OBJECT-ORIENTED DATABASE PROGRAMMING LANGUAGES

Invited Paper

Antonio Albano and Giorgio Ghelli

Dipartimento di Informatica
Università di Pisa
Corso Italia, 40 – 56100 Pisa, Italy

ABSTRACT

Database programming languages are languages whose main concern is in the structuring and handling of databases. They were developed to compensate for traditional DBMSs, which were limited both in the modeling mechanisms, which favour the efficient use of secondary memory at the expense of expressiveness, and in the integration of the database modeling mechanisms with those of programming languages. Recently several database programming languages have been defined which are based on the object-oriented paradigm. An overview of the main characteristics of this approach and of some of these languages is presented. The conclusion identifies areas where further research is required.

1 INTRODUCTION

Over the last 25 years the features of DBMSs have evolved to improve both the management of persistent data and the development of applications that have access to them. Basically, there have been two major generations of DBMSs. The network and hierarchical DBMSs were the first generation (1970s). They were the first to provide the fundamental features for storing and retrieving permanent data efficiently, with transaction management, concurrency control, and a procedural oriented data manipulation language. The relational DBMSs were the second generation (1980s). They offer non-procedural data manipulation languages, data independence, ad hoc interactive query facilities, and, most importantly, a well understood and well defined data model.

As the importance of DBMSs was first recognized in the area of business applications, the implementors of DBMSs were biased toward the specific requirements of this area: efficient use of secondary memory and fast access to large sets of interconnected data with a relatively simple structure. More recently, because technology costs have come down, performance has improved, and new facilities have been introduced to support the development of applications; the class of applications adopting DBMSs has been increasing and spreading from business applications to applications using complex data of different types, such as those in office information systems, CAD/CAM, CASE and knowledge base management. Commercially available DBMSs, based on hierarchical, network or relational data models, are not suitable for these new types of applications, mainly for the following reasons:

- The data model abstraction mechanisms favour the efficient use of secondary memory at the expense of expressiveness. As the information to be represented becomes increasingly complex, there is a growing need for more expressive data structuring mechanisms to model the structure of the information directly and naturally. Over-simplified structuring mechanisms facilitate the work of the DBMS but complicate the solution of the problem, to the detriment of the comprehensibility of the representation and of the efficiency of the operations.
- The schema cannot describe facts which can be derived procedurally from others, or facts which can be manipulated on the basis of operators explicitly defined by the system designer using a mechanism such as data abstraction. Procedural aspects must be treated in application programs.
- The second generation of DBMSs have been quite effective for interactive query languages but not for programming environments to develop interactive applications. Applications using the data base are programmed by languages which host the data model operators, using ad hoc mechanisms to exchange information between a program and the DBMS. Consequently, the programming environment is unlikely to be suitable for the growing complexity of the applications, due to the scarce integration of the data model abstraction mechanisms with those of the programming language.

These limits have been widely recognized in literature and many proposals have been made to overcome them. From the latter half of the 1970s, the attention was initially focussed on the integration of the relational data model abstraction mechanisms in the type system of a programming language, to construct an integrated programming environment to develop database applications. The term database programming language was used to refer to these approaches. Examples of this approach are languages such as Pascal/R [Schmidt 77], and more recently DBPL [Schmidt 90].

At the end of the 1970s new database programming languages were proposed to support a more expressive data model, named semantic data model, to overcome the modeling limitations of the traditional data models. This resulted in the development of so-called conceptual languages such as ADAPLEX [Smith 81], Taxis [Mylopoulos 80] and Galileo [Albano 85]. Another approach to integrate data models and type systems was persistent languages, where all data values, whatever their type, are allowed the full range of persistence [Atkinson 83]. An excellent survey of these approaches is given in [Atkinson 87].

At present the main research issue in this area is the design of a database programming language using an approach based on the object-oriented programming paradigm, experimented initially in Simula-67 and Smalltalk, and used with success in areas such as AI and user interface development [Dahl 66] [Goldberg 83]. The object-oriented programming paradigm would be very promising for developing a third generation of DBMSs because both the problem of the expressivity of the data model and the problem of integrating procedural and data modeling aspects could be tackled.

This work was carried on with the partial support of Selenia, project "Very Large Knowledge Bases", of E.E.C. Esprit Basic Research Action 3070 FIDE, and C.N.R. Project Sistemi informatici e calcolo parallelo.

CH3001-5/91/0000/0726/$01.00 © 1991 IEEE
The remainder of this paper presents these attempts to integrate
the object-oriented programming paradigm and database
programming languages. Section 2 focuses on the minimal set of
features of object-oriented databases. Section 3 describes the
approaches used in designing the Galileo, O2 and ORION database
languages. Finally, Section 4 provides directions for future research
in object-oriented database programming languages (OODBPs).

The focus of this paper is on the language features. Other
important areas of object-oriented DBMSs, such as implementation,
query optimization techniques, transaction and concurrency
management, will not be considered. The discussion will be
informal on the whole, and should not be seen as a complete
description of the languages discussed or the research performed or
currently in process.

2 FEATURES OF OBJECT-ORIENTED DATABASES

Object-oriented databases is currently a very active area of
research in universities, research centers, and industrial research
laboratories. However, at present a data model for object-oriented
databases does not have an universally accepted definition, as
happened with the relational data model, defined formally by Codd,
although there is a general agreement on the minimum set of
features, and this will be presented in this section [Atkinson 89]
[Bancilhon 88a] [Banerjee 87] [Dittrich 90] [Wegner 87a, 87b]
[Zdonik 90].

Objects
An object is a software entity which has an internal state equipped
with a set of local operations (methods), to manipulate that state;
methods define the object behavior. The request to an object to
execute an operation is called a message, to which the object
replies. The structure of an object state is modeled by a set of
variables (or attributes) which can have values of arbitrary
complexity, including other objects which become components of
the object. An object encapsulates its state, when the state can only
be accessed and modified through the operations associated with
that object.

Objects are denotable and expressible as "first-class" values of the
language, i.e. they can be assigned as values of variables, and used
as data structure components, as parameters or as results of
functions. Since objects can be used as components of other objects
and shared, when objects are updated their modification is reflected
in all the other objects in which they appear as components.

Finally, each object is distinct from all other objects and values,
and has an identity that persists over time, despite any changes to
the value of its state. For instance, the object representing the person
John is different from any other object representing another person,
but will remain the same even if his address or some other attribute
changes. A new identity is created any time a new object is created.
Consequently, two objects which have been created in two different
moments are always different, whatever their attribute values.

Types
A type specifies some properties of all the expressions with that
type, and in particular it specifies the structure of the values denoted
by an expression.

The type of an object is called an object type. An object type
specifies at least the signature of the objects of that type, i.e. the
name, the type of the parameters and the type of the result of all
the methods which can be applied to the objects of that type. Usually,
an object type also specifies the implementation of these methods
and the internal structure of the state of the objects.

Objects can be created only when their object type has been
defined. In object-oriented jargon the definition of an object type
which includes the definition of its state and the implementation of
its methods, is called a class, but in this paper the term "class" is
only used to denote collections of objects.

Type and Inheritance hierarchies (Iat)
In object-oriented languages the signature of an object type, its
implementation and the structure of the objects in that type can be
defined by inheritance, i.e. by specifying how the new
signature/implementation/state structure extends the old one. Since
the signature of an object type is a set of pairs 
\((\text{name}, \text{type})\), it can
be extended either by adding new pairs, or by modifying the type
associated with that name. This also applies for the implementation
of methods and for the structure of the state.

On the other hand, subtyping is a reflexive antisymmetric and
transitive relation defined on the set of all the types, including the
object type, such that if \(T\) is a subtype of \(U\) (written \(T \supseteq U\))
any expression with type \(T\) also has type \(U\), and can then be used
in any context where a value of type \(U\) is expected. This property is
called context inheritance.

Inheritance can be strict, when properties can only be redefined in
a controlled way, so that an object type defined by inheritance is a
subtype of the type from which it inherits; or it can be non-strict,
when properties can be redefined freely.

In imperative languages, when the same operator is defined on a
subtype and on a supertype, such as + on integers and reals, the
meaning of each instance of that operator is established at compile
time, depending on the type associated with the parameters by the
compiler. In object-oriented languages, however, when the
implementation of a method is redefined (overridden), the meaning
of an invocation of that method is not decided on the basis of the
compile-time type of the object but at run time instead, on the basis
of the object type used to create the object (late binding). Despite
its apparently technical nature, late binding is a fundamental feature
of the object-oriented paradigm, and is essential to reach the high
level of extendibility and reusability of object-oriented code.

An object type can be defined by inheritance starting from only
one other object type (simple inheritance) or from several other
ones (multiple inheritance). In the latter case, if name clashes are
not forbidden, a conflict resolution mechanism must be provided.
An important consequence of inheritance is that applications can
be developed incrementally by specializing object type definitions,
as a result of evolution in time, or of a design methodology which
takes advantage of common features in types.

The inheritance mechanism was initially experimented in Simula
67 and Smalltalk, and later in languages based on Lisp, C, Pascal,
and in conceptual languages for databases such as ADAPLEX,
Galileo and Taxis [Smith 81] [Albano 85] [Mylopoulos 80]. It is
currently recognized as the most appropriate proposal for software
extendibility, i.e. the possibility of defining new objects by
combining and extending pre-existing definitions [Meyer 88].

Software extendibility is extremely important in data base
applications, since data bases, by their very nature, are used for
many years after installation and have to be continually modified to
deal with new requirements. It is thus extremely useful to be able to
extend the data base schema without having to interrupt the
applications in progress. Inheritance makes it possible to specialize
the behaviour of the objects incrementally; it seems to be the most
natural mechanism for certain types of modifications to the schema.
Objects, types, and type hierarchies are essential features for any object-oriented language, but the following features must also be taken in consideration for an object-oriented database language.

Persistence
Data persistency is the period of time during which data exists and can be used. In traditional languages, the life-time of an object depends on the life-time of the program using it, whereas object-oriented languages only provide save and restore mechanisms for the entire state of an operating session. In applications using data bases, persistent objects must be handled, i.e. objects which, once created, continue to exist independently of the program which created them and can be used in other applications until they are explicitly deleted.

Rich type system
A rich type system is needed in order to model the entities of the real world directly (also called the domain of discourse), so that there is a one-to-one correspondence between entities and their representation as objects. The type system should be strong, so that any typed expression can never cause run time failures, since strong type systems allow that level of software reliability which is required in database applications to be reached.

Integrity constraints
Database languages also need a mechanism to declare integrity constraints specifications, i.e., static constraints which impose restrictions on the admissible values of the state of an object, and dynamic constraints which impose restrictions on the ways in which the state can evolve. These constraints must be automatically maintained by the system.

Classes
The word "class" is often used with the meaning of "type" in object-oriented languages, but here it will be used with a different meaning. Whereas an object type defines a set of possible values, a class is a modifiable collection of objects of a certain type which have been actually created. Often a class definition has two different roles: it introduces the definition of the type of its elements (intensional aspect), and supplies a name to denote the set of the objects of that type currently in the model (extensional aspect). In this case the system often enforces the constraint that all the objects created with the type associated with that class belong to that class.

Classes of object models sets of entities of the observed world, while relationships between such entities are represented using the aggregation mechanism: by having a method in each object which returns the associated object / objects.

Data retrieval operations are performed beginning with classes.

Class hierarchies (SubclassOf)
This is an antisymmetric, reflexive and transitive relation in the set of classes, such that if (C1 SubclassOf C2), then C1 is said to be a subclass of C2 and the following properties hold:

- the type of the elements of C1 is a subtype of the type of the elements of C2 (intensional constraint);
- the elements of C1 are a subset of the elements of C2 (extensional constraint).

For example, if we are interested both in Persons and Females, we have to model two different and essential facts: (a) the type Female is a subtype of the type Person, because a value of type Female can be used wherever a value of type Person is required; (b) the set of all actual Females is a subset of all actual Persons (i.e. the class Females is a subclass of the class Persons) (Fig. 1).

![Figure 1: Subtypes and subclasses](image1)

A subclass can be defined from a single superclass or from several superclasses. Moreover, subclasses of the same superclass can be defined in three different ways: a) the sets of the elements of subclasses may overlap (subsets); b) the sets of the elements of subclasses are disjoint (disjoint subsets); and c) the subclasses are disjoint and their union is equal to the superclass (partition) (Fig. 2).

![Figure 2: Kinds of subclasses](image2)

Usually there are two ways to populate subclasses:

- by creating elements with an appropriate constructor; these elements will also appear as elements of its superclasses, because of the extensional constraint of the subclass relation.
- by moving objects from a superclass into the subclass. Thus, objects can change the most specific class which they belong to during their life-time. For instance, a person can belong to the subclass of students, then employees, and finally be just a person again.

Because of the semantics of the extensional constraint of the subclass relation, when an object is removed from a class, it is also removed from its subclasses; but when it is removed from a subclass, it will remain in the superclasses.

Views and metadata
The database should be accessible by different users depending on their personalized views, and the data base description should be accessible by a meta database.

3 APPROACHES TO OODBPL
The management of an object-oriented data base, which has the
above-listed features, entails new linguistic solutions to identify suitable data modeling abstraction mechanisms, and original solutions for the problems of data access and storage.

The proposals of recent years to develop object-oriented databases programming languages aim either at designing database programming languages from the scratch, or at extending languages such as Smalltalk, C++, or Common Lisp with mechanisms to model data bases; in particular, persistency, concurrency and transactions.

An important feature of some object-oriented databases programming language is strong typing, statically checked. A strongly typed, statically checked language is essential in program engineering for languages used in the final encoding of applications; particularly for complex applications, which evolve in time, as happens with database applications. Examples of languages which are not strongly typed are those supported by O2 [Lécuit 89], Statice [Symbolics 88], Orion [Banerjee 87], Gemstone [GemStone 86], Vbase [Ontologic 87] and Ontos [Ontologic 89], while an example of a strongly typed, statically checked language is Galileo, which does not however support completely the object-oriented paradigm.

3.1 A Brief Analysis of three OODBPLs: Galileo, O2, ORION

This section briefly compares the features offered by the object-oriented database programming languages Galileo, O2, and ORION to model the following aspects of database applications: object type, type hierarchies, classes, classes hierarchies. These languages were chosen to show different language design philosophies. Figures 3, 4 and 5 show some simple examples to illustrate the "flavor" of object and class definitions in the three languages.

use rec type % an object definition %
  Circle <->
    (Center: (x: real and y: real))
    and Radius: real
    and Area: (x: null): real:=
      use r := Radius of this
      in times(times(r, r), 3.14159) ;

use rec type % a class of objects definition %
  Person <->
    (TaxCode: string
    and Name: string
    and BirthYear: int
    and Age: 0r Year of CurrentDate() - BirthYear of this
    and Addresses: var seq Address
    and IntroduceYourself (x: null): null:=
      assert BirthYear of this <= Year of CurrentDate()
      elsefai1 Year of CurrentDate() must not be greater than this year!)
    key(TaxCode);

use rec type % a subclass definition %
  Student <->
    (is Person
    and StudentNumber: string and YearOfAdmission: int
    and IntroduceYourself (x: null): null:=
      assert Year of CurrentDate() - BirthYear of this > 17
      elseif Age must be greater than 17!)
    key(StudentNumber);

Figure 3: An example in Galileo

(make-class 'Person
  :superclasses nil
  :attributes ((TaxCode :domain string )
    (Name :domain string)
    (BirthYear :domain integer)
    (Addresses :domain set-of Address))
  :methods '((IntroduceYourself nil)) )

(make-class 'Student
  :superclasses 'Person
  :attributes '((StudentNumber :domain string ))
  :methods 'IntroduceYourself() )

Figure 4: An example in ORION

class Circle
  type tuple (Center: tuple (x: real, y: real)
    Radius: float )
  method Area(): real;

class Person with extension
  type tuple (TaxCode: string,
    Name: string,
    BirthDay: integer,
    Addresses: set(Address))
  method Init(): Person,
    Delete(),
    Age(): integer,
    IntroduceYourself();

class Student with extension
  inherits Person
  type tuple (StudentNumber : string,
    YearOfAdmission: integer)
  method Init(): Student
    IntroduceYourself();

Figure 5: An example in O2

Galileo, a prototype developed at the University of Pisa, was designed to fully integrate general programming facilities with those needed for programming database applications. An existing programming language, ML, was extended with new orthogonal features, specifically designed for database applications. Galileo was the first example of a complete interactive, strongly typed database functional language with state and side effects, which offers inclusion polymorphism: a functional type system was enriched for the first time with a subtype relation, to support the class hierarchies of a semantic data model [Albano 85] [Albano 88].

ORION, a prototype developed at MCC, Austin, Texas, extends Common LISP with constructs to model object-oriented databases. The ORION project dealt with several interesting object-oriented databases issues such as: schema evolution, the management of complex objects and object versions, query optimization, transactions with complex object locking, and the management of multimedia data. The project has finished but the prototype will be commercialized [Banerjee 87] [Kim 90].

O2, a prototype developed at Alair, Paris, adopts a multi-language approach by providing a set of constructs to model object-
oriented databases which are embedded in C and Basic; the resulting languages are called CO2 and Basic02, respectively. A company has been created to develop a commercial product [Lecluse 89].

The three languages all support an object-oriented data model, but there are differences in the modeling possibilities.

Object. The three languages all support the notion of object with identity, without state encapsulation. In Galileo and O2 all the language values can be either temporary or persistent, as persistence is an orthogonal property of the type system. In ORION only objects can become persistent, and the management of versions is provided.

Object type. The three languages all support a construct to define object types, with associated methods. Only ORION has the restriction that the state of an object can have components of the following types only: elementary, object, and set of objects.

The other languages have a collection of type constructors (e.g., tuples, set) that can be used without restrictions to define a component of the object state. Only the Galileo type system provides a constructor to distinguish constant values from modifiable ones, and this has important consequences on the possibility of statically typechecking definitions and applications. In the other languages this control on the modifiability of a component of the object state can be enforced by methods or by the authorization mechanism.

Only ORION allows the definition of composite objects, i.e. objects that have other objects as components with the possibility of enforcing (a) the constraint that an object is a part of only one object, and (b) that the existence of an object depends on the existence of its parent object.

Methods are defined differently in the three languages. In Galileo object types are kinds of abstract data types with a record structure, with fields of any type, and with functional fields to model methods, which can refer recursively to the whole object (self recursion). In ORION and O2 only the method signature is specified in an object type definition, since the body of a method is specified separately as a Common LISP or CO2 function with the object type to which the method belongs as the type of the first parameter.

Classes. Only Galileo distinguishes between the notions of an object type and a class, the set of all objects of a certain type in the database. In ORION an object type definition entails automatically defining a variable with the same name to denote all the objects of that type; in O2 the approach is similar when an object type is defined with the option 'with extension'.

In all these languages, associations between objects are only modeled by aggregation, i.e. by the definition of objects which have other objects as components.

Type hierarchies. In all these languages, object types can be organized into hierarchies and the benefits of the inheritance mechanism can be exploited. However only the Galileo type system permits static typechecking of definitions in order to check that a subtype is well defined when a component or a method is predefined.

Class hierarchies. This mechanism has an extensional and intensional aspect. This is well illustrated in Galileo, where they are modeled with two distinct mechanisms: subclasses to deal with the extensional aspect, and type hierarchies for the intensional aspect. The extension of a subclass is a subset of the extension of the superclass. A subclass can be defined from a single superclass (subclasses of the same class are not disjoint) and can be populated by creating new elements, and also by moving objects from a superclass into the subclass. A predicate tests whether an element of a superclass is also a member of a subclass.

O2 does not enforce the extensional constraint; ORION provides disjoint subclasses, except when a class is defined by multiple inheritance, and subclasses cannot be populated by moving objects from a superclass into the subclass. A construct accesses the element of a class without considering those elements which also belong to a subclass.

4 CONCLUDING REMARKS AND RESEARCH DIRECