Declarative Definition of Object-Oriented Multidatabase Mappings *

Jan Chomicki
Kansas State University†

Witold Litwin
Université de Paris - Dauphine‡

Abstract

We propose a language for declarative specification of mappings between different object-oriented multidatabases. The language is based on OSQL, the query language of the IRIS OODBMS. We also outline the operational semantics of our language.

*To appear in *Distributed Object Management*, M. T. Ozsu, U. Dayal and P. Valduriez, editors, Morgan Kaufmann Publishers, 1993.

†Work performed while visiting HP Labs, Palo Alto, California. Address: Department of Computing and Information Sciences, 234 Nichols Hall, Kansas State University, Manhattan, KS 66506-2302. Email: `chomicki@cis.ksu.edu`.

‡Work performed while visiting Santa Clara University and HP Labs, Palo Alto, California. Email: `litwin@hplabs.hp.com`.

1 Introduction

Declarative definition of multidatabase mappings is one of the central problems of multidatabase interoperability. For *object-oriented* (OO) databases, one may need to map types (classes) in one database into types in another database, or functions into functions, or objects into objects. In the presence of schematic discrepancies [KLK91], one may further need to map objects (instances) into types or vice versa, or types into functions, etc. Because of the autonomy of different databases, one kind of data elements in one database may need to be mapped into the same or a different kind of data elements in another database.

We propose a language in which OO multidatabase mappings can be defined in a declarative way using a small set of constructs. Those constructs make possible the definition of various kinds of derived data, namely *derived objects*, *derived types*, and *derived functions*. In particular, we introduce the notions of *object*, *type* and *function schemas* that are suitable for dealing with schematic discrepancies, in effect providing "higher-order" features without using a higher-order language. We base our language on the OSQL query language and its dialects [Fo91; Ann91; RDK$^+$91], to which we will collectively refer as OSQL. As a result, we obtain a generalization of OSQL suitable for multidatabase interoperability. Other declarative OO languages, e.g., the query language of O2 [BDK92], could be similarly extended.

A derived type may contain existing objects selected from another type using a derivation expression (a query). Alternatively, it may contain distinct instances of an *object schema* (also called *derived objects*), i.e., objects with distinct OIDs, derived from other data elements using a derivation expression. A single declaration of an *object schema* defines a family of derived objects, one object for each combination of parameter values. Similarly, a type (resp. function) schema defines a family of OSQL derived types (resp. functions).

We address only the issues of interoperability of semantically heterogeneous OO databases with a common data model. How to define mappings between heterogeneous OO models (or from heterogenous OO models into a common one) is an open problem. Mapping relational to OO databases has been addressed elsewhere [RDK$^+$91]. In [KS91], a comprehensive framework for classifying different kinds of heterogeneity arising in the relational data model was presented. Finally, we ignore here the implementation issues related to the physical distribution of the interoperable databases.

Multidatabase mappings can also be expressed in a sufficiently powerful higher-order logic programming language [KLW90; BG92]. However, efficient implementation techniques for such languages do not yet exist. The authors of [KLW90; BG92] suggest that their languages should be translated into Prolog, but Prolog, with the concomitant termination and efficiency problems, is far from being accepted as a database query language. In contrast, as we propose an extension to an *existing* query language, our approach should be more readily implementable.

Multidatabase mappings can also be expressed procedurally [Ken91]. In the presence of schematic discrepancies, however, the procedural approach leads to a much less transparent database schema resulting from the mapping. The procedural approach relies also on the underlying trigger mechanism which may be in conflict with the autonomy of different databases.

In section 2, we survey OSQL and its dialects, and discuss the issues of semantic heterogeneity for OSQL databases. We introduce object and type schemas in section 3, and function schemas in section 4. In section 5, we discuss the placement of derived types in an inheritance hierarchy and the problem of non-monotonic inheritance (overriding). In section 6, we discuss the evaluation of queries. We summarize related work in section 7. Section 8 concludes the paper.

2 OSQL and its dialects

2.1 OSQL

OSQL is a declarative query language in which queries are formulated using a functional notation. An OSQL database has two distinct parts: schema and state. A schema consists of type and function definitions, and subtype hierarchy, where the OSQL notion of *type* corresponds to the notion of *class* in most OO database systems. A state consists of type extensions and function values.

An object (instance of a type) may be *literal* or *surrogate*. A literal object may be *atomic* (integer, real, string,...) or *aggregate* (set, bag, tuple, list). The extensions of user types consist of surrogate objects. To simplify the terminology, we will use the term *value* for *atomic literal object* and *object* for *user-defined surrogate object*, although one should note that all database elements (even types and functions) in OSQL are objects. A user type is explicitly created using the `CREATE TYPE` command. A type can be a subtype of other types, giving rise to a subtype/supertype hierarchy. If an object belongs to the extension of a type, it also belongs to the extensions of all the supertypes of this type.

A type extension is populated using the `CREATE` *Type* command (where *Type* is a user type) which creates an object and makes it a member of the extension of *Type*. At the time of object creation, values of various functions on this object may be provided. An object can be made a member of the extension of any type using the `ADD TYPE` command. Each type T gives rise to a unary predicate T that is true of every object belonging to the extension of T.

