The ODMG Object Model: Does it Make Sense? *

Suad Alagić
Department of Computer Science
Wichita State University, Wichita, KS 67260-0083, USA
e-mail: alagic@cs.twsu.edu

Abstract

The ODMG Object Model is shown to have a number of problems. A major confusion is caused by the intended type of polymorphism and the way it is expressed in the Model. Dynamic type checking is required even in situations when static type checking is possible. There are situations in which there is no way that type checking can determine whether a particular construct is type correct or not. The model of persistence in the ODMG Standard is not orthogonal, which has undesirable pragmatic consequences on complex objects. The discrepancies between the ODMG Object Model and the particular language bindings of the ODMG Standard are non-trivial. This paper presents solutions to some of these problems together with the associated formal system. Without such a formal system the recommended ODMG bindings are open to a wide range of different, and sometimes confusing interpretations. The criticism expressed in the paper is intended to be helpful in developing future releases of the ODMG Standard.

*This material is based upon work supported in part by the U.S. Army Research Office under grant number DAAH04-96-1-0192.

OOPSLA '97 10/97 GA, USA

1 Introduction

The ODMG Object Model is the core of the ODMG Standard [Cat96]. As the Standard has been developed by the major vendors of object-oriented database management systems, the model deserves careful attention. Indeed, we may have to live with it for many years to come.

In this paper we show that the ODMG Object Model is behind the current state of object-oriented research. We reveal a number of problems in the Model and propose their solutions. We naturally hope that future releases will not have the problems reported in this paper.

The paper was written when the Release 1.2 [Cat96] was available. In the phase of the final preparation of the paper for the proceedings, ODMG 2.0 appeared [CBB97]. It was very difficult to make a careful account of the effects of all the changes in the new version of the Standard. Every attempt has been made to do so within the severe time constraints. Some differences between the two versions of the ODMG Standard will be pointed out in the paper. We were very pleased to see that the new version of the ODMG Standard seemed to recognize some of the problems reported in this paper.[1] But ODMG 2.0 is still far enough from a consistent standard. The core of our criticism still applies to the 2.0 version of the ODMG Standard just as well.

This paper also presents the core of a formal system which may be used to specify the intended bindings of the ODMG Object Model. Without such a formal system those bindings are open to a wide range of

[1]The first version of this paper was completed in October of 1996.

sometimes very confusing interpretations.

The difficulties in coming up with a good object model for the ODMG Standard are understandable. This paper thrives toward conceptual and even formal clarity. It aims to show that it is possible to achieve these goals, and at the same time meet the very pragmatic goals of the Standard.

A major confusion exists in the ODMG Object Model regarding the polymorphic facilities which the Model is actually supporting. The Model has a very inappropriate way of expressing situations requiring parametric polymorphism, which causes major problems in the underlying type system.

A related group of problems in the ODMG Object Model covers situations in which dynamic type checking is required in spite of the fact that static type checking is possible. Excessive dynamic checks affect efficiency and reliability of database transactions. It is unacceptable to carry out a dynamic check every time a transaction is executed if the check could have been performed once and for all when the transaction was compiled. Perhaps worse, expensive recovery procedures may have to be invoked only because of run-time type errors which could have been detected at compile time.

Yet another group of problems are those coming from the rigidity of the ODMG Object Model which makes it impossible to express frequently occurring situations in a natural way. This forces database programmers to use techniques which bypass the type system. A very likely and frequent consequence is undetected type errors that may cause database transactions to fail at run time.

A further class of problems are those in which it appears that type safety has been established by static type checking, and yet a transaction fails at run time due to a type error. Perhaps the most embarrassing are situations in which there is no way that type checking can determine whether a particular construct is type correct, no matter how sophisticated the checker may be. Such situations do exist in the ODMG Object Model. But they can be avoided, and we show how.

Particularly important are those aspects of the ODMG Object Model which are related to the model of persistence. The ODMG Standard does not have an orthogonal model of persistence. This leads to specific pragmatic problems related to complex objects. We offer an orthogonal solution for a model of persistence which is tied with the proposed formal system.

A closely related fact is that object-oriented database type systems necessarily require dynamic checks in some situations. It is essential to make those situations explicit, and not to confuse them with situations in which dynamic checks are not necessary. Our analysis is intended to eliminate such a confusion from the ODMG Object Model.

When introducing desirable features of a typed object-oriented model, one should have in mind what kind of demands those features place on the type system, and what kind of type technology is required to guarantee efficiency and reliability. This paper reveals some of such hidden subtleties in the ODMG Standard. They require advanced typing techniques, as well as a variety of reflective techniques elaborated to some extent in the paper.

This paper consists of two parts. The first part of the paper (sections 1–7) is informal. It presents the problems in the ODMG Object Model and our solutions to these problems. The second part of the paper (sections 8 – 14) presents the core of a formal system for a family of ODMG Object Models.

Many of the problems in the ODMG Object Model come from the fact that it is intended to be common for programming languages such as C++ and Smalltalk. But the underlying paradigms of these two languages are so different: C++ is strongly and mostly statically typed, and Smalltalk is mostly untyped (or dynamically typed). A reflective paradigm which treats classes as objects ([Feb89], [OPS95]) is truly object-oriented, but it defeats the whole purpose of a type system. It is thus hard to imagine that the requirement of the ODMG Standard to have a unified type system across the database and the programming language can ever be satisfied.

This paper shows that it is in fact possible to define a family of ODMG Object Models of increasing level of sophistication. Unlike the existing ODMG Object Model, each model in this family may be well and even formally defined, without any of the confusing points in the current ODMG Object Model.

2 The top class interface

All class interfaces in the ODMG Object Model are derived by inheritance from the interface `Object`. But in addition, the ODMG Object Model contains a mysterious type any. any stands for any type and it is used in situations where a type parameter is in fact required. This produces a type system which is not sound. Since the ODMG Object Model contains `Object`, any is redundant, and should be identified with `Object`, as illustrated below.

```
interface Collection : Object {
  unsigned long cardinality();
  boolean is_empty();
  void insert_element(in Object element);
//ODMG: void insert_element(in any element);
  void remove_element(in Object element);
  boolean contains_element(in Object
                                    element);
//ODMG: void contains _element(in any
                                    element);
  Iterator create_iterator();
};
```

With exception handling omitted, this is how the iterator class should look like:

```
interface Iterator: Object {
  void reset();
  boolean not_done();
  boolean next(out Object next_obj);
//ODMG: boolean next(out any next_obj);
  void advance();
  Object get_element();
//ODMG: any get_element();
};
```

The usage of any leads to obvious type errors that cannot be detected at compile time. Here is an example:

```
Iterator p;
p.reset();
select p.get_element().salary();
```

In our model the select clause fails a static type check. But what happens in the ODMG Object Model? The result type of `get_element` is any, where any stands for any type. Thus it is impossible for the type checker to determine whether the select clause is type correct. Does any have a method salary? All the types certainly do not.

In order to introduce a dynamic check one would have to write `((Person)(p.get_element()).salary()`. But the class indicator is intended to be used for going down the inheritance hierarchy and `Person` is not derived by inheritance from any. It is thus unclear whether this solution is even possible and correct according to the ODMG Object Model. ODMG 2.0 has an exception `InvalidCollectionType` in the interface `iterator`. When any is identified with `Object`, the problem disappears.

This simple change produces a model which requires usage of `Object` where one would use a type parameter in a model which supports parametric polymorphism. Using `Object` in place of a type parameter, as one would do in Java^{TM}, requires downcasting, which means dynamic checking. This is contrary to parametric polymorphism, which can be implemented with static type checking. In addition, quite contrary to the intent of parametric polymorphism, using `Object` implies heterogeneity of types of collection elements. But at least the model is consistent, unlike the current ODMG Object Model. And in fact, this basic model corresponds to the Java binding which appears in the ODMG 2.0 Standard.

