$.
- If $o_1, o_2 \in \mathcal{C}(t)$ and $o_1 \cong_{\mathcal{O}} o_2$ then $o_1 \equiv o_2$.
- If $o_1 \cong_{\mathcal{O}} o_2$ and $o_1 \in \mathcal{C}(t)$ and $o_2 \in \mathcal{C}(s)$, then $t \cong_{\mathcal{S}} s$.

Databases of the form $(\mathcal{S}, \mathcal{O}, \mathcal{C})$ satisfying the well typing conditions form the semantic domain of our data model. As we noted in the beginning of this section, the semantic structures presented in this section only describes a "snapshot" of the state of a database. Our major contribution is to show that a declarative query language can be constructed for creating and manipulating these semantic structures.

We show in Figure 1 an example of a well-typed database state. This is a database of employees and departments we considered in Section 1. The database consits of four classes: $\{mary, john, judy\}$ of type $class(Empl)$, $\{director, account\}$ of type $class(Dept)$, $\{mary', john', judy'\}$ of type $class(Empl')$, and $\{director', account'\}$ of type $class(Dept')$. The latter two are the classes that would be derived from the former ones by an identity preserving query, and the query induces the equivalence relations over type variables and oid's.

3 The Query Language

We are now in position to define a query language for object-oriented databases whose models are described in the previous section. In this section, we describe the syntax of each construct, its typing constraint, and its semantics as the effect on a given database state.

We use the following operations on records and oid's: [ℓ_1=v_1,...,ℓ_n=v_n] defines a record with v_i for the value of each field ℓ_i, as seen in the examples. X.ℓ selects the value contained in the filed ℓ of X. If o denotes an oid, then value(o) returns the value associated with the oid. As a convention, we usually omit value if it is combined with a field selection operation. That is, o.Field is an abbreviation for value(o).Field. We also assume the availability of standard primitive operations on atomic types such as integer arithmetic operations.

The constructs of the language falls into five categories: class creation, set operations, identity preserving transformation, oid operations, and reduction operation.

3.1 Class Creation

The query language provides two ways to define classes of objects. The first is to define a set of mutually dependent classes by enumerating all the objects in each class using the following construct:

$$
\begin{array}{l}
\textsf{class} \\
\quad \textsf{Name}_1\text{=}\{\textsf{o}_{1,1}\text{=}\textsf{O}_{1,1},\ldots,\textsf{o}_{1,n(1)}\text{=}\textsf{O}_{1,n(1)}\} \\
\textsf{and Name}_2\text{=}\{\textsf{o}_{2,1}\text{=}\textsf{O}_{2,1},\ldots,\textsf{o}_{2,n(2)}\text{=}\textsf{O}_{2,n(2)}\} \\
\quad\vdots \qquad\qquad\qquad \vdots \\
\textsf{and Name}_m\text{=}\{\textsf{o}_{m,1}\text{=}\textsf{O}_{m,1},\ldots,\textsf{o}_{m,n(m)}\text{=}\textsf{O}_{m,n(m)}\} \\
\textsf{end}
\end{array}
$$

where each $\textsf{Name}_i$ is a program variable in the language bound to a class of objects, $\textsf{O}_{i,j}$'s are object values, and $\textsf{o}_{i,j}$'s are oid descriptors for the corresponding objects. The oid descriptors can appear in any object value $\textsf{O}_{i,j}$ to refer to the corresponding oid, allowing mutually dependent object definitions. The declaration works as an operation that defines m new classes and registers them in the database. It extends the schema $\mathcal{S}$ with the set of type equations $\{t_1 = \tau_1, \cdots, t_m = \tau_m\}$, provided that the following typing relation holds: $\textsf{O}_{i,j} : \tau_i$ for any i, j under the assumption that $\textsf{o}_{k,l} : t_k$ holds for all k, l, where each t_k is a fresh type variable introduced for each $\textsf{Name}_k$. In accordance with the extension of the schema, the schema instance is extended as follows: First, for every i ($1 \leq i \leq m$), a set of new oid's $\{o_{i,1}, \ldots, o_{i,n(i)}\}$ is created and the schema instance $\mathcal{O}$ is expanded with the set of oid equations $\{o_{i,j} = O_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n(i)\}$ where each $O_{i,j}$ is the value denoted by $\textsf{O}_{i,j}$. Next, each program variable $\textsf{Name}_i$ is bound to the set of oid's $\{o_{i,1}, \ldots, o_{i,n(i)}\}$. Finally, the oid classification $\mathcal{C}$ is augmented so that each t_i is mapped to the set of oid's $\{o_{i,1}, \ldots, o_{i,n(i)}\}$. The type of each $\textsf{Name}_i$ is therefore *class*(t_i).