Functions can have any number of argument and result types. Function values for particular arguments are defined either explicitly (*stored functions*) or implicitly using an OSQL query (*derived functions*). A type inherits all the functions defined on its supertypes (*monotonic inheritance*), but it can also redefine some of them (*non-monotonic inheritance* or overriding).

The following is an OSQL example consisting of a user type definition (together with two stored function definitions), a derived function definition and an object creation command that creates two objects.

```
CREATE TYPE Employee SUBTYPE OF Person FUNCTIONS (
      Name String,
      Subordinates {Employee}
)
```

```
CREATE FUNCTION Boss (Employee e) →Employee b AS
      SELECT b
      FOR EACH Employee b
      WHERE e IN Subordinates(b)

CREATE Employee FUNCTIONS (Name, Subordinates)
      :boria ('Boris Yeltsin',{:sasza, :wania})
      :misza ('Mikhail Gorbachev',{})
```

HOSQL [RDK+91] is an extension of OSQL to the heterogeneous multidatabase environment. It supports multidatabase queries to databases having OSQL-like schemas and a procedural mapping of schemas in other data models, e.g., relational, into HOSQL schemes. HOSQL also permits using database names as prefixes to type or function names in queries to solve name conflicts between autonomous databases. Finally, HOSQL allows the specification of subtype/supertype relationships and the inheritance of functions between types in different databases.

OSQL/IPL [Ann91], adds to OSQL the capability to use derived functions written in a variant of the Scheme programming language.

2.2 Multidatabase mappings

In the multidatabase environment, it is often necessary to use multiple OO databases together. For instance, there may be a type `Employee` in `DB1` and another type `Senior` in `DB2` consisting of all the objects from `Employee` that correspond to employees over 65 years old. Furthermore, schematic discrepancies may need to be resolved, e.g., one might need to view the values of a derived function in `DB1` as types in `DB2`. For example, in `DB1`, the position of an employee may be represented using a function `Position` that maps `Employee` objects to position names. On the other hand, in `DB2`, the type `Employee` may be partitioned into subtypes corresponding to different positions, e.g., `Secretary` or `Manager`.

Consequently, the definition of multidatabase mappings requires richer notions of type and function than those available in OSQL and its dialects. In the context of relational databases, it was postulated [KLK91] that multidatabase mappings involving schematic discrepancies should be formulated in a higher-order logical language. This approach was further pursued and applied to the case of mappings involving multiple data models [BG92]. In our approach, "higher-order" features are obtained without using a higher-order language.

Multidatabase mappings can be categorized along several different dimensions. They can be classified as *explicit* (defined as a set of tuples) or *implicit* (defined using a query expression), partial or total, 1:1 or 1:n or m:n. Of special interest to us is the classification stating which elements of one database are related through the mapping to which elements of the other database. We distinguish the following kinds of database elements: value, object, type and function. Accordingly, there are 16 kinds of mappings written as X$\mapsto$Y where X,Y$\in$ {Value,Object,Type,Function}, for example Value$\mapsto$Type.

Our goal is to specify mappings declaratively. We mostly consider 1:1 and n:1 mappings, conjecturing that other cases can be dealt with using the set (bag) construct of OSQL. Similarly, a mapping involving aggregate objects can be defined using OSQL, once the mappings for the components of those objects have been defined.

At present, only some of the above 16 kinds of mappings can be expressed declaratively in OSQL. Side-effect-free OSQL functions are unable to construct new objects, types or functions, and are therefore limited to representing Value$\mapsto$Value, Object$\mapsto$Value, Type$\mapsto$Value and Function$\mapsto$Value mappings. The last two kinds of mappings can be represented only by accessing the meta-information about the database. The remaining 12 kinds of mappings can be defined in OSQL procedurally. One should use functions whose bodies contain explicit commands to create new objects, types or functions. However, the names of the constructed types or functions are then built on-the-fly and are hidden in the programs that implement the mappings. Therefore, there is no explicit database schema for the data collection resulting from the mapping. This makes the use of this data, e.g., for querying, difficult.

Another problem with the procedural approach is the reliance on the underlying trigger mechanism that is necessary to adjust mappings in the presence of object creations ("mapping faults" [Ken91]) and deletions [1]. Such a mechanism seems to be in conflict with the *autonomy* of different databases. For instance, one would need procedures to monitor insertions and deletions to `Employee` in order to keep the extension of `Senior` consistent with its definition. These problems do not arise in our approach.

We will show now how our extensions to OSQL allow the declarative specification of multidatabase mappings. Table 1 sums up various concepts that we discuss.

3 Derived types

We distinguish several different cases: types defined by a membership condition, type schemas, types defined using object schemas and meta-types.

3.1 Types defined by a membership condition

Any result of a query can be made into a derived type, provided the result consists of surrogate objects. This is consistent with the philosophy of OSQL where user-defined types also consist of surrogate objects.

The syntax for a definition of such a type is very simple:

```
CREATE TYPE Type AS
        Query
```

The syntax of *Query* is that of an OSQL query. A variety of derived types can then be defined, especially subtypes and type unions. For instance, consider the need for mapping the type `Employee` in a database `DB1` into a more specific type `Senior` in another database `DB2`. The corresponding declaration is:

[1] As indicated in [Ken91], the proper handling of object deletions in this context is very much an open problem.

```
CONNECT DB2
CREATE TYPE Senior AS
      SELECT e
      FOR EACH DB1.Employee e
      WHERE DB1.Age(e)>65
```

This definition may be seen as an explicit 1:1 mapping consisting of a pair of types (`DB1.Employee, DB2.Senior`). Also, this definition implicitly maps the extension of `DB1.Employee` into the extension of `DB2.Senior`. The latter mapping is actually an identity mapping which is partial because the `Employee` objects of age less or equal to `65` will not be mapped to anything.

In the above definition, the prefix `DB2` (before `Senior`) is omitted, as the connection to `DB2` is in force. Since name conflict is not our primary concern, we will omit database names and `CONNECT` statements for simplicity. We will come back to these issues in section 5.

In OSQL, at present the only way to define the discussed mapping would be to define `Senior` as a type and populate it initially using the `ADD TYPE` statement, selecting all senior employees from `Employee`. Then, one would need to monitor the changes to the extension of `Employee`, and update the extension of `Senior` accordingly.

Similarly, we can define in our framework another subtype `SpouseAlsoEmployed`:

```
CREATE TYPE SpouseAlsoEmployed AS
      SELECT e1
      FOR EACH Employee e1, Employee e2
      WHERE Spouse(e1)=e2
```

This definition is more complex as it involves a selection, self-join, and projection. Furthermore, as OSQL provides a union operator, *type unions* can be defined, for example:

```
CREATE TYPE Student AS
      SELECT s
      FOR EACH Graduate s
      UNION
      SELECT s
      FOR EACH Undergraduate s
```

This is an example of n:1 mapping between types. The corresponding mapping between the extensions of `Graduate`, `Undergraduate` and `Student` is a total (every graduate or undergraduate is a student) identity mapping.

Summing up, one can declare explicit Type$\mapsto$Type mappings, using derived types defined by a membership condition. Moreover, such mappings can be maintained without requiring procedures for adjustments when type extensions change.

3.2 Type schemas

We now allow the type definition to contain parameters, so that a single declaration can define multiple types. Each individual type corresponds to a combination of parameter values.

The corresponding syntax for a type schema definition is as follows:

```
CREATE TYPE Type[Parameters] AS
       Query
```

The *Query* can contain the parameters appearing in *Parameters*. One can view a type schema as being named by the term *Type*[*Parameters*] [2]. For every substitution that replaces *Parameters* by objects or values of appropriate types, we obtain a derived type, called an *instance type* [3]. This process is called the *instantiation* of the type schema. Instance types may be thought of as named by *ground terms* (terms without parameters). There may be infinitely many instance types if one of the parameters is drawn from an infinite domain, e.g., integers. This creates new issues for query evaluation that we address in Section 6. Type schemas are not types (they don't have extensions consisting of objects), but instance types are.

Example 3.1 Assume that in a database the function `Pos` maps an element of `Employee` into the corresponding position: manager, secretary, etc. Another database models this universe differently, as a family of subtypes consisting of all employees holding the same position. The corresponding mapping from the first database to the second can be expressed as follows:

```
CREATE TYPE EmpPos[String p] AS
       SELECT e
       FOR EACH Employee e
       WHERE Pos(e)=p.
```

For example, the instance type `EmpPos['secretary']` is the subtype of `Employee` consisting of secretaries.

The above definition is an example of an implicit 1:1 Value$\mapsto$Type mapping that maps a string `p` to the type `EmpPos[p]` (an instance type of `EmpPos[String]`). The corresponding mappings of type extensions are partial identity mappings.

Example 3.2 Consider that `EmpDept[Dept]` is a type schema defining subtypes of `Employee` belonging to individual departments and `Contractor[Dept]` is a type schema defining subtypes of `Contractor` hired by individual departments.

One can map these type schemas into that of the set of people paid by a given department:

```
CREATE TYPE Compensated[Dept d] AS
       SELECT e
       FOR EACH EmpDept[d] e
       UNION
```

[2] We use square brackets to distinguish terms naming type and function schemas from function applications.

[3] *Instance* is used here in the logic programming sense (instance of a *term*), not in the object-oriented programming sense (instance of a *type*).

```
SELECT e
FOR EACH Contractor[d] e.
```

Using type schemas, we can formulate implicit Object↦Type and Value↦Type mappings. To express implicit Type↦Type and Function↦Type mappings, function schemas (section 4) and meta-types (subsection 3.4) are necessary.

Moreover, meta-types in conjunction with type schemas make possible the definition of mappings that achieve the effect of dynamically adjusting the type hierarchy (this type of mapping is not properly captured by our classification).

Example 3.3 Assume the following type hierarchy: `Paid` and `Unpaid` are subtypes of `Employee`. We have also a type schema `Emp[Bank]` whose instance types consist of all employees of a given bank. An instance type `Emp[b]` is a subtype of `Paid` if `b` is a solvent bank and a subtype of `Unpaid` if `b` is an unsolvent bank.

Creating types `Paid` and `Unpaid` makes perfect sense, because we may want to define functions applicable only to one of them (for example, unemployment compensation will be paid only to the members of `Unpaid`).

We have to be careful about how those types are populated. Initially, we just have `Employee`. The instances of the type schema `Emp[Bank]` are subtypes of `Employee`. Now `Paid` is the *union* of those instance types `Emp[b]` that correspond to solvent banks. `Unpaid` is defined similarly. Moreover, a given instance type `Emp[b]` is a subtype of either `Paid` or `Unpaid` depending on the status of the corresponding bank `b`.

This can be expressed as follows:

```
CREATE TYPE Paid AS
      SELECT e
      FOR EACH Bank b, Emp[b] e
      WHERE solvent(b)
```

3.3 Object schemas

So far, we have only considered types whose extensions are obtained from extensions of other types. New objects could not be created. However, objects can also be implicitly defined using the same philosophy. We call such objects *derived objects*. Using derived objects, for example, values can be mapped to objects in a declarative way. Derived objects are defined using *object schemas.*

The existence of derived objects is predicated on the existence of appropriate values or objects in the database. The syntax for the definition of an object schema is as follows:

CREATE *Type*[*Parameters*] PROPERTIES (*Functions*) AS
      Query

The semantics is that for every distinct tuple in the value of *Query*, there is a distinct object of type *Type* [4]. The `PROPERTIES` clause is optional, but when present means that all the functions in *Functions* are automatically defined. Each such function takes as an argument an object of type *Type* and returns the result equal to the appropriate component of the tuple out of which the object was formed. This is similar to creating multiple objects using a single `CREATE` *Type* command in OSQL. The only difference is that the population of *Type* may vary in time depending on the value of the *Query*.

Example 3.4 We adopt here the stock market example from [KLK91]. The stock information is represented as a stored OSQL function:

```
CREATE FUNCTION Stock(Date d, String c) → Price p.
```

The second argument of this function is a string representing the name of a company.

Suppose we want to create a new type `Company` and an object of type `Company` for every company name that appears as an argument in the `Stock` stored function. Such an object should also have a property `Name` equal to the company name. We specify this application as follows (the type `Company` is created implicitly):

```
CREATE Company[String c] PROPERTIES (Name) AS
      SELECT DISTINCT c
      FOR EACH Date d, Price p
      WHERE Stock(d,c)=p.
```

Suppose now that the `Stock` function is not defined for any arguments. Then there are no objects of type `Company`. Assume now that through appropriate updates, we obtain:

```
Stock(06/05/95,'IBM') = $10
Stock(06/06/95,'IBM') = $8
Stock(06/06/95,'HP') = $15.
```

Now there are two objects of type `Company`: one with `Name` equal to `'IBM'`, the other – to `'HP'`. The first has the object identifier equal to the Skolem term `Company['IBM']`, the second – to `Company['HP']`.

Object schemas are used to represent implicit Value↦Object (see the example above), Object↦Object, Type↦Object (an example is given later) and Function↦Object domain mappings. The last two kinds of mappings require the presence of type and function schemas, and meta-types.

3.4 Meta-types

Meta-types consist of types or functions. It is convenient to assume two built-in meta-types: `Type` and `Function` containing all types (resp. functions) in the

[4] The OIDs of derived objects may be viewed as *Skolem terms* [Mai86; KW89]. In [AB91], several possible ways of constructing and maintaining OIDs of derived objects are discussed.

database [5]. Type schemas do not belong to `Type`, but instance types do. Similarly, function schemas (section 4) do not belong to `Function`, but instance functions do. So actually `Type` and `Function` may contain infinitely many elements. `Type` and `Function` are populated indirectly using the definitions of types, functions, type schemas and function schemas.

Example 3.5 Suppose we want to construct the inverse to the mapping described in example 3.1. Thus, we are going to create `NewEmp`, the union of all the instance types of the type `EmpPos[String]` (this is an example of a union of an infinite number of types):

```
CREATE TYPE NewEmp AS
       SELECT e
       FOR EACH Type EmpPos[p]
       WHERE EmpPos[p](e).
```

The function `NewPos` that computes the position of a given employee in `NewEmp` is defined as follows:

```
CREATE FUNCTION NewPos :  NewEmp e → Position p AS
       SELECT p
       FOR EACH Type EmpPos[p]
       WHERE EmpPos[p](e).
```

Using meta-types together with type and function schemas, we can represent implicit Type↦Value mappings (as above) and also Type↦Type, Function↦Value, Function↦Type and Type↦Function mappings. In conjunction with derived objects, implicit Type↦Object and Function↦Object mappings can also be represented, as seen in the following example.

Example 3.6 As in the previous example, we start from a type schema `EmpPos[p]`. Assume `p` is a string and we want to create new objects of type `PositionObject` corresponding to instance types of `EmpPos[String]`. We write:

```
CREATE PositionObject[String p] PROPERTIES (Name) AS
       SELECT p
       FOR EACH Type EmpPos[p]
```

4 Derived functions

Derived functions are present in OSQL, so we deal here only with *function schemas*. The syntax of a function schema definition is as follows:

CREATE FUNCTION *Function*[*Parameters*] : *Arguments* → *Result* AS
    Query

All the parts of the function schema definition are the same as in an OSQL function definition, except for the function name that has parameters which can appear

[5] `Type` and `Function` will of course be different for different databases.

in *Query*. A function schema with given values of *Parameters* is just an OSQL function (called an *instance function*), named by a ground term.

Example 4.1 For the OSQL function `Stock` defined in example 3.4, we create a function schema that is parameterized by the name of the company:

```
CREATE FUNCTION StockComp[String c] :  Date d → Price p AS
      SELECT p
      FOR EACH Price p
      WHERE p=Stock(d,c).
```

The instance functions of this function could be, for example, `StockComp['IBM']` or `StockComp['HP']`.

The above is an example of a Value$\mapsto$Function mapping. Object$\mapsto$Function mappings are constructed similarly.

We show now an example of a query that is expressible only if function schemas are present.

Example 4.2 Assume that we have separate functions for every company: `IBMStock`, `HPStock` etc. For each such function, we can define a function schema with a parameter that is a constant (this function has just one instance function). For example:

```
CREATE FUNCTION StockComp['IBM']: Date d → Price p AS
      IBMStock(d).
```

Now, we can ask "higher-order" queries like "which stocks have ever gone above $200?" as follows:

```
SELECT c
FOR EACH String c, Function StockComp[c], Date d
WHERE StockComp[c](d)>$200
```

As indicated earlier in this section, function schemas, used together with other constructs introduced so far, complete the framework necessary for representing all 16 kinds of mappings under consideration (section 2). Moreover, we can now model *integrator* functions [RDK+91] as follows.

Example 4.3 Consider the type schema `EmpPos[Position]` and the function schema `SalPos[Position]` which is a position-specific way of computing the salary of an employee. A function that integrates those functions and computes the salary of every employee in `Emp` is defined as follows:

```
CREATE FUNCTION Salary (Emp e) → Integer AS
      SELECT SalPos[p](e)
      FOR EACH Type EmpPos[p], Function SalPos[p]
      WHERE EmpPos[p](e)
```

If we are concerned that such function may return multiple results which need to be reconciled, `Salary` may be used within a *reconciliation* function.

5 Subtyping and inheritance

5.1 Subtype hierarchy

Establishing subtype relationship between derived types and other types (derived or not) is not an easy task. In OSQL, a subtype hierarchy is finite and static (to change it, explicit `CREATE TYPE` commands have to be issued). In our approach, there may be infinitely many types (instances of type schemas). Moreover, a subtype hierarchy may depend on the database state (see example 3.3). Therefore, we will have to live without an explicit subtype hierarchy. To what degree can we have an implicit one?

A subtype hierarchy is necessary for the proper handling of inheritance. We should be able to determine whether a given function is applicable to a given combination of argument values (monotonic inheritance). If more than one function with the same signature is applicable, then the one with the "most specific" signature has to be selected (non-monotonic inheritance). The first task can, however, be accomplished without consulting an explicit subtype hierarchy. It is enough to perform a number of type membership checks, namely check whether each argument value is of the appropriate type. Derived types are defined using queries, thus type membership checks can be performed using query evaluation.

Non-monotonic inheritance is much more difficult. In some specific cases though, a subtype relationship can be readily established. For example, every instance type `EmpPos[p]` (example 3.1) is a subtype of `Employee`. In [AB91], some general rules for establishing subtype relationships are given, although without considering type schemas. But the problem of providing a *complete* set of such rules and the *decidability* of subtype relationships in the presence of derived types are still open. To appreciate how difficult it is to establish subtype relationships, consider the following derived type definitions:

```
CREATE TYPE T1 AS
      SELECT e
      FOR EACH T e
      WHERE C1(e)

CREATE TYPE T2 AS
      SELECT e
      FOR EACH T e
      WHERE C1(e) AND C2(e)
```

In this example, `T2` indirectly becomes a subtype of `T1`. But to establish that fact, we have to infer that the second `WHERE` condition implies the first, a task that is computationally difficult in general. Moreover, one can not infer subtype relationships just on the basis of derived type definitions, as subtype relationships can depend on the database state. In example 3.3, `Emp[b]` is a subtype of `Paid` if `b` is a solvent bank.

In our approach, subtype relationships between derived types have to be explicitly provided by the user as *subtype declarations*. Of course, such relationships can be partially computed using rules, as in [AB91]. Subtype declarations are used to resolve overloaded function names. If there is no sufficient subtype information to resolve an overloaded function name at run-time, an error is generated, similarly to the case of multiple-inheritance conflicts.

It is the responsibility of the user to ensure that subtype declarations are consistent with the definitions of derived types. Possibly some tools may be created that provide the user with a partial assistance in this task (which also appears computationally hard). However, providing subtype declarations when subtype relationships depend on the database state requires a richer language than the one we are currently considering.

5.2 Inheritance across databases

So far we have considered mappings of different types and functions in separation. But in reality types and functions are intimately related: types serve as arguments and results of functions. Moreover, functions are used to defined derived types.

Consider a multidatabase mapping from `DB1` to `DB2` that introduces a derived type `Senior`:

```
CONNECT DB2
CREATE TYPE Senior
      SUBTYPE OF DB1.Employee AS
      SELECT e
      FOR EACH DB1.Employee e
      WHERE DB1.Age(e)>65
```

Let us consider now what is involved in such a mapping. Clearly, the extension of `DB2.Senior` is defined to be a subset of the extension of `DB1.Employee` consisting of employees satisfying a condition. But mapping the objects alone is not enough: we need to have functions in `DB2` defined on `DB2.Senior` to represent the information about senior employees.

Our approach here is to provide appropriate subtype declarations. The type `DB2.Senior` is declared to be a subtype of `DB1.Employee`. Consequently, the functions defined on `DB1.Employee`, for example `Salary`, can also be applied to `DB2.Senior`. It may be the case that a function f defined on `DB1.Employee` is subsequently redefined in a subtype S of `DB1.Employee`. Then if a `DB2.Senior` object e is in the extension of S, the latter definition will be used to evaluate $f(e)$.

An alternative approach would be to assume no inheritance between different databases. This would be rather inconvenient: type and function definitions from `DB1` would have to be replicated in `DB2` and appropriate mappings between them set up.

For example, to deal with employee salaries in the example above, we would need to define a new function in `DB2` (the notation is simplified):

```
CREATE FUNCTION Salary (Senior e) → Decimal AS
      DB1.Salary(e)
```

If we want to map a function defined in `DB1` into another function in `DB2`, we have to make sure that the argument and result types of the former are mapped to the right types in `DB2` (this is called *type coherence* in [AB91]).

6 Operational semantics

We outline here the operational semantics of derived types and functions. First, we discuss a basic repertoire of operations necessary to implement derived types and functions. Next, we discuss top-down evaluation of derived types and functions.

6.1 Basic operations

The following operations should be supported (in addition to the operations already provided for OSQL):

- *type or function schema instantiation*: obtaining a single derived type (function) from the corresponding schema and given parameter values.
- *membership test* in a derived type or an instance type. This is used to check whether a given object is of the given type.
- *instance enumeration* of all instances of a given type or function schema. This is required whenever the meta-type `Type` (or `Function`) appears as the range of a term in the `FOR EACH` part of an OSQL query (see examples 3.5 and 3.6).

Schema instantiation consists of textual replacement of parameter occurrences in the schema definition by the corresponding parameter values. As a result, a type (or function) definition is obtained which is very much like an OSQL type (resp. function) except for the fact that some types or functions in it may be named by ground terms. Notice that schema instantiation is done at run-time, not at compile-time, because the set of instances of a given (type or function) schema may depend on the database state.

Given an object and a derived type or an instance type, we can check *type membership* by evaluating the appropriate type definition (which is just a query) and checking whether the object is an element of the result. This will work as long as the extension of the type is finite (the type should be *safe*). Determining type safety is a further research issue. Let us just mention that as long as derived types are built from user types using `SELECT` and `UNION`, they will be safe because user types are finite. Only if infinite literal types, e.g., `Integer`, are used in a `SELECT` list, the resulting derived types may be unsafe. Alternatively, we could develop special-purpose routines that check whether an object belongs to the result of a query by substituting the object into the query definition and evaluating the resulting definition.

Instance enumeration is by far the most difficult of the above three operations. It is implemented by *matching* a possibly non-ground term against the entire, possibly infinite, type hierarchy.

Example 6.1 Consider the following definition from example 3.5:

```
CREATE TYPE NewEmp AS
      SELECT e
      FOR EACH Type EmpPos[p]
      WHERE EmpPos[p](e).
```

To enumerate the instance types of the type schema `EmpPos[String]`, the term `EmpPos[p]` is matched against the type hierarchy to obtain the values for `p`. However, the schema `EmpPos[String]` has infinitely many instances, thus there are infinitely many possible values for `p`.

So, if a type schema is to be used for instance enumeration, it should have finitely many instances. In this example, this may be achieved by the introduction of another type `Position`:

```
CREATE TYPE Position AS
      SELECT p
      FOR EACH Employee e
      WHERE Pos(e)=p
```

and the redefinition of the type schema `EmpPos[String]` to `EmpPos[Position]`:

```
CREATE TYPE EmpPos[Position p] AS
      SELECT e
      FOR EACH Employee e
      WHERE Pos(e)=p.
```

In the case, however, where such redefinition is impossible (if, for example, `EmpPos[String]` is defined in a different autonomous database), the definition of `NewEmp` will not lead to infinite enumeration only if the query processor recognizes the fact that the `WHERE` condition in this definition makes it possible to limit the attention to instance types with *non-empty extensions*. In this example, there is only a finite number of such instance types.

Sometimes there is not enough information in the query to allow the query processor to make this type of inference. In the definition from example 3.6:

```
CREATE PositionObject[String p] PROPERTIES (Name) AS
      SELECT p
      FOR EACH Type EmpPos[p]
```

a new derived object should be constructed for every instance of `EmpPos[String]` and the enumeration of those objects is clearly infinite. If we intend, however, to construct new objects only for those instance types that have non-empty extensions, another definition should be given:

```
CREATE PositionObject[String p] PROPERTIES (Name) AS
      SELECT p
      FOR EACH Type EmpPos[p], NewEmp e
      WHERE EmpPos[p](e)
```

where `NewEmp` is the type defined earlier.

Moreover, some type schemas have infinitely many instances with non-empty extensions, for example the following type schema:

```
CREATE TYPE YoungerThan[Integer k] AS
      SELECT e
      FOR EACH Employee e
      WHERE Age(e)<k.
```

Type schemas with infinitely many non-empty instances can not be used where instance enumeration is required. Nevertheless, they can still be used elsewhere in queries.

6.2 Top-down evaluation

We claim that both the evaluation of derived functions and the computation of the extensions of derived types can be handled with only small extensions to the top-down model used in OSQL, *provided the type and function schemas are completely instantiated during the evaluation*. This requirement translates into several syntactic restrictions on the occurrences and definitions of type and function schemas.

Consider a derived function that is defined by a `SELECT` expression. The evaluation of this function consists of the following steps:

1. establishing whether this particular function should be selected for application to the given arguments (checking the validity of arguments and overload resolution).
2. establishing the ranges for the variables (`FOR EACH`). The order of range declarations matters.
3. evaluating the query using the condition (`WHERE`) to select appropriate values.
4. computing the result (the `SELECT` list).

Checking the validity of a function invocation can be accomplished by testing membership of the arguments in the corresponding types (determined from the function signature). This can be done, as shown earlier in this section. If more than one function is applicable, the most “specific” function should be selected, as shown in section 5. For this process to be possible, the signature should consist of fixed types, either OSQL types or instances of type schemas. A type schema can not appear in the signature, because at the time of function invocation, there is not enough information to determine which instance of such a schema should be selected.

In the second step, an appropriate function definition has already been selected. Can a type schema be used as a range for a variable? If all the parameters of such a type are among the arguments of the function invocation or have been bound by earlier range declarations (the order is important), the type schema is fully instantiated and there is no problem (see example 3.3). However, in other cases there is not enough information to uniquely identify the range.

In the evaluation of the query (third step) and the computation of the result (fourth step), a type schema has to be already fully instantiated, as those steps can not instantiate it.

The second and the third step should sometimes be interleaved in order to avoid enumerating infinitely many instance types (see example 6.1 and the discussion that follows).

Function schemas can be safely used in the definitions of other derived functions or in queries if at the time of evaluation their parameters are fully instantiated.

7 Related work

Some of the concepts behind derived types and functions go back to late 1970s. The notions of type defined by a *membership condition* and *type union* were already present in SDM [HM81]. To our knowledge, these proposals have never been implemented. The concept of *derived object* was introduced by Maier [Mai86] and was subsequently refined in [KW89; HY90; AB91]. *Type schemas* (called *parameterized classes*) were mentioned in [AB91] without, however, any study of their properties. This is not surprising because the authors relate the semantics of their language to that of IQL [AK89] in which only databases with a fixed number of classes are allowed. In particular, the notion of *instance type* was not explored. *Meta-types* are present in every OODBMS, but usually contain only a finite number of stored user types or functions, not infinitely many instance types or functions. We are not aware of any mention of *function schemas*, although they are a natural analogue of type schemas. Moreover, none of the above works deals with the operational semantics of type and function schemas. In [AB91], the operational semantics of derived objects is discussed.

Previous OSQL extensions, HOSQL [RDK+91] and OSQL/IPL, adopt a procedural approach to domain mapping. Such an approach suffers from a number of problems, as indicated in section 2. To remedy those, Kent [Ken91] proposed some declarative constructs: *derived subtypes* (types defined by a membership condition) and *covered types* (essentially type unions). Most mappings (particularly those involving schematic discrepancies) are, however, still specified procedurally in his approach. In our approach, the new language constructs are sufficiently expressive to formulate a large class of domain mappings in a completely declarative way.

Our goal was to extend OSQL to support multidatabase mappings. As shown in [BG92], second-order logic can also be used for this purpose. That approach is similar in spirit to F-Logic [KW89; KLW90], a very general second-order logic programming language. In this logic, some of our mappings also can be elegantly expressed.

Consider the definition of the type schema `EmpPos[String]` (subsection 3.2). In F-Logic, we can express it as follows:

```
E : pos(P) ← E : emp[pos→P].
```

The inverse mapping can be equally easily specified by reversing the arrow.

It remains to be seen which approach to multidatabase mappings, one based on logic or one based on a single data model, will be more practical. One may observe, however, that the latter has an advantage of keeping as the kernel the data model that various database users are already accustomed to. Using a logic language to express a mapping between different OSQL databases would require providing two-way translations between this language and OSQL. On the other hand, using logic is considerably more general as it makes possible the mapping between different data models [BG92].

A recent proposal [KKS92] applies the ideas of F-logic to an object-oriented SQL-like language. This paper contains many interesting ideas like various notions of query typing, but does not deal with the issues of type and function schemas.

Recent work on views in object-oriented databases is presented in [SN90; TYI88; KDN90; HZ90; Ber92].

8 Conclusions and further work

We have extended OSQL with constructs making possible a declarative specification of a broad class of OO multidatabase mappings. Using our constructs, schematic discrepancies can be resolved. We summarize the relationships between various kinds of mappings and the supporting concepts in Table 1. We have also outlined the corresponding operational semantics.

Mapping	*Concept*
Object$\mapsto$Value, Value$\mapsto$Value	Derived functions (OSQL)
Type$\mapsto$Value, Function$\mapsto$Value	Derived functions (OSQL) with meta-types
Value$\mapsto$Object, Object$\mapsto$Object	Object schemas
Type$\mapsto$Object, Function$\mapsto$Object	Object schemas with meta-types
Value$\mapsto$Type, Object$\mapsto$Type	Type schemas
Type$\mapsto$Type	Type schemas with meta-types
Value$\mapsto$Function, Object$\mapsto$Function	Function schemas
Function$\mapsto$Function	Function schemas with meta-types
Function$\mapsto$Type, Type$\mapsto$Function	Type and function schemas with meta-types

Table 1: Summary of concepts

This work opens up a lot of interesting problems, e.g., inference of subtype relationships between derived types, safety (finiteness of extensions) of derived types, or enumeration of instance types of type schemas. It is also challenging to characterize more precisely the class of mappings that can be expressed in our framework. Finally, one should extend this work to the problem of interoperability between heterogenous OO languages.

Acknowledgments

Discussions with Bill Kent helped to shape these ideas, as well as comments from Ravi Krishnamurthy and Jurgen Annevelink.

References

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In *ACM SIGMOD International Conference on Management of Data*, 1991.

[AK89] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *ACM SIGMOD International Conference on Management of Data*, 1989. To appear in *Journal of the ACM*.

[Ann91] J. Annevelink. Database Programming Languages: A Functional Approach. In *ACM SIGMOD International Conference on Management of Data*, 1991.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System*. Morgan Kaufmann, 1992.

[Ber92] E. Bertino. A View Mechanism for Object-Oriented Databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *International Conference on Extending Database Technology*. Springer-Verlag, LNCS 580, 1992.

[BG92] T. Barsalou and D. Gangopadhyay. M(DM): An Open Framework for Interoperation of Multimodel Multidatabase Systems. In *IEEE International Conference on Data Engineering*, 1992.

[Fo91] D.H. Fishman and others. Overview of the Iris DBMS. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, 1991.

[HM81] M. Hammer and D. McLeod. Database Description with SDM: a Semantic Data Model. *ACM Transactions on Database Systems*, 1981.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *International Conference on Very Large Data Bases*, 1990.

[HZ90] S. Heiler and S. Zdonik. Object Views: Extending the Vision. In *IEEE International Conference on Data Engineering*, pages 86–93, 1990.

[KDN90] M. Kaul, K. Drosten, and E.J. Neuhold. ViewSystem: Integrating Heterogenous Information Bases by Object-Oriented Views. In *IEEE International Conference on Data Engineering*, pages 2–10, 1990.

[Ken91] W. Kent. Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language. In *International Conference on Very Large Data Bases*, 1991.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In *ACM SIGMOD International Conference on Management of Data*, 1992.

[KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. Language Features for Interoperability of Databases with Schematic Discrepancies. In *ACM SIGMOD International Conference on Management of Data*, 1991.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical report, Dept. of Computer Science, SUNY at Stony Brook, 1990. To appear in *Journal of the ACM*.

[KS91] W. Kim and J. Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*, December 1991.

[KW89] M. Kifer and J. Wu. A Logic for Object-Oriented Programming (Maier's O-Logic: Revisited). In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1989.

[Mai86] D. Maier. A Logic of Objects. In *Workshop on Deductive Databases and Logic Programming*, 1986.

[RDK+91] Ahmed. R., P. DeSmedt, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, and M.-C. Shan. Pegasus: A System for Seamless Integration of Heterogenous Information Sources. In *IEEE COMPCON*, 1991.

[SN90] M. Schrefl and E. Neuhold. Object Class Definition by Generalization using Upward Inheritance. In *IEEE International Conference on Data Engineering*, pages 4–13, 1990.

[TYI88] K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema Virtualization in Object-Oriented Databases. In *IEEE International Conference on Data Engineering*, pages 23–30, 1988.