3 Persistence

A model of persistence is orthogonal if objects of any type may be persistent or transient [ABD89], [AM95]. In a type system with orthogonal persistence, there may exist both persistent and transient objects of any type. *The model of persistence in the ODMG Standard is not orthogonal.* It is based on persistence capable classes. Objects of persistence capable classes may be persistent or transient. Objects of other classes are always transient.

In Release 1.2 persistence capable classes must be declared as such in the schema. In ODMG 2.0 persistence capable classes are eliminated from ODL, but then they reappear both in the C++ and the Java bindings (but not in the Smalltalk binding). Intriguing examples of classes the are not persistent capable according to the ODMG 2.0 are `Transaction`

(this rules out long transactions), `Database` (!?) and `d_Extent` (used for managing extents of persistence capable classes in the C++ binding).

In the C++ binding of the ODMG Standard persistence capable classes, in addition to being derived from other classes, must also be derived from a distinguished class `d_Object` which enables persistence. But `d_Object` is not the top of the inheritance hierarchy. In fact, there is no such class in the C++ type system. The model of persistence is thus not orthogonal. Similar techniques are used in several extensions of C++ supporting persistence [AG89]. There are in fact even less flexible solutions. For example, E [RCS93] makes a clear distinction between database classes, whose objects are always persistent, and other classes, whose objects are transient.

ODMG 2.0 Java binding does not specify how a class becomes persistence capable.

The lack of orthogonality has its disadvantages. For example, consider a class Shape for which persistence capability has not been declared in the schema. If we now have a complex persistent object which has a shape, an awkward situation occurs. A persistent object has a component which cannot be persistent. Of course, we can define a new class `Persistent_shape`, equipped with persistence capabilities. But does it really make sense to have both Shape and `Persistent_shape` in the model?

The Java binding of the ODMG 2.0 struggles with this issue trying to reconcile non-orthogonal persistence with persistence by reachability. But the whole point is that persistence by reachability requires orthogonal persistence. Persistence by reachability is essential for object-oriented database technology, since it allows proper handling of complex objects. Support for complex objects is one of the key advantages of the object-oriented database technology over the relational database technology.

For the sake of completeness and fairness, we would also like to mention a database architecture which is object-oriented, largely untyped and reflective, and which does not support persistence by reachability [OPS95]. At the same time, a very active research and development work is under way to extend Java with orthogonal persistence. A representative project is PJava [ADJ96].

This paper offers a truly orthogonal, object-oriented model of persistence. The model is based on inheritance and on message passing. This last feature makes persistence in our model per object based. The class interface `Object` is extended by a method `persists` and a test `is_persistent`. Both are thus inherited by all other classes, and none of this disturbs any of the properties of the type system.

```
interface Object{
  boolean same_as(in Object anObject);
  boolean is_persistent();
  Object copy();
  void delete();
  void persists(in string name);
};
```

The message `persists` promotes the receiver object to longevity and makes the argument string the name of this object in the currently valid persistent scope. In a model with single, flat name space per database, this scope would correspond to the currently opened database. This binding thus corresponds to the effect of the method `bind` of the `Database` interface that follows.

Persistence capabilities are thus associated with the top of the inheritance hierarchy. This technique works well with Java, Smalltalk and Eiffel. In fact, we would argue that there is probably no other more natural way of supporting generic and orthogonal persistence capabilities in object-oriented systems. In spite of that, we are not aware of a single object-oriented model of persistence that is based on this simple observation.

An extension of the interface `Object` in ODMG 2.0 associates locking primitives with this interface. This is just yet another point of disagreement that we have with the ODMG Standard. Messages for promoting an object to longevity belong to the user interface. Locking is merely an implementation technique and should not be placed in the user model, just as persistence implementation techniques do not belong there.

Another component of a model of persistence is a binding mechanism for names of persistent objects. The ODMG Standard supports a single, flat name space which corresponds to the entire database. This is very impractical and it is expected to be changed

in future revisions of the Standard. In fact, ODMG 2.0 has a very elaborate meta-level support which includes schemas, scopes and modules. We still have not been able to understand how this elaborate addition, and a single, flat name space, coexist in a single model.

The binding mechanism for names of persistent objects is given in the `Database` class interface. As in other models of persistence, this is an instance where explicit dynamic type checking is required.

```
interface Database: Object {
  void open(in string database_name);
  void close();
  void bind(in Object an_object,
                        in string name);
  Object lookup(in string object_name);
  // ODMG 1.2:
  // any lookup(in string object_name);
};
```

Note that the above mechanism leads naturally to orthogonality of persistence. Named objects are persistent (these are called persistence roots), and so are all their components, direct or indirect (reachability). This is essentially the approach taken in PJava [ADJ96]. Persistence by reachability or transitive persistence also appears at several places in ODMG 2.0, but it is somehow combined with persistence capable classes.

The confusion about the type any in the ODMG Object Model applies to this situation as well. Here is an illustration which shows how object lookup may be performed.

```
Database d; Person p;
p=d.lookup("John Doe");
```

According to the ODMG C++ binding `d.lookup( "John Doe")` returns a reference of type any. The object is then fetched and its type checked. If that type is not `Person`, a run-time error occurs. One problem with this is that it appears that assignments cannot be statically type checked. But that is not true. The usual rule in object-oriented languages is that the compile-time type of the right-hand side must be derived by inheritance from the compile-time type of the left-hand side. In order to guarantee type safety by static type checking, a more restrictive discipline must be adopted. The compile-time type of the right-hand side must be a subtype of the compile-time type of the left-hand side.

In the model proposed in this paper two solutions are possible. Both apply to situations in which dynamic type checking is required when retrieving a persistent object. One is based on the reverse assignment ?= as in Eiffel, which necessarily involves a dynamic type check:

```
p ?= d.lookup("John Doe");
```

If ordinary assignment were used above, it would not have type checked. The type of the right-hand side is not derived by inheritance, nor is it a subtype of the left-hand side. But the reverse assignment type checks, because it generates a mandatory run-time check. Such a check is not generated with the ordinary assignment. If the fetched object is indeed of the type `Person`, its identity will be assigned to p. Otherwise, p will be assigned a nil identifier. Testing p after the assignment reveals what happened.

The other solution is more in the spirit of the ODMG Standard. According to the `Database` interface, all that can be specified for the type of an object with a given name fetched from the database is that its type is any. In our type system that type is `Object`. In either case, very little can be done with such an object. A dynamic type check asserts that the fetched object is of a specific, expected type.

```
Database d; Person p;
p=(Person)(d.lookup("John Doe"));
```

But the above solution does not work according to the ODMG Standard 1.2. The class indicator `(Person)` should not be applied to the result of the method `lookup`. Indeed, the result type of `lookup` is any, and `Person` is not derived by inheritance from any. If any is identified with `Object`, the problem disappears.

To prove our point, in ODMG 2.0 the result type of the method `lookup` is indeed `Object`, as it should be. But then a new interface `Dictionary` is introduced in ODMG 2.0 with a method `lookup` whose result type is again any.

```
interface Dictionary: Collection {
  exception KeyNotFound(any key);
  void bind(in any key, in any value);
  void unbind(in any key)
                  raises(KeyNotFound);
  any lookup(in any key)
                  raises(KeyNotFound);
  boolean contains_key(in any value);
};
```

Our criticism applies just the same to this new interface, as well as to the retrieve methods of interfaces `List` and `Array` of both versions of the ODMG Standard. Furthermore, in the C++ binding of the ODMG 2.0 the result type of the method `lookup_object` of the class `d_Database` is `d_Ref_Any`, which causes even worse problems. At the same time, the class `d_Dictionary` of the C++ binding makes proper usage of parametric polymorphism in specifying the result type of the method `lookup`.

In the conclusion of this section we point out a major controversial issue related to persistence: object identifiers. It appears that in the ODMG Object Model these identifiers are hidden from the users, as they should be. But in the C++ binding of the ODMG Standard object references are explicitly present in the language.

This is contrary to years of research and experience in database systems. It is also contrary to the intent of the object-oriented data model as presented in [ABD89]. It is hard to imagine a type system that can deal with all the problems caused by the explicit availability of object identifiers to the users. We hope that this dangerous feature is not a consequence of the ideas behind the ODMG Object Model. Rather, it is a consequence of compliance with C++, which has references explicitly in the language. Regrettably, other C++ database programming languages suffer from the same problem. An example is O++ [AG89].

Java does not have this problem, and neither does the ODMG 2.0 Java binding.

4 Self types

In this section we show that the ODMG Object Model is in fact too rigid. This problem is not specific to the ODMG Object Model alone. This rigidity guarantees type safety in most situations. But then there are situations in which a database programmer must bypass completely the type system in order to accomplish perfectly natural tasks. The consequences may be very serious. A more flexible model is obtained by using self types [BCM93], [BSG95], [AC96], as illustrated below:

```
interface Object{
  boolean same_as(in Object anObject);
  MyType copy();
 // ODMG: Object copy();
  void delete();
};
```

This definition differs from the ODMG Object Model only in the result type of the copy method. But this seemingly small difference has major implications. `MyType` is a distinguished type variable denoting the type of object executing the message. `MyType` allows more flexible typing discipline in comparison with the ODMG Object Model since it changes its interpretation in a derived class. For example:

```
interface Person: Object{
  unsigned short age();
};
interface Employee: Person {
  float salary();
};
Person John; Employee Doe;
...
select John.copy().age();
select Doe.copy().salary();
```

In the class `Person`, `MyType` as the result type of the method `copy()` stands for `Person`, and in the class `Employee`, `MyType` stands for `Employee`. Because of that both select clauses type check. But according to the ODMG Object Model, neither one of them would. The reason is that the result type of `copy` is `Object`, which is not equipped with a method `age`, nor with a method `salary`.

One could argue that the problem may be solved using the class indicator as follows:

```
select ((Person)John.copy()).age();
select ((Employee)Doe.copy()).salary();
```

But according to the ODMG Object Model, the class indicator implies a dynamic check, which is not necessary in a model with self types. Besides, what is the point in making a copy of an object in such a way that the copy apparently contains only the features of the class `Object`. If a copy of a person object is created, presumably the copy should be a person object as well. Thus it appears that copy should be redefined in the derived classes accordingly. Although the ODMG Object Model is not specific on this point, it appears that such a redefinition is not possible. It is certainly forbidden in the Java and the C++ bindings of the ODMG Standard. Yet, if it were allowed, such a redefinition would have been known at compile time, and then the check could be static. This is an instance which shows the rigidity of the type system underlying the ODMG Object Model.

A closely related issue is the thorny problem of object creation which was completely avoided in ODMG 1.2. ODMG 2.0 attempts to deal with it, as illustrated by the following two interfaces.

```
interface ObjectFactory{
  Object new();
};

interface CollectionFactory: ObjectFactory
{
  Collection new_of_size(in long size);
};
```

ODMG 2.0 states that the operation `new` inherited by `Collection` from the interface `Object` creates a `Collection` object with system-dependent default amount of storage for its elements. But the result type of `new` is `Object`, and not `Collection`!

The C++ binding of the ODMG 2.0 contains specification of `new` based on overloading which completely bypasses the type system:

```
class  d_Object {
 ...
 void* operator new(size_t size,
                    d_Database *database,
                    const char *typename);
 ...
};
```

As if the result type `void` was not enough, the physical size of the newly created object appears as the parameter of `new`, and the type name is passed at run-time as a string. If a programmer makes a mistake, the consequences may be disastrous. The whole point about the operator `new` in C++ is that it is meant to be type safe. ODMG 2.0 has an unacceptable explanation for how anomalies would be avoided in the above situation [CBB97].

In a type system which is based on self types, a distinction must be made between inheritance and subtyping. If an interface B is derived by inheritance from an interface A, contrary to the ODMG Object Model, it does not follow that B is a subtype of A. Substituting an instance of B where an instance of A is expected is allowed only if B is in fact a subtype of A. Otherwise, type safety cannot be guaranteed by static type checking. Multiple dispatch techniques must be used (see, for example, [BC96], [Ala97]) in order to guarantee type safety. A more expressive model is thus obtained at the expense of a strictly more sophisticated type technology, and even more sophisticated run-time technology. The tradeoffs must be well understood.

For example, covariant extensions of the signatures of the inherited methods are allowed in O2. With single dispatch this produces a system which is not type safe.

The formal rules for inheritance and subtyping are specified starting with section 9.

5 Parametric interfaces

Unlike Eiffel and C++, the ODMG Object Model does not offer general support for parametric classes. It has a roundabout way of supporting it only for collection classes, via "type generators". This view is shared by O2. The most likely reason for this controversial issue is the fact that the ODMG Standard defines bindings for both C++ and Smalltalk, as well as for Java in ODMG 2.0. The latter two of course do not support parametric polymorphism. C++ supports parametric polymorphism and thus the C++ binding of the ODMG Standard does not have many of the problems discussed in this paper. But the C++ binding has problems of its own. They include a model of persistence which is not orthogonal, and explicit presence of object identifiers in the language.

A clean solution from the viewpoint of the type system is to have explicit support for generic or parametric classes, as specified below:

```
interface Collection <T> {
 //ODMG:  interface Collection: Object {
   ....
  Iterator<T>  create_iterator();
 // ODMG: Iterator create_iterator();
};
```

Explicit support of parametric polymorphism allows type safe iterators, as illustrated below. A point of confusion in the ODMG Standard is that the ODMG Object Model does not define `Iterator` as a parametric class and its C++ binding does.

```
interface Iterator<T> {
  //ODMG: interface Iterator: Object {
      ....
      T get_element();
  //ODMG: any get_element();
};
```

Type safety can now be accomplished with static type checking, as illustrated below:

```
Iterator<Person> p;
  //ODMG: Iterator p;
  ....
select p.get_element().age();
```

The select clause type checks. In a model without parametric polymorphism one would have to introduce a dynamic check. One of the advantages of parametric classes is that they can be handled with static type checking.

The problems with the lack of parametric polymorphism and the rigid typing discipline for binary methods come up in deriving class interfaces `Set` and `Bag` from `Collection`. This is how it works in a type system with parametric polymorphism and self types:

```
interface Set <T> : Collection<T> {
  // ODMG: interface Set: Collection {
  MyType union_with(in MyType other);
  // ODMG: Set union_with(in Set other);
  MyType intersection_with(in MyType other);
  MyType difference_with(in MyType other);
  boolean is_subset_of(in MyType other_set);
  //ODMG:
  //boolean is_subset_of(in Set other_set);
  boolean is_proper_subset_of(
                      in MyType other_set);
  boolean is_superset_of(
                      in MyType other_set);
  boolean is_proper_subset_of(
                      in MyType other_set);
};
```

Consider now an example involving a class interface derived by inheritance from `Set`. It is unclear how one would accomplish this with type generators available in the ODMG Object Model for the collection classes `Collection, Set, Bag, List` and `Array`. There is no provision in the model to derive one type generator from another. This problem does not appear in the C++ binding of the ODMG Standard.

```
interface Bounded_Set <T> : Set<T> {
  unsigned long bound();
  void set_bound(in unsigned long
                                  new_bound);
};
Bounded_Set<Person> Employees, Candidates;
...
select
Employees.union_with(Candidates).bound();
```

According to the ODMG Object Model the expression `Employees.union_with(Candidates).bound()` fails to type check. The type of the result of `union_with` is `Set<Person>` and thus does not have a bound. In our more flexible type system the type check succeeds, as one would naturally expect. Using the class indicator leads to the issues already discussed.

It is essential to note that `Bounded_Set` as derived above by inheritance from `Set` is not a subtype of `Set`. This is caused by the appearance of `MyType` in the argument position, and follows from the rules given in sections from 9 onwards. The remarks given in section 4 on the required technology which handles properly this situation apply.

6 Higher-order typing

The C++ binding of the ODMG Standard reveals further subtleties of the type system required by the in-

tended ODMG Object Model. The type of elements of a collection cannot be just any type. Elements of a collection are required to be equipped with methods for the copy semantics. The type of polymorphism is thus not even parametric, it is more complicated. An example of an element type that satisfies this requirement is given below:

```
interface Person {
   unsigned short age();
   Person copy();
   boolean equal(in Person e);
};
```

The type of polymorphism which captures the intent of this ODMG requirement is either matching [BSG95] or F-bounded polymorphism [CCH89], [Ala94]. Parametric polymorphism is its particular case. This is how it works. Define the element type of a collection type as:

```
interface Element<T> {
  T copy();
  boolean equal(in T e);
 };
```

According to the ODMG Standard elements of ordered collections are also required to be equipped with a comparison method. Because of that, we also define the element type of ordered collections as follows:

```
interface Ord_element<T> : Element<T> {
  boolean less_than(in T e);
 };
```

A generic collection type whose elements are equipped with the copy semantics is now defined as follows:

```
interface Collection
      <T | T subtype_of  Element<T> >{
      ...
};
```

An F-bounded condition on the type parameter `T` is `T subtype_of Element<T>`. For example, `Person` as defined above satisfies this condition. A particular case of F-bounded polymorphism is bounded quantification (constraint genericity in Eiffel [Mey92]). To obtain this particular case, F-bound is replaced with a fixed class, for example `T subtype_of Ordered_element`.

A collection type whose elements are equipped both with the copy semantics and a comparison function is defined as follows:

```
interface Ordered_collection
       <T | T subtype_of Ord_element<T> >{
     ...
};
```

Note that `Person` now does not satisfy the condition `Person subtype_of Ord_element<Person>`.

F-bounded polymorphism is not just an exotic typing notion. It is in fact required for typing some very fundamental components of most database system architectures. Consider what is involved in typing the index abstraction:

```
interface Order<T> {
      boolean less_than(in T element);
 };

interface Index <T,To | T inherits_from To,
                 To subtype_of Order<To>>
{ void build(Collection<T> collection);
  Iterator<T> iterator();
  T find(To key);
  void assign_iterator(Iterator<T>
                                  iterator);
};
```

`Index` has two type parameters. `T` stands for the type of elements of the underlying collection, and `To` for the type of the indexing (key) attributes. There are two type constraints. The first one requires that the set of features of `To` is a subset of the set of features of `T`, hence `T inherits_from To`. The other constraint on `To` guarantees that `To` is equipped with the comparison function, hence `To subtype_of Order<To>`.

Similar subtleties are involved in typing keyed collections and dictionaries that appear in ODMG 2.0. Among the existing typed object-oriented programming languages, Eiffel is the closest to being able to deal with these subtleties.

7 Reflection

There are both hidden and explicit reflective features in the ODMG Standard. An elaborate meta-level extension in ODMG 2.0 is likely to be relevant to the discussion in this section. However, due to the time constraints, we have not been able to evaluate fully this new development in the Standard.

7.1 Type reflection

Type reflection treats types as objects of computation. This is possible only in a very restrictive fashion in order not to defeat the whole purpose of the type system. A conservative approach proposed in this paper is to allow only construction of new types at compile time. The approach is not only type safe, but also based on static type checking.

The simplest form of type computation carried out at compile time is type substitution required for parametric interfaces. Type reflective facilities in the ODMG Object Model appear in the form of type generators. The ODMG Object Model has type generators for collection classes `Collection<T>`, `Set<T>`, `Bag<T>`, `List<T>` and `Array<T>` as a roundabout way of providing limited parametric polymorphic features.

According to the ODMG 1.2 Object Model `Struct` is a type generator which allows definition of application oriented structures. Methods applicable to such structures are defined in the following interface.

```
interface Struct{
  unsigned long size();
  void set_element(in any field,
                   in any value);
  any get_element(in  any field);
  void clear_element(in  any field);
  Struct copy();
  void delete();
};
```

Apart from the fact that the above interface includes the usual confusion related to the type any, it also raises a question whether structures are objects. The intent of the ODMG Object Model is that structures are not objects, but according to the above ODMG interface they are. If the idea is to make structure types objects, then a type becomes a run-time notion. It is hard to imagine a type system that would offer a resolution to this confusing situation. To prove our point, this interface has been eliminated from ODMG 2.0.

The ODMG Object Model is intended to subsume the relational model. In ODMG 1.2 `Table(a1: t1, a2: t2,..., an:tn)` is defined to be equivalent to `Bag<Struct<a1 t1,a2 t2,...,an tn>>`. ODMG 2.0 states that `Table` is semantically equivalent to a collection of structures. But the ODMG Standard does not say anything about relational operators that `Table` should be equipped with, in addition to the operators on bags. Quite contrary to the ODMG Standard, a natural and truly object-oriented approach leads to a definition of `Table` as a class derived by inheritance from the class `Bag`.

With proper usage of parametric polymorphism and self types, a collection class `Bag` of the ODMG Object Model can now be defined as follows:

```
interface Bag <T> : Collection<T> {
 // ODMG: interface Bag: Collection  {
 unsigned long occurrences(in T element);
 MyType  union_with(in MyType other);
 // ODMG: Bag union_with(in Bag other);
 MyType intersection_with(in MyType other);
 MyType difference_with(in  MyType other);
 };
```

A class `Table` can now be derived from `Bag` as follows:

```
interface Table <T> : Bag<T> {
  Table<T*R> natural_join(Table<R>);
};
```

In order to illustrate the issues, only one specific database operator has been introduced in the interface `Table`: the natural join.

Typing a generic natural join function causes well-known difficulties [SFS90], [OBB89]. The solution presented above is based on type computation carried out at compile time. This computation is expressed in terms of operators on types [Car88]. Two such operators are *meet*, denoted *, and *join* of types, denoted +. Interestingly enough, typing joins requires meets of types. Type computations are necessarily very restrictive, so that they always terminate.

7.2 Linguistic reflection

A major step toward more general reflective facilities which typically cannot accommodate static type checking is in treating programs (transactions and queries) as objects. This is exactly what the ODMG Object Model does. Transactions are defined as objects of a particular class. The C++ binding of the ODMG Standard also treats queries as objects. In order not to obscure the main idea by C++ specifics, we will maintain the style of presentation of this paper. In our view queries may be viewed as objects of a class with the following interface:

```
interface Query<T> {
  void construct_query(in string query);
  void pass_int_arg(in unsigned short arg);
  void pass_float_arg(in float arg);
  void pass_string_arg(in string arg);
  Collection<T> execute();
    ....
};
```

The corresponding class in the C++ binding of the ODMG Standard is not parametric, which only causes further unnecessary dynamic checks. The method `construct_query` takes a string representing a query and associates this query with the object executing this message. Actual parameters can then be passed to the query using methods such as `pass_int_arg`, `pass_float_arg`, `pass_string_arg`, etc. This amounts to reconstructing a query at run time.

The method `execute` applied to a query object returns a collection which is the result of the query. It requires a compiler that can be called at run-time. Type checking will be thus carried out by the compiler as usual. But it will happen at run time. This technique is called run-time reflection [SSS92], [AM95]. The type parameter `T` stands for the type of elements selected by the query. Type checking involved in the execute message ensures that the argument query of this method indeed has this property. Here is a specific example of usage of this parametric class:

```
Query<Person> q;
.........
q.construct("select p  from Person p
  where p.age > $1 and p.salary >=  $2");
.........
q.pass_int_arg(40);
q.pass_float_arg(75,000.00);
q.execute();
.........
q.pass_int_arg(55);
q.pass_float_arg(100,000.00);
q.execute();
```

Run-time reflection based on a callable compiler has been a part of the technology of persistent programming languages such as Ps-algol [PsA87] and Napier [MBC88]. However, expressing it in a formalism of typing rules presents a real challenge. A further point is that type checking of the above argument query requires type inference [OB89], [OBB89], because of the usage of $1 and $2, as required by the ODMG Standard.