In Figure 2, we show an example of class declaration which defines a part of the database of employees and departments in the previous section. The example

```
class
    Empl = {mary=[Name="Mary",Salary=6850,Department=director,Boss=mary],
            john=[Name="John",Salary=3770,Department=account,Boss=mary],
            judy=[Name="Judy",Salary=3120,Department=account,Boss=john]}
and Dept = {director=[Name="Director",Manager=mary],
            account=[Name="Account",Manager=john]}
end
```

Figure 2: A Class Creation Example

declares mutually dependent two classes denoted by Empl of type $class(Empl)$ and by Dept of type *class*(*Dept*). In the declaration, the oid descriptors mary, john, ... denote new oid's *mary*, *john*, ..., respectively. This declaration bounds program variable Empl to the set $\{mary, john, judy\}$ and Dept to $\{director, account\}$. It also adds type equations for $Empl$ and $Dept$ to the schema $\mathcal{S}$, and oid equations for *mary*, *john*, *judy*, *director*, *account* to the schema instance $\mathcal{O}$. The oid classification $\mathcal{C}$ is augmented so that *Empl* and *Dept* are mapped to the corresponding sets of oid's.

The other way to create a class is to use the following construct:

class M from x_1 in X_1, ..., x_n in X_n where P

where $x_1, \ldots, x_n$ may appear in M and P. This construct works as follows. Let each X_i has type $\{t_i\}$. If M has type τ under the assumption that $x_1 : t_1, \ldots, x_n : t_n$, then the schema $\mathcal{S}$ is augmented with the equation $t = \tau$ where t is a new type variable introduced for the new class. Then this expression has type *class*(t). The schema instance is augmented as follows. First, the product is generated from the sets of oid's $X_1, \ldots, X_n$. Then, for each product element $(o_1, \ldots, o_n)$, the following operation is simultaneously performed: each x_i is bound to o_i, and under this binding, if the predicate P holds, then M is evaluated to object value O, a fresh oid o for O is created in the new class being defined, and the schema instance $\mathcal{O}$ is augmented with the equation $o = O$. The oid classification $\mathcal{C}$ is augmented so that t is mapped to the set of the new oid's.

3.2 Set Operations

Set expression of the form:

$\{O_1, O_2, \ldots, O_n\}$

defines a set of oid's. If each O_i denotes an oid of type t, then the expression returns the set of the oid's, and the result type is $\{t\}$. Otherwise, if each O_i is a value of atomic or structured type τ, then it returns a set of new oid's $\{o_1, \ldots, o_n\}$, and the schema instance $\mathcal{O}$ is augmented with the set of oid equations $\{o_1 = O_1, \ldots, o_n = O_n\}$ where each O_i is the value evaluated from O_i. The result type is $class(s)$ where s is a fresh type variable. The schema $\mathcal{S}$ and the oid classification $\mathcal{C}$ is augmented with the equation $s = \tau$ and the map from s to the set $\{o_1, \ldots, o_n\}$, respectively.

The basic operations on sets are union and intersection of two sets of oid's written as $X \cup Y$ and $X \cap Y$, respectively, where both X and Y must have the type either $\{t\}$ or *class*(t). The type of the result of intersection is $\{t\}$. The type of the result of union is *class*(t) if either of the two sets has type *class*(t); otherwise the result type is $\{t\}$.

In the spirit of SQL and set comprehension in programming languages, the language also support general set selection whose syntax is:

select M from x_1 in X_1, ..., x_n in X_n where P

where each X_i must be a set of oid's and P is a predicate. This corresponds to the mathematical set comprehension notation $\{M \mid (x_1 \in X_1) \wedge \cdots \wedge (x_n \in X_n) \wedge P\}$. The evaluation proceeds as follows: First, the product of the n sets of oid's are created, and then M (and P) is evaluated simultaneously for each of the products. The resulting set has type $\{s\}$ if the type of X_i is $\{t_i\}$ for each i and M has type s under the type assumption $x_1 : t_1, \ldots, x_n : t_n$. For example,

select x from x in Empl where x.Salary>4000 returns the subset of Empl whose salary is greater than 4000. The type of the resulting set is $\{Empl\}$.

3.3 Identity Preserving Transformation: Parallel Map

Parallel map is the main feature of our language, which allows us to transform a set of classes of objects into another set of classes of possibly different structure preserving the object identity up to equivalence relation of oid's. This operation is derived from the mechanism developed in [NO95] for data-parallelism on recursive data, and is the main source of parallelism of the language. The significance of data-parallel database query will be discussed in Section 5.

The syntax for parallel map is given as follows:

```
map M foreach x in X
```

The intuitive meaning of this construct is to transform the structure of a class X by simultaneously applying an operation (represented by a function λx.M) over the values of every objects in X. The types of this expression is determined as follows. Suppose X has either a set type $\{t\}$ or a class type *class*(t) such that $t = \tau \in \mathcal{S}$. A fresh type variable s is introduced, and the type equivalence relation is augmented by the equivalence $t \cong_{\mathcal{S}} s$. If the type of M is τ' under the type assumption x: τ, the schema $\mathcal{S}$ is augmented with the type equation $s = \tau'$. The resulting set has type $\{s\}$ (*class*(s)) if X has type $\{t\}$ (*class*(t), respectively.) The execution of this operation is as follows: suppose X is a set of oid's $\{o_1, \ldots, o_n\}$ and $o_i = O_i \in \mathcal{O}$ for each i. A set of new oid's $\{o'_1, \ldots, o'_n\}$ is first created, and the equivalence relation is augmented with the equivalences $o_i \cong_{\mathcal{O}} o'_i$ $(1 \le i \le n)$. For each i $(1 \le i \le n)$, x is first bound to the corresponding object value O_i and then the map operation M is evaluated to a value O'_i of type τ'. (In this step, the map operations are executed simultaneously.) Finally, the schema instance $\mathcal{O}$ is expanded with the set of equations $\{o'_1 = O'_1, \ldots, o'_n = O'_n\}$, and the set of oid's $\{o'_1, \ldots, o'_n\}$ is returned as the result. The oid classification $\mathcal{C}$ is augmented so that s is mapped to the set of the new oid's $\{o'_1, \ldots, o'_n\}$.

A simple parallel map example follows:

```
let Empl' = map [Name=x.Name,
                 Department=x.Department]
        foreach x in Empl
```

which creates another restricted "view" of the class Empl. It is also possible to apply several parallel maps to a set of mutually dependent classes and to create another set of mutually dependent ones, as shown in Section 4, with the following concurrent let construct:

```
let X1 = map ...
and X2 = map ...
 ...
and Xn= map ...
end
```

3.4 Oid Operations

There are three oid operations:

value(o)	oid dereference,
M=M'	equivalence checking, and
M as X	oid coercion.

We have already introduced the oid dereference at the beginning of this section.

M=M' checks equivalence between arbitrary two values. If M and M' denote two oid's, say o and o', respectively, then the expression returns true if $o \cong_{\mathcal{O}} o'$; it returns false otherwise. This equivalence checking for oid's is extended to that for arbitrary values in the obvious way.

M as X is a powerful facility to coerce an oid denoted by M to its equivalent oid in the set of oid's denoted by variable X. The type of this expression is therefore the type of the oid's belonging to X. The program variable X can be the one being defined in a concurrent let construct. This facility makes it possible to update a class of objects while preserving the dependencies between the objects. Consider the following example:

```
let Empl' = map [Name=x.Name,
                 Department=x.Department,
                 Boss=(x.Boss as Empl')]
            foreach x in Empl
```

x.Boss as Empl' coerces the oid denoted by x.Boss that belongs to Empl to the equivalent oid which will belong to the new set Empl' being defined. (If there are multiple occurrences of the same name, the inner-most binding of the form Empl'=... enclosing the coercion operation is selected.) The resulting set of oid's has type $\{Empl'\}$ such that $Empl'$ = [Name:*string*, Department:*Dept*, Boss:*Empl'*], and is a new "view" of class Empl. Due to the effect of the coercion operator, the Boss field of every object in the new class is assigned an oid in the new class that shares the same object identity with the equivalent oid in the original class. The coercion facility also can be used to define more powerful operations such as update of database, as will be noted in Section 4.

M as X does not work for arbitrary set X of oid's, even if the oid's in the set is created from the class to which the oid denoted by M belongs. This is because the set X may be created from a subset of the original class generated by a select operation. In such a case, the oid being coerced may not have the corresponding oid in X. This difficulty is circumvented by our distinction of sets by the two types $\{t\}$ and *class*(t), the latter of which guarantees that the set contains an equivalent oid for every oid of type s such that $s \cong_{\mathcal{S}} t$. The typing constraint for the coercion operator is therefore stated as follows: M as X is allowed only if M has type t, X is the name for the value of type *class*(s), and the equivalence relation $s \cong_{\mathcal{S}} t$ holds.

3.5 Reduction Operation

Following approaches for querying database collections [TBN91, BNTW95] and set operations in database programming languages [BBKV88, BO96], we include the following general aggregate operation:

```
reduce X with unit=ε, op=⊗
```

where X has type $\{t\}$ with $t = \tau \in \mathcal{S}$, $\otimes$ is an associative and commutative binary operation on the values of type τ that returns a value of type τ, and ϵ is a unit element for the binary operation. If X is a set $\{o_1, \ldots, o_n\}$ ($n \geq 0$) such that $o_i = v_i \in \mathcal{O}$ (for each i, $1 \leq i \leq n$), the result of the reduction operation is $\epsilon \otimes v_1 \otimes \cdots \otimes v_n$. For example,

```
let Salary = map x.Salary foreach x in Empl
in reduce Salary with unit=0, op=+
```

returns the sum of salary of all the employees. The reduction operator works as the set flattening operation if it is combined with the set union operator:

```
reduce S with unit={}, op=∪
```

where S denotes a set of sets and $\{\}$ denotes the empty set. The reduction operation is also a source of data-parallelism, as will be noted in Section 5.

4 Query Examples and Language Extensions

In this section, we first show some examples of object-oriented database queries and then describe the extensions of the language for views and updates.

4.1 Examples of Queries

Suppose the following type equations are defined in the schema:

Empl = [Name: *string*, Salary:*int*, Department:*Dept*]
Dept = [Name: *string*, Manager: *Empl*]

and Empl and Dept are bound to classes of type $class(Empl)$ and $class(Dept)$ respectively.

The first example is a simple projection on these classes. Suppose the user is only interested in the name of each department and the name of the managers of each department. The query below projects those attributes and bound the result to Dept':

```
let Dept' = map [Name = x.Name,
                 MngrName = x.Manager.Name]
            foreach x in Dept
```

This query is a usual relational-style query, and forgets the sharing relation between the class of employees on Manager attribute. There are cases, however, where we want to reflect the sharing relation between the two classes of employees and departments in the query result. The following is an example of such a query:

```
let Dept' = map [Name = x.Name,
                 Manager = (x.Manager as Empl')]
            foreach x in Dept
and Empl' = map [Name = x.Name]
              foreach x in Empl
end
```

The sharing relation on Manager attribute is retained by coercing the original oid to the corresponding oid in the new class Empl'.

Next, we show an example of a more complex transformation of the class Empl to different structure. The query below computes the colleagues of each employee, i.e. the employees belonging to the same department, and add the result as a new attribute Colleague. The result of the entire query is bound to Empl':

```
let Empl' =
  map [Name = x.Name,
       Department = x.Department,
       Boss = x.Boss as Empl',
       Colleague =
          select (o as Empl) from o in Empl
           where (x.Department=o.Department
                  ∧ x.self≠o)]
  foreach x in Empl
```

In this query, we have assumed that each object value is, when parallel map is applied, automatically augmented with the additional field self that contains the oid of the object itself. In the rest of the paper, we assume this convention.

In a relational database, the query above should be done through joins. In our model, we can extend each employee object directly so that it includes the related objects, while preserving the identity of Empl objects. Note also that the objects in Colleague attribute belong not to the old class Empl but to the newly created class Empl'. This allows us to continue query processing by issuing a new query on Empl' as shown in the following example.

```
select o.Name
from o in Empl'
where (select o2
       from o2 in o.Colleague
       where #(o2.Colleague) > 10) ≠ {}
```

This query returns the names of employees that have a colleague who has more than 10 colleagues. In the query, # stands for a function that returns the number of elements of a given set. # X can be expressed by a combination of a parallel map and a reduction operation as: reduce (map 1 foreach x in X) with unit=0, op=$+$.

These examples demonstrate that our model successfully integrates the benefits of both relational databases and object-oriented databases. In our model, we can easily write a set-based declarative query that transforms classes objects into new classes of different structures that preserve object identity and the mutual dependence relation among objects.

The next example shows an object-creating query using class . . . from . . . where . . . construct. Suppose there is a class of objects of type $class(Computer)$ defined as:

$Computer$ = [Hostname: $string$, Type: $string$, AssignedTo: *Department*]

Assuming that each employee is privileged to use only the computers that are assigned to one's department, the following query computes the class of objects ComputerAccount by joining Empl and Computer belonging to the same department:

```
let ComputerAccount =
  class [Emp = o1, Cmp = o2]
  from o1 in Empl, o2 in Computer
  where o1.Department = o2.Department
```

ComputerAccount has type $class(ComputerAccount)$ with a type equation $ComputerAccount$ = [Emp : $Empl$, Cmp : $Computer$].

The query in Figure 3 transforms the classes Empl and Dept defined above into mutually dependent classes using the coercion operator as. The query adds Manager attribute to the employee objects, adds Member attribute to the department objects, and creates a new set Mngr of managers from Empl as a set of objects having Name, Managing, and Subord attributes. Managing is the set of departments one manages, and Subord is the set of members of the department one manages. Note that the query retains the mutually dependent structure between the objects. This is accomplished by coercing some oid's to those of newly created objects by using as.

There is one subtle point to be made in this query: the objects assigned to Manager field of Emp2 and Dpt2 are coerced to the objects in Emp2. Although it may be more appropriate to coerce them to the objects in Mngr, the typing constraint does not allow it, since the type of Mngr is not $class(t)$ but $\{t\}$. In this particular case, we see that employee objects referred to through Manager attribute always have corresponding objects in Mngr, and therefore coercing them to Mngr never causes errors. Such an analysis could be incorporated in our model by using some techniques in programming language research such as abstract interpretation, but we do not discuss the issue in this paper.

As shown in the example above, parallel map and as construct provide us with a uniform way to transform a class of objects into another class of objects of arbitrary mutually dependent structure. Moreover, transformations induce and maintain the equivalence classes of those objects representing the same "real-world entity." For example, in the query shown in

```
let Emp2 = map [Name = x.Name,
                Department = (x.Department as Dpt2),
                Manager = (x.Department.Manager as Emp2)]
           foreach x in Empl
and Mngr = map [Name = x.Name,
                Managing = (select (o as Dpt2) from o in Dept where o.Manager = x.self),
                Subord = (select (o as Emp2) from o in Empl where o.Department.Manager = x.self)]
           foreach x in (select o.Manager from o in Dept)
and Dpt2 = map [Name = x.Name,
                Manager = (x.Manager as Emp2),
                Member = (select (o as Emp2) from o in Empl where o.Department = x.self)]
           foreach x in Dept
end
```

Figure 3: Transforming Mutually Dependent Classes

Figure 3, objects in Empl, Emp2, and Mngr are properly related by the equivalence relation. This enables us to perform identity test among the objects belonging to different types of classes. For example, the following query is possible.

```
select o1.Name
from o1 in Mngr, o2 in Emp2
where o1 = o2 ∧ o2.Department = "Account"
```

This query returns the names of employees who is a manager of some department and is also a member of the account department. In other words, this computes the intersection of Mngr and the set of all members of the account department.

4.2 Extension for Views and Updates

The facilities for writing a set-based and identity-preserving query immediately yield *identity-preserving view* mechanism; as in relational databases, a view is just an unevaluated query. For example, the query in Figure 3 can be changed to a view definition only by replacing the keyword let with a keyword view. view . . . and . . . end construct is the same as let . . . and . . . end except that the evaluation is delayed until one of the names defined in the construct is referred to. After defining this view, Emp2, Mngr and Dpt2 are used as classes in a query such as:

```
select o.Manager.Subord
from o in Emp2
where o.Name = "John"
```

which is evaluated by first "materializing" all the classes defined in the view, and then evaluating the query itself. In relational databases, only the relations that are referred to in the query need to be materialized. In our model, however, all classes defined in a view construct need to be materialized even if some of them are not yet referred to because they may have mutual references.

We can also describe update of database by means of parallel map and oid coercion. Consider a hypothetical update query in Figure 4(a). The query intends to update the database by adding 100 to the salary of all the employees in the account department. We can translate the hypothetical query to a query written by means of our query primitives as shown in Figure 4(b). This updating query indicates that updating objects is equivalent to mapping the target objects (here, employee objects) to new objects, and switching all references to the target objects into references to the new objects. In the example above, all the objects in Dept need to be mapped because they have references to employee objects. In this way, any update query can be systematically transformed to a query composed of parallel map and oid coercion, by analyzing type dependence. Furthermore, updates that alter the type of objects are also allowed. The straightforward

```
update Empl=[Name=Empl.Name,
             Salary = (if Empl.Department="Account" then Empl.Salary+100 else Empl.Salary),
             Department=Empl.Department]
end
```

(a) A Hypothetical Update Query

```
let Empl = map [Name=x.Name,
                Salary=(if x.Department="Account" then x.Salary+100 else x.Salary),
                Department=(x.Department as Department)]
           foreach x in Empl
and Dept = map [Name = x.Name, Manager = (x.Manager as Empl)]
           foreach x in Dept
```

(b) A Translated Query Code

Figure 4: An Update Query

execution of the translated code may not be as efficient as the corresponding in-place update that may be possible in some imperative language. However, some optimizations are possible. For example, we can discard the old data, since they are replaced by the updated data and are never referred to. In the case the type of objects are not changed, we can translate the update query to the code performing in-place update.

5 Data-Parallel Processing of Database Queries

The proposed paradigm of an object-oriented database query language is suitable not only for identity preserving transformation of classes of objects, but also for achieving data-parallel query processing on parallel machines, especially on recently emerging massively parallel distributed memory multicomputers. The paradigm has two desirable properties. First, it easily scales up with respect to the number of processors. Secondly, we can apply a class of data-parallel algorithms for parallelizing queries including navigations.

5.1 Data-Parallel Query with Navigations

The mechanism of parallel map achieves data-parallel evaluation of a query on a collection of objects in the obvious way, if the map operation does not include any navigations. However, as mentioned in the introduction, navigations sometimes conflict with parallelization. Suppose a database contains the following class describing the ancestry trees of viruses:

```
class
  Virus = {
        iadhk=[Code="IADHK",Ancestor=iadhk],
        iadhl =[Code="IADHL", Ancestor=iadhk],
        iab37=[Code="IAB37", Ancestor=iadhk],
        iackb=[Code="IACKB",Ancestor=iackb],
        iackj =[Code="IACKJ", Ancestor=iackb],
        iackg=[Code="IACKG",Ancestor=iackj],
        iacka=[Code="IACKA",Ancestor=iackg],

        ... and more and more.}
end
```

where the fields Code and Ancestor respectively indicates each virus's identification code and the immediate ancestor; in the case the ancestor is unknown, the Ancestor field is specially set to the oid of the object itself. On this database, consider the following query: detecting the origin of each virus. Processing this query requires traversing virus objects recursively via the oid contained in the field Ancestor. The conventional way of implementing the required traversal as a recursive program, however, does not achieve the desired parallelization, particularly when the class Virus contains a long chain of the ancestor

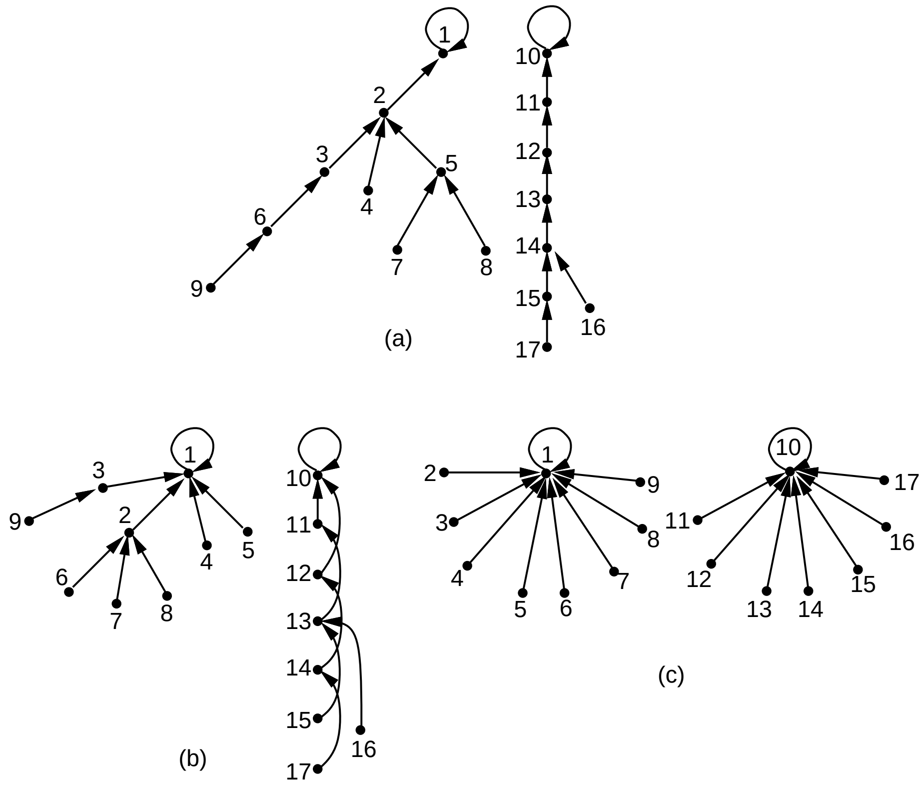

Figure 5: The Data-Parallel Root Finding Algorithm in a Forest of Objects

relation. This is because each recursive computation must wait until the successive recursive computation finishes, i.e. the recursive program sequentializes the oid dereferences. Moreover, if the ancestor relation has many confluences, i.e. there are many objects to which multiple Ancestor pointers go, much redundant duplications of computation are generated.

In the following, we show how this difficulty can be overcome by combining parallel map and the technique of *pointer jumping* [Jaj92] which is well known in the field of imperative data-parallel programming. The above query can be regarded as a problem of finding the root nodes in a forest of trees such as the one illustrated in Figure 5 (a). There is an efficient data-parallel algorithm based on pointer jumping technique for this root finding problem. The algorithm is iteration of the following data-parallel operation: the reference to the parent in each node is simultaneously updated by its reference's reference, resulting new structure as shown in Figure 5 (b). This operation is repeated until all the reference in every node is set to its root node (Figure 5 (c)). Since every reference is doubled in each iteration, every pointer in a forest of trees of N nodes refers to the root node at most after $O(\log_2 N)$ iterations.

This algorithm can be directly applied to the above query by regarding the dots, the integer numbers, and the arrows in the figure as objects, oid's, and reference relations, respectively. The pointer jumping technique can be used to write a wide range of data-parallel algorithms including not only root finding but also parallel prefix, tree contraction, and so on. If we are able to describe such algorithms in our language, we can achieve effective parallelization of various queries including object traversals. In the previous work [NO95], it has been shown that various parallel algorithms based on the pointer jumping technique can be expressed in a declarative programming language through the notion of *parallel recursion*, which is a process of repeatedly transforming a system of equations by parallel map into another system until the final result is obtained.

Our query language is also designed to be able to express parallel recursion on classes of objects. As parallel recursion is iteration of parallel map, it is sufficient to introduce the following primitive for bounded iteration.

```
iterate N times do X=M end
```

In this construct, N is an integer which indicates the number of iterations, and M is the code for each transformation. The transformed class of objects is assigned to the variable X in each iteration. For example, the following code creates a class of new virus objects by replacing the Ancestor field in each object in the class Virus with the Origin field that is assigned the oid of its most distant ancestor:

```
let Virus = map [Code=x.Code,
                 Origin=(x.Ancestor as Virus)]
            foreach x in Virus end;

iterate log(#Virus) times do
   Virus = map [Code=x.Code,
                Origin=(x.Origin.Origin as Virus)]
           foreach x in Virus
end
```

where # Virus is the number of oid's contained in the class and log is the logarithm with base 2. The query first transforms the class Virus to the class of type *class*(*Virus*′) with type equation $Virus' =$ [Code : $string$, Origin : $Virus'$] where each Origin field is set to its immediate ancestor. Then the pointer jumping technique is applied to the class of virus objects by repeatedly transforming the class via parallel map. It is sufficient to iterate $\log_2$ #Virus for every object to reach its most distant ancestor. Though it is possible to stop iteration just when all the objects reach to the most distant ancestors, testing this condition requires, in each iteration, an additional reduction with a boolean operator or over the set of virus objects. We therefore take the way to iterate by a fixed number as a gentle solution for termination checking.

6 Implementation Strategy

We claim that the proposed model serves not only as a formal model to account for object-oriented databases but also as a basis to develop a practical object-oriented database systems. In this section, we describe two implementation strategies of our query language: one for conventional single processor architectures and the other for distributed memory massively parallel multicomputers.

6.1 Implementation on Conventional Single Processor Architectures

Our set-based query primitives such as parallel map is designed to be evaluated in data-parallel fashion. However, they can equally well be implemented on a conventional single processor machines. Most of the features of our query language can be implemented by using the existing technologies for implementing databases on conventional architectures, except for the features that is specific to our query language: oid coercion operation as, equality test between oid's, and oid dereference operation. In order to implement these features, we represent an oid using a pair (i, t) where i is an *instance identifier* and t is the type of that oid. An instance identifier corresponds to a "real-world entity," and is shared by all the objects representing different views of the same entity. For example, in the example in Figure 3 that derives the class Emp2 from the class Emp, we first introduce a fresh type variable s for Emp2, derive objects for Emp2 from objects in Emp, and then assign an oid (i, s) to each object for Emp2 where i is the same instance identifier of the source object from which that object is derived. In this way, the same instance identifier is assigned to all objects representing the same entity.

By this representation of oid's, equality test for two oid's is easily implemented: it just tests the equality of instance identifiers of those oid's. Coercion of an oid is also easy. It just changes the type in the oid to the type of the target class. We can coerce an oid to a class that is being defined, since the type of the target class is statically known by a preceding type checking phase. Finally, oid dereference operation is implemented as a map from oid's to object values. A map for oid dereference is maintained by the system as a hash table whose key is an oid and the contents of whose entry is the object value of that oid. Since only limited combinations of instance identifiers and types would be used, we use hashing function to avoid a very sparse table.

6.2 Implementation on Massively Parallel Multicomputers

To achieve data-parallel execution of our query language, we propose an implementation strategy for distributed memory massively parallel multicomputers. The basic strategy is, similar to that of conventional data-parallel languages such as Dataparallel C [HQ91], to evaluate queries in so called SPMD (single program multiple data-streams) style [Kar87]. In the SPMD execution, every processor executes the same program (compiled code) on its own copy of scalar data, while a set of oid's are distributed over the processors by assigning each data element (oid) to a distinct processor. In the rest of this section, we assume that underlying hardware system is a distributed memory multicomputer consisting of unbounded number of processors, each of which has a unique processor id.

There are four sources of parallelism in the language: the class creation operation class, the set restriction operation select, the parallel map map, and the reduction operation reduce. Among them, the reduction operation can be effectively parallelized due to binary operator's associativity and commutativity. In particular, if the hardware supports special machine instruction for reduction, it can be utilized for speed up.

To achieve parallelization of the other operations, we must invent a run-time representation of schema instance, i.e. system of oid equations, suitable for data-parallelism. In the previous work [NO95], a method has been proposed for expressing system of equations on a massively parallel distributed memory multicomputer model. We can apply this method for implementation of our query language with some modifications as follows. Each oid is represented by (p,t), a pair of a processor id p and the type of the oid t. The processor number p corresponds to instance identifier in the implementation strategy for conventional machines. This indicates objects that represents the same "real-world entity" are assigned to the same processor. A schema instance is represented by maintaining in each processor a table that associates each type variable with the corresponding object value. The object value associated with an oid (p,t) is stored in the local table of processor p with t as the key. Note that an oid (p,t) in a processor p has a unique corresponding object value stored in the local table, since no two objects in the same class are assigned to the same processor. Dereference of an oid (q,t) in a processor p returns the value stored in the local table with t as the key, if $p = q$; otherwise, it retrieves the value stored in the table of processor q with t as the key via inter-processor communication. When a parallel map is applied to transform a set of objects represented by a set of oid's $\{(p_i,t) \mid 1 \leq i \leq n\}$, every processor p_i simultaneously applies the same map operation to the object value associated with t in the local table, and the result value of the map operation is registered to the table with s as the key, where s is the type of oid's of the new object set. The parallel map finally returns a set of oid's $\{(p_i,s) \mid 1 \leq i \leq n\}$ as a result. An oid equivalence checking between two oid's (p,t) and (q,s) returns true if and only if $p = q$ and $t \cong_{\mathcal{S}} s$. An oid coercion on (p,t) to a set of oid's X returns an oid (p,s), where s is the type of oid's belonging to X.

The above method is designed to work only in a simple setting. To support full specification of the query language, we must consider nested parallelism, i.e. execution of parallel operations in another parallel operation. An obvious way to execute such nested parallel queries is to give up nested parallel execution, and to sequentially execute the inner parallel constructs. Such a solution, however, significantly limits the parallelism to be exploited. In order to achieve full parallelization, a technique called *flattening* has been studied by some researchers to support nested data-parallelism in the array based data-parallel languages [Ble90, PP93]. They have proposed systematic ways to flatten the nested parallelism, and an array based data-parallel language NESL [Ble93], which supports the nested parallelism based on such a method, has been actually implemented. A similar technique may be applicable to our language. A bit of difficulty lies in our data representation, the equational data formulation, but the authors believe that the difficulty can be overcome by further investigations.

7 Conclusion and Future Work

This paper proposed an object-oriented data model and its query language. We based our development by modeling an object-oriented database as a system of oid equations of the form $\{o_1 = O_1, \ldots, o_n = O_n\}$ which associates each oid o_i with a value O_i. Every O_i can include any o_j defined within the system of equations for expressing object sharing and mutual dependencies. On this model, we developed a query language that transforms a system of equations to a new system of equations. We demonstrated that the language can express wide range of transformation, including those that cannot be described in the existing object-oriented query languages. In our language, object identity is up to an equivalence relation over oid's in the system of equations. This equivalence relation is automatically induced and maintained for each time the system of equations is transformed by a query. By this mechanism, the language supports identity-preserving queries and views. Another advantage of our model is that it naturally supports data-parallelism. Even those queries that crucially depend on navigation can be evaluated in a data-parallel fashion in our language.

The discussion on our object-oriented data model and query language has been focused on object identity so far. There are some important aspects of object-oriented paradigm which are not covered in our proposal. One of most important is *inheritance*. In object-oriented databases, the term inheritance implies *method inheritance*, i.e. the ability to share code among classes, and *extent inclusion,* i.e., hierarchical organization of classes induced by inclusion relation of their extents. A conventional approach is to represent both of them by simple subtype relation [Car88]. This appracoh can certainly be adopted to our model. A more promising approach would be to integrate the model presented here with a polymorphic type system for database programming language [BO96] based on record polymorphism [Oho95]. As demonstrated in [BO96], method inheritance and extent inclusion can be more accurately represented by polymorphic typing of record structures. We believe that these features can be cleanly integrated with our model of objects.

We have not provided an algebra for our query language either. An important further investigation is to develop an algebra that works as a basis of equational reasoning, query optimizations, and so on. It would be also challenging to develop a systematic method for flattening nested parallel queries, which will enable us to achieve a full implementation of the data-parallel object-oriented query language.

References

[AB91] Serge Abiteboul and Anthony Bonner. Objects and views. In *Proc. ACM SIGMOD Conference*, pages 238–247, Jun. 1991.

[ABGO93] Antonio Albano, R. Bergamini, Giorgio Ghelli, and Renzo Orsini. An object data model with roles. In *Proc. VLDB Conference*, pages 39–51, Aug. 1993.

[AK89] Serge Abiteboul and Paric C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Conference*, pages 159–173, Jun. 1989.

[BBKV88] Francois Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. FAD. a powerful and simple database language. In *Proc. VLDB Conference*, pages 97–105, Sep. 1988.

[BCD89] Francois Bancilhon, Sophie Cluet, and Claude Delobel. A query language for the O_2 object-oriented database system. In *Proc. Int. Workshop on DBPL*, pages 122–138, Jun. 1989.

[BKK88] Jay Banerjee, Won Kim, and Kyung-Chang Kim. Queries in object-oriented databases. In *Proc. IEEE ICDE*, pages 31–38, Feb. 1988.

[Bee95] Catriel Beeri A Formal Approach to Object-Oriented Databases. *Data and Knowledge Engineering*, 5:353–382, 1990.

[Ble90] G.E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Ble93] G.E. Blelloch. NESL: A nested data parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, 1993.

[BNTW95] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, Sep. 1995.

[BO96] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1), 30-76, 1996.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[CK86] Hong-Tai Chou and Won Kim. A unifying framework for versions in a CAD environment. In *Proc. VLDB Conference*, pages 336–344, Aug. 1986.

[Day89] Umeshwar Dayal. Queries and views in an object-oriented data model. In *Proc. Int. Workshop on DBPL*, pages 80–102, Jun. 1989.

[DG92] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6):85–98, Jun. 1992.

[HQ91] Hatcher, P.J. and Quinn, M.J. *Data-Parallel Programming on MIMD Computers*. The MIT Press, 1991.

[HY90] Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proc. VLDB Conference*, pages 455–468, Aug. 1990.

[HZ90] Sandra Heiler and Stanley B. Zdonik. Object views: Extending the vision. In *Proc. IEEE ICDE*, pages 86–93, Feb. 1990.

[Jaj92] J. Jaja. *An Introduction to Parallel Algorithm*. Addison-Wesley, 1992.

[Kar87] A. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57, May 1987.

[KC86] Setrag Khoshafian and George P. Copeland. Object identity. In *Proc. ACM OOPSLA Conference*, pages 406–416, Nov. 1986.

[Kim89] Won Kim. A model of queries for object-oriented databases. In *Proc. VLDB Conference*, pages 423–432, Aug. 1989.

[KSW86] Peter Klahold, Gunter Schlageter, and Wolfgang Wilkes. A general model for version management in databases. In *Proc. VLDB Conference*, pages 319–327, Aug. 1986.

[LRV88] C. Lecluse, P. Richard, and F. Velz. O_2, an object-oriented data model. In *Proc. ACM SIGMOD Conference*, pages 424–433, Jun. 1988.

[NO95] S. Nishimura and A. Ohori. A calculus for exploiting data parallelism on recursively defined data (preliminary report). In *Proc. International Workshop on Theory and Practice on Parallel Programming, LNCS* vol. 907, pages 413–432, 1995.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.

[PP93] J. Prins and D. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 119–128, 1993.

[Run92] Elke A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proc. VLDB Conference*, pages 187–198, Aug. 1992.

[SS90] Marc H. Scholl and Hans-Jorg Schek. A relational object model. In *Proc. ICDT*, *LNCS* vol. 470, pages 89–105. Springer-Verlag, Dec. 1990.

[SZ89] Gail M. Shaw and Stanley B. Zdonik. An object-oriented query algebra. In *Proc. Int. Workshop on DBPL*, pages 103–112, Jun. 1989.

[TBN91] Val Tannen, Peter Buneman, and Shamim Naqvi. Structural recursion as a query language. In *Proc. Int. Workshop on DBPL*, pages 9–19, Aug. 1991.

[YCWT93] Philip S. Yu, Ming-Syan Chen, Joel L. Wolf, and John Turek. Parallel query processing. In Nabil R. Adam and Bharat K. Bhargava, editors, *Advanced Database Systems*, *LNCS* vol. 759, chapter 12, pages 229–258, 1993.