8 Schemas

We now proceed with a formal development. Names in a schema are bound to classes, objects and class extents. Every schema should contain at least a binding for the class `Object`. We thus have:

$$\overline{\{interface\ Object\ Ro\}\ schema}$$

A statement of the form S *schema* asserts that S is a schema. $dom(S)$ denotes the set of identifiers for which a binding is specified in a schema S. In addition to `Object`, a schema will naturally contain other predefined bindings from the ODMG Standard, such as those for predefined simple types and collection types.

A binding for a class is introduced by a statement of the form `interface C R`, where `R` is a structure expression containing signatures of methods of the class `C`. For example, `Ro = {boolean same_as(in Object anObject); MyType copy(); void delete();}`. Every valid structure expression is formally derived from the empty structure expression. This relationship is expressed as $R \leq \{\}$;, where $\leq$ is the inheritance ordering. The inheritance ordering for structures is defined in section 10 and the inheritance ordering for object types in section 9.

The rule for binding an identifier to an interface presented below is recursive since an interface (such as `Object` above) often naturally refers to itself, hence the condition $C \leq Object$. The axiom $\mathcal{S}\ schema \vdash MyType \leq Object$ is naturally a part of the formal system.

$$\frac{\mathcal{S}\ schema,\ \ C \notin dom(\mathcal{S}),\ \ \mathcal{S}, C \leq Object \vdash R \leq \{\};}{\mathcal{S} \cup \{interface\ C\ R\}\ schema}$$

$\mathcal{S}$ will always denote a schema, so that the statement $\mathcal{S}\ schema$ will in most cases be abbreviated to $\mathcal{S}$. Thus $\mathcal{S}$ stands for a collection of assertions about bindings of identifiers to interfaces (object types), extents and objects.

A binding for a class `C2` derived by inheritance from a class `C1` is introduced by a statement of the form `interface C2: C1 R`, hence the rule:

$$\frac{\mathcal{S}\ schema,\ \ interface\ C1\ R1 \in \mathcal{S},\ \ C2 \notin dom(\mathcal{S}),\ \ \mathcal{S}, C2 \leq Object \vdash R2 \cup R1 \leq \{\};}{\mathcal{S} \cup \{interface\ C2 :\ C1\ R2\}\ schema}$$

The expression $R2 \cup R1 \leq \{\}$; reflects the requirement that no redefinitions of the inherited signatures are allowed, other than those caused by the change of interpretation of `MyType`.

If `C` is bound to a class `C`, then `C` is necessarily derived by inheritance from the class `Object`.

$$\frac{\mathcal{S}\ schema,\ \ \ interface\ C\ R\ \in \mathcal{S}}{\mathcal{S} \vdash C \leq Object}$$

Likewise, if a schema `S` contains a binding `interface C2:C1 R2`, then the inheritance relationship expressed as $C2 \leq C1$ is derivable from the schema `S`.

$$\frac{\mathcal{S}\ schema,\ \ \ interface\ C2 : C1\ R2\ \in \mathcal{S}}{\mathcal{S} \vdash C2 \leq C1}$$

The rules which allow parametric interfaces to be declared in a schema are given in section 12.

An identifier a is bound to an object (yet unspecified) of a class `C` according to the following rule:

$$\frac{\mathcal{S}\ schema,\ \ \ \mathcal{S} \vdash C \leq Object,\ \ \ a \notin dom(\mathcal{S})}{\mathcal{S} \cup \{C\ a\}\ schema}$$

If a schema $\mathcal{S}$ contains a binding $C\ a$, (i.e., $(C\ a) \in \mathcal{S}$), then we can naturally deduce from $\mathcal{S}$ that a is of class (type) C, denoted $a : C$. This rule is also a part of the formal system.

The above rules are easily extended to capture classes with extents as defined in ODMG 2.0. Unlike interfaces, classes in ODMG 2.0 are object types whose instances can be directly created. A class `C` with an extent `E` introduces an additional binding of `E` to a collection of objects of the type `C`. Two rules required to capture this idea are specified below:

$$\frac{\mathcal{S} \cup \{class\ C\ R\}\ schema,\ \ \ E \notin dom(\mathcal{S})}{\mathcal{S} \cup \{class\ C\ (extent\ E)\ R\}\ schema}$$

$$\frac{\mathcal{S}\ schema,\ \ \ class\ C\ (extent\ E)\ R \in \mathcal{S}}{\mathcal{S} \vdash E : Collection < C >}$$

9 Inheritance and subtyping

The ODMG Object Model identifies inheritance and subtyping. Thus `Employee` is a subclass and a subtype of `Person`. An instance of `Employee` may be substituted where an instance of `Person` is expected. This is a discipline from Eiffel and C++ and it causes some well-known problems [Coo89]. They do not appear in C++ because of a rigid type system. At the same time, this rigidity requires unsafe features which amount to bypassing the type system altogether. A more flexible type system requires a distinction between inheritance and subtyping ([CHC90], [BCM93]).

The collection of all object types (classes) is thus equipped with two orderings: `inherits_from`, denoted $\leq$, and `subtype_of`, denoted <:. If C1 $\leq$ C2 holds, then C1 is derived by inheritance from C2. However, if C1 <: C2 holds, then an instance (object) of C1 may be substituted where an instance of C2 is expected, with type safety guaranteed by static type checking.

The class `Object` is defined in such a way that it is both the top of the inheritance and the subtyping

orderings. Thus any class C derived by inheritance from `Object` ($C \leq Object$) will also be a subtype of `Object` ($C <: Object$). This explains why the type of the argument of the method `same_as` in the class `Object` is `Object`, just as in the ODMG Object Model. Another reason is that `same_as` is the object identity test and not the value equality test.

$$\frac{\mathcal{S} \vdash C \leq Object}{\mathcal{S} \vdash C <: Object}$$

9.1 Inheritance ordering

The inheritance ordering is reflexive and transitive. These properties are expressed in our type system as follows:

$$\frac{}{\mathcal{S} \vdash Object \leq Object}$$

$$\frac{\mathcal{S} \vdash C \leq Object}{\mathcal{S} \vdash C \leq C}$$

$$\frac{\mathcal{S} \vdash C3 \leq Object, \quad \mathcal{S} \vdash C2 \leq C3, \quad \mathcal{S} \vdash C1 \leq C2}{\mathcal{S} \vdash C1 \leq C3}$$

9.2 Subtyping ordering

The subtyping ordering is also reflexive and transitive:

$$\frac{\mathcal{S} \vdash C \leq Object}{\mathcal{S} \vdash C <: C}$$

$$\frac{\mathcal{S} \vdash C3 \leq Object, \ \mathcal{S} \vdash C2 <: C3, \ \mathcal{S} \vdash C1 <: C2}{\mathcal{S} \vdash C1 <: C3}$$

An important connection between the two orderings is expressed by the following rule:

$$\frac{\mathcal{S} \vdash C2 \leq Object, \ \mathcal{S} \vdash C1 <: C2}{\mathcal{S} \vdash C1 \leq Object}$$

If C1 $\leq$ C2 or C1 <: C2, then an instance of C1 may be viewed as an instance of C2. In fact, safety of a type system depends critically on whether this is allowed for inheritance or subtyping.

10 Structures

Contrary to Smalltalk and Eiffel, class interface is not the only type structuring construct in the ODMG Object Model. A more primitive one is structure (record), as in C++ and O2. Instances of structures are not objects but values. Structures are a natural starting place for the development of the formal rules for object types in the ODMG Object Model. The relationship $R \leq \{\}$; (R is a valid structure) is defined inductively as follows:

$$\frac{\mathcal{S} \vdash C \leq Object}{\mathcal{S} \vdash \{C\ m;\} \leq \{\};}$$

$$\frac{\mathcal{S} \vdash \{C1\ m1; C2\ m2; \ldots; Ck\ mk;\} \leq \{\};, \quad \mathcal{S} \vdash Cl \leq Object, \ \ ml \neq mi \ for\ i = 1, 2, \ldots, k}{\mathcal{S} \vdash \{C1\ m1; C2\ m2; \ldots; Ck\ mk; Cl\ ml;\} \leq \{\};}$$

Two more rules are needed to allow structures within structures. The only difference from the above rules is that the condition $C \leq Object$ is replaced by the condition $C \leq \{\};$, and the condition $Cl \leq Object$ is replaced by the condition $Cl \leq \{\};$. Similar remarks apply to some of the rules for structures given in the sequel.

The inheritance relationship among structures is defined simply as the subset relation:

$$\frac{\mathcal{S} \vdash R2 \leq \{\};, \mathcal{S} \vdash R1 \leq \{\};, \ \ R1 \subseteq R2}{\mathcal{S} \vdash R2 \leq R1}$$

This relationship is obviously reflexive and transitive.

The subtyping relationships among structures are much more subtle. Subtyping is certainly reflexive and transitive. Formal statements of these properties follow the pattern in the corresponding rules for object types. The general subtyping rule for structures is:

$$\frac{\begin{array}{c}\mathcal{S} \vdash Ci \leq Object \ \ for\ 1 \leq i \leq k, \\ \mathcal{S} \vdash C'i \leq Object \ \ for\ k < i \leq n, \\ \mathcal{S} \vdash C'i <: Ci \ \ for\ 1 \leq i \leq k\end{array}}{\begin{array}{l}\mathcal{S} \vdash \{C'1\ m1; \ldots; C'k\ mk; \ldots; C'n\ mn;\}; \\ \quad <: \{C1\ m1; \ldots; Ck\ mk;\};\end{array}}$$

Selecting a field of a structure is governed by the following familiar typing rule:

$$\frac{\mathcal{S} \vdash e : \{C1\ m1; \ldots; Ck\ mk;\};}{\mathcal{S} \vdash e.mi : Ci}$$

In order to define the type rules for class interfaces, we extend the type expressions for structures to allow structure expressions of the form $\{C1\ m1(C11, \ldots, C1p1); \ldots; Ck\ mk(Ck1, \ldots, Ckpk);\};$. These extended structure expressions allow signatures of methods. The first two rules given below define inheritance as it applies to extended structures.

$$\frac{\mathcal{S} \vdash Ci \leq Object,\ for\ 1 \leq i \leq k,\quad \mathcal{S} \vdash C \leq Object}{\mathcal{S} \vdash \{C\ m1(C1, C2, \ldots, Ck);\} \leq \{\};}$$

$$\frac{\begin{array}{c}\mathcal{S} \vdash \{C1\ m1(C11, \ldots, C1p1); \ldots; \\ Ck\ mk(Ck1, \ldots, Ckpk);\} \leq \{\}; \\ \mathcal{S} \vdash Cl \leq Object, \\ \mathcal{S} \vdash Cli \leq Object, for\ 1 \leq i \leq pl, \\ ml \neq mi\ for\ 1 \leq i \leq k\end{array}}{\begin{array}{c}\mathcal{S} \vdash \{C1\ m1(C11, \ldots, C1p1); \ldots \\ Ck\ mk(Ck1, \ldots, Ckpk); Cl\ ml(Cl1, \ldots, Clpl);\} \leq \\ \{\};\end{array}}$$

The rule that follows defines subtyping for extended structures:

$$\frac{\begin{array}{c}\mathcal{S} \vdash Ci \leq Object\ \ for\ 1 \leq i \leq k, \\ \mathcal{S} \vdash C'i \leq Object\ \ for\ k < i \leq n, \\ \mathcal{S} \vdash C'i <: Ci,\ \ for\ 1 \leq i \leq k, \\ \mathcal{S} \vdash C'ij \leq Object,\ \ for\ 1 \leq i \leq n,\ 1 \leq j \leq pn, \\ \mathcal{S} \vdash Cij <: C'ij\ \ for\ 1 \leq i \leq k,\ 1 \leq j \leq pk, \\ Cij \neq MyType\end{array}}{\begin{array}{c}\mathcal{S} \vdash \{C'1\ m1(C'11, \ldots, C'1p1); \ldots; \\ C'k\ mk(C'k1, \ldots, C'kpk); \ldots; C'n\ mn(C'n1, \ldots, \\ C'npn);\}; <: \{C1\ m1(C11, \ldots, \\ C1p1); \ldots; Ck\ mk(Ck1, \ldots, Ckpk);\};\end{array}}$$

11 Objects and messages

Object creation (as defined in OQL) is now specified by the following rule:

$$\frac{\mathcal{S} \vdash R \leq \{Ci\ fi;\};\ \ 1 \leq i \leq n,\quad \mathcal{S} \vdash ei : Ci, 1 \leq i \leq n,\quad class\ CR \in \mathcal{S}}{\mathcal{S} \vdash C(f1 : e1, f2 : e2, \ldots, fn : en) : C}$$

The condition $R \leq \{Ci\ fi;\};$ guarantees that R has an attribute fi of type Ci for $1 \leq i \leq n$.

As an interface typically refers to itself, the subtyping rule for object types is naturally recursive. In the ODMG Object Model $C2$ is a subtype of $C1$ only if the interface $C2$ is derived by inheritance from the interface $C1$. In our type system there are additional requirements, specified in the recursive rule that follows:

$$\frac{\begin{array}{c}\mathcal{S} \vdash interface\ C1\ R1 \in \mathcal{S}, \\ interface\ C2 : C1\ R2 \in \mathcal{S}, \\ \mathcal{S}, C2 <: C1 \vdash (R2 \cup R1) < MyType/C2 > <: \\ R1 < MyType/C1 >\end{array}}{\mathcal{S} \vdash C2 <: C1}$$

In the above rule $R < MyType/C >$ denotes the structure expression R in which the class C has been substituted for $MyType$.

The typing rule for messages is rather tricky. It will be given on the assumption of static type checking. The first thing that static type checking must ensure is that an object o executing a message `o.m(a1,a2,...,an)` is in fact equipped by an appropriate method. A further requirement is that the types of arguments `a1,a2,..,an` of the message `o.m(a1,a2,...,an)` must be subtypes of the corresponding argument types `C1,C2,...,Cn` of the method signature.

$$\frac{\begin{array}{c}\mathcal{S} \vdash interface\ C\ R \in \mathcal{S}, \\ \mathcal{S} \vdash R \leq \{Cm\ m(C1, C2, ..., Cn);\}; \\ \mathcal{S} \vdash o : Co, \quad \mathcal{S} \vdash Co \leq C, \\ \mathcal{S} \vdash ai : Cai,\ \mathcal{S} \vdash Cai <: Ci,\ 1 \leq i \leq n\end{array}}{\mathcal{S} \vdash o.m(a1, a2, ..., an) : Cm < MyType/Co >}$$

The conditions $R \leq \{Cm\ m(C1, C2, ..., Cn);\};$ and $Co \leq C$ ensure that the type Co of the object o has the required method signature. And finally, recall that $MyType$ denotes the type of the object executing the message. Thus the type of the result is not just Cm, but rather $Cm < MyType/Co >$, which is the result of substitution of Co for $MyType$ in Cm.

The above rule simplifies if we identify inheritance and subtyping (and thus there is only one ordering) by disallowing usage of `MyType`. Of course, the same applies to many of the previous and the remaining rules. This simplified, but also less flexible type

system, would correspond to the Java binding of the ODMG 2.0 Standard. It is in fact a particular case of the type system presented in this paper.

12 Parametric interfaces

The C++ binding of the ODMG Standard naturally relies heavily on parametric classes. This section provides only the basics for developing the formal rules for parametric interfaces.

The rule that follows allows interfaces with a single type parameter to be declared in a schema. A generalization to a finite number of parameters is trivial.

$$\frac{\mathcal{S}\ schema, C \notin dom(\mathcal{S}),\quad \mathcal{S}, T \leq Object, C<T> \leq Object \vdash R \leq \{\};}{\mathcal{S} \cup \{interface\ C<T>\ R\}\ schema}$$

A binding for a parametric class `C2` derived by inheritance from a parametric class `C1` is introduced by a statement of the form `interface C2<T>: C1<T> R2`, hence the rule:

$$\frac{\mathcal{S}\ schema,\quad interface\ C1<T>\ R1 \in \mathcal{S},\quad C2 \notin dom(\mathcal{S}),\quad \mathcal{S}, T \leq Object, C2<T> \leq Object \vdash R2 \cup R1 \leq \{\};}{\mathcal{S} \cup \{interface\ C2<T>:\ C1<T>\ R2\}\ schema}$$

Instantiation of a parametric interface to a specific one is now governed by the following rule:

$$\frac{\mathcal{S} \vdash A \leq Object,\quad interface\ C<T> R \in \mathcal{S}}{interface\ C<A> R<T/A> \in \mathcal{S}}$$

The above rule says that if a parametric interface `interface C<T>` is declared in a schema, then for any interface `A` in the schema, an interface `C<A> R<T/A>` *is also available in the schema.*

13 Queries

Space limitations make it impossible to present in this paper a complete set of typing rules for OQL. In this section we specify only the formal rules for a few characteristic forms of queries. Consider a *typical OQL query of the form:*

```
select projection
from e1 as x1, e2 as x2, ..., en as xn
where e'
```

The from clause determines the types of control variables `x1,x2,...,xn`. Each `ei` is required to be an expression of a collection type `Collection<Ci>`, hence the type of `xi` is `Ci`. In order to make queries on complex objects possible, each `e(i+1)` may be a component of `ei` for `i=1,...,n-1`. With the types of control variables determined this way, type checking of the qualification condition `e'` must produce `boolean`. In addition, if under the same assumption, the type of `projection` is `C`, the result of the query is of type `Bag<C>`.

$$\frac{\mathcal{S} \vdash e1 : Collection<C1>,\quad \mathcal{S} \cup \bigcup_{i=1}^{k}\{Ci\ xi\} \vdash e(k+1) : Collection<C(k+1)>, 1 \leq k < n,\quad \mathcal{S} \cup \bigcup_{i=1}^{n}\{Ci\ xi\} \vdash e' : boolean,\quad \mathcal{S} \cup \bigcup_{i=1}^{n}\{Ci\ xi\} \vdash projection : C,}{\mathcal{S} \vdash \text{select projection from e1 as x1, ..., en as xn where e': Bag}<C>}$$

If `select distinct` option is used in the above query, the result type is `Set<C>`. The typing rule for OQL queries of the form:

```
select projection
from e1 as x1, e2 as x2, ..., en as xn
where e'
order by e1',e2',...,em'
```

is just slightly more complex. Each `ej'` must be available in the result. This means that if the type of `ej'` is $C'j$, and C is the type of `projection`, we must have $C \leq C'j$. In addition, the result is ordered and thus its type is `List<C>`.

$$\frac{\mathcal{S} \vdash e1 : Collection<C1>,\quad \mathcal{S} \cup \bigcup_{i=1}^{k}\{Ci\ xi\} \vdash e(k+1) : Collection<C(k+1)>, 1 \leq k < n,\quad \mathcal{S} \cup \bigcup_{i=1}^{n}\{Ci\ xi\} \vdash e' : boolean,\quad \mathcal{S} \cup \bigcup_{i=1}^{n}\{Ci\ xi\} \vdash projection : C,\quad \mathcal{S} \cup \bigcup_{i=1}^{n}\{Ci\ xi\} \vdash ej : C'j,\quad \mathcal{S} \vdash C \leq C'j, 1 \leq j \leq m}{\mathcal{S} \vdash \text{select projection from e1 as x1, ..., en as xn where e' order by e1',e2',...,em': } List<C>}$$

Note how parametric polymorphism plays a crucial role in the above formal rules.

14 Reflection

The typing rules for type reflective facilities presented in this paper pose no particular problem. At the same time, we are not aware of a collection of formal typing rules that would capture all the subtleties involved in the general linguistic reflective facilities of the kind presented in [SSS92].

Given interfaces C1 and C2, we define their meet, denoted C1*C2, as an interface such that C1*C2 $\leq$ C1, C1*C2 $\leq$ C2. Furthermore, if C is an interface such that both C $\leq$ C1 and C $\leq$ C2 hold, then C $\leq$ C1*C2.

$$\frac{\mathcal{S} \vdash C1 \leq Object, \quad \mathcal{S} \vdash C2 \leq Object}{\mathcal{S} \vdash C1 * C2 \leq Object}$$

$$\frac{\begin{array}{c}\mathcal{S} \vdash C1 \leq Object, \quad \mathcal{S} \vdash C2 \leq Object, \\ \mathcal{S} \vdash C \leq C1, \quad \mathcal{S} \vdash C \leq C2\end{array}}{\mathcal{S} \vdash C \leq C1 * C2}$$

The definition of join of types is symmetric and it is thus omitted. Now we can specify the typing rule for the `natural_join` operator:

$$\frac{\begin{array}{c}\mathcal{S}, T \leq Object \vdash a : Table < T >, \\ \mathcal{S}, R \leq Object \vdash b : Table < R >\end{array}}{\mathcal{S} \vdash a.natural\_join(b) : Table < T * R >}$$

It is easy to see that the meet `C1*C2`, with `interface C1 R1` and `interface C2 R2`, is well-defined if $R1 \cup R2 \leq \{\}$;. Likewise, the join `C1+C2` is well defined if $R1 \cap R2 \leq \{\}$;.

15 Conclusions

Release 1.2 of the ODMG Object Model contains major problems revealed in this paper. ODMG 2.0 attempts to avoid some of them, but makes only a modest step in that direction. In fact, the core of our criticism applies to ODMG 2.0 as well. The goal of our critical analysis is to help in making future releases better. Should that ever actually happen, the mission of this paper would have been accomplished.

In particular, it is hard to imagine a sound type system for the ODMG Object Model. No matter how sophisticated a type checker may be, there will always be situations in which the checker will not be able to decide whether a particular construct is type correct or not. One consequence of this conclusion is that extensive run-time checks are required. These checks make transactions less efficient and prone to failure at run time. It is in our opinion unacceptable to be forced to invoke expensive recovery procedures just because of run-time type errors that could have been detected at compile time.

Our analysis shows that one can define not only a single ODMG Object Model, but a family of models of the increasing level of sophistication. All the models in the family can be formally defined. These particular formal definitions appear as particular cases of the formal system presented in this paper. Vendors could start by implementing the basic model, gradually moving to the more sophisticated ones as extensions of the basic model. This approach eliminates the confusion that exists in the current model, and makes clear what level of compliance and sophistication a particular vendor is actually providing.

The most basic model is obtained by identifying any with `Object`. In addition, the Model should be equipped with orthogonal persistence. Consequently, persistence capable classes should be eliminated from the Model. In this model inheritance is identified with subtyping, just as in the ODMG Object Model.

This model corresponds to the type system underlying Java. In fact, the ODMG 2.0 Java binding corresponds to this model, except that it has a model of persistence which is not orthogonal. This model can be mapped to the type system of C++ simply because the type system of C++ is more powerful. But dealing with orthogonal persistence in C++ would remain an issue. The model is also as close as one can get to Smalltalk.

The main problem with this model is that it is too rigid. Extensive downcasting with dynamic checks is required, and even that cannot solve all the problems, as illustrated by the problems of cloning objects. But at least this way the confusion that exists at present in the ODMG Object Model is eliminated, and the model is extended with orthogonal persistence.

A strictly more sophisticated and more flexible model is obtained from the basic one by allowing limited redefinitions of method signatures using the

type variable MyType. This produces a strictly more powerful type system in which a distinction must be made between inheritance and subtyping. The expressibility of the model is significantly increased at the expense of a strictly more sophisticated type technology. If substitutability is based on subtyping, type safety can be guaranteed by static type checking. However, if substitutability is based on inheritance, as in Eiffel, a more sophisticated implementation technology based on multiple dispatch is required.

In any case a formal definition of the ODMG Object Model is required. We have presented a collection of formal rules that have not been defined as of yet for the ODMG Object Model. Without such a formal system the recommended ODMG bindings are open to a wide range of different and sometimes contradicting interpretations. If an attempt had been made earlier to formalize such rules, the problems reported in this paper would have been discovered.

The importance of parametric polymorphism in database-oriented object models is obvious from the collection classes predefined in the ODMG Object Model. The confusion that currently exists in the ODMG Object Model regarding parametric polymorphism must be eliminated. Either the ODMG Object Model supports it, or it does not. If not, then `Object` must be used consistently instead of a type parameter. A clean solution which produces a third, strictly more sophisticated model, is to provide full support for parametric polymorphism.

Whether this family of ODMG Object Models should be extended further with more advanced features is probably a topic for debate. We have demonstrated that even a more sophisticated type system is required if we want to type properly essential components of any database architecture. Such a type system would support bounded quantification (as in Eiffel), and matching or F-bounded polymorphism.

Finally, hidden subtleties in the ODMG Object Model require all of the above features, as well as type reflection. This is the most sophisticated model in this hierarchy. Admittedly, the ODMG Standard may not want to go this far. But the fact that the ODMG Object Model is a database object-oriented model requires these sophisticated features, regardless of whether the ODMG Standard wants to be explicit about them or not. In fact, Java has limited reflective facilities related to the presence of the `Class` class in the language. And ODMG 2.0 has elaborate meta-level architecture, which is yet to become the focus of our investigations.

Acknowledgment

I would like to thank Kennis Lai for her comments on an earlier version of this paper, and Svetlana Kouznetsova and Jose Solorzano for their comments on the final version.

References

[AC96] M. Abadi and L. Cardelli, On Subtyping and Matching, Proceedings of ECOOP'96, *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 1098, pp. 145-167, 1996.

[AG89] R. Agrawal and N. Gehani, Ode: Object Database and Environment; Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++, Proceedings of the 2nd International Workshop on Database Programming Languages, Portland, Oregon, 1989.

[Ala94] S. Alagić, F-bounded Polymorphism for Database Programming Languages, Proceedings of the 2nd East-West Database Workshop, *Workshops in Computing*, Springer-Verlag, pp. 125-137, 1994.

[Ala97] S. Alagić, Constrained Matching is Type Safe, Proceedings of the 6th Int. Database Programming Language Workshop, to appear, 1997.

[ABD89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. and S. Zdonik: The Object-Oriented Database System Manifesto, Proceedings of the First Object-Oriented and Deductive Database Conference, Kyoto, 1989.

[ADJ96] M. Atkinson, L. Daynes, M.J. Jordan, T. Printezis and S. Spence, An Orthogonally Persistent Java^{TM}. ACM SIGMOD Record, No. 4, Vol. 25, pp. 68-75, 1996.

[AM95] M. Atkinson and R. Morrison. Orthogonally Persistent Object Systems, *VLDB Journal*, Vol. 4, pp. 319-401, 1995.

[BC96] J. Boyland and G. Castagna, Type-Safe Compilation of Covariant Specialization: a Practical Case, Proceedings of ECOOP '96 Conference, *Lecture Notes in Computer Science* Vol. 1098, pp. 3-25, Springer-Verlag, 1996.

[BCM93] K. Bruce, J. Crabtree, T. P. Murtagh and R. van Gent, A. Dimock and R. Muller, Safe and Decidable Type Checking in an Object-Oriented Language, Proceedings of the OOPSLA Conference, pp. 29-46, 1993.

[BSG95] K. Bruce, A. Schuett, and R. van Gent, PolyTOIL: a Type-Safe Polymorphic Object-Oriented Language, Proceedings of ECOOP '95, *Lecture Notes in Computer Science* Vol. 952, pp. 27-51, Springer-Verlag, 1995.

[CCH89] P. Canning, W. Cook, W. Hill, W. Olthoff and J.C. Mitchell, F-Bounded Polymorphism for Object-Oriented Programming, Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, pp. 273-280, 1989.

[Car88] L. Cardelli, Types for Data Oriented Languages, In: J.W. Schmidt, S. Ceri and M. Missikoff (Eds), Advances in Database Technology - EDBT '88, *Lecture Notes in Computer Science*, 303, Springer-Verlag, Berlin, pp. 1-15, 1988.

[Cat96] R. G. G. Cattell (ed), *The Object Database Standard: ODMG-93, Release 1.2*, Morgan Kaufmann, 1996.

[CBB97] R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, D. Wade, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997.

[CHC90] W. R. Cook, W. L. Hill and P. S. Canning, Inheritance is not Subtyping, In: Proceedings of the Conference on Principles of Programming Languages, ACM Press, 1990, pp. 125-135.

[Coo89] W. R. Cook, A Proposal for Making Eiffel Type Safe, *The Computer Journal*, Vol. 32, no. 4, 1989, pp. 305-311.

[Feb89] J. Ferber, Computational Reflection in Class Based Object-Oriented Languages, Proceedings of the OOPSLA Conference, pp. 317–326, 1989.

[GJS96] J. Gosling, B. Joy and G. Steele, *The Java^{TM} Language Specification*, Addison-Wesley, 1996.

[MBC88] R. Morrison, F. Brown, R. Connor, A. Dearle, The Napier88 Reference Manual, Universities of Glasgow and St. Andrews, PPRR-77-89, 1989.

[Mey92] B. Meyer, Eiffel: *The Language*, Prentice-Hall, 1992.

[RCS93] J. E. Richardson, M. J. Carey and D. Schuh, The Design of the E Programming Language, *ACM Transactions on Programming Languages and Systems*, Vol. 15, pp. 494-534, 1993.

[OPS95] M. T. Ozsu, R. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz, TIGUKAT: A Uniform Behavioral Objectbase Management System, *VLDB Journal*, Vol. 4, pp. 445-492, 1995.

[OB89] A. Ohori and P. Buneman, Static Type Inference for Parametric Classes, Proceedings of the OOPSLA Conference, pp. 445-456, 1989.

[OBB89] A. Ohori, P. Buneman and V. Breazu-Tannen, Database Programming in Machiavelli: a Polymorphic Language with Static Type Inference, Proceedings of the ACM SIGMOD Conference, pp. 46-57, 1989.

[O296] ODMG C++ Reference Manual, O2 Technology, 1996.

[PsA87] The Ps-Algol Reference Manual, Persistent Programming Research Report 12, Universities of Glasgow and St. Andrews, 1987.

[SFS90] D. Stemple, L. Fegaras, T. Sheard and A. Socorro, Exceeding the Limits of Polymorphism in Database Programming Languages, In: F. Bancilhon and C. Thanos (Eds), Advances in Database Technology - EDBT '90, *Lecture Notes in Computer Science*, 416, pp. 269-285, Springer-Verlag, 1990.

[SSS92] D. Stemple, R. B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M. Atkinson, and S. Alagić, Type-Safe Linguistic Reflection: A Generator Technology, ESPRIT Research Report CS/92/6, Department of Mathematical and Computational Sciences, University of St. Andrews, 1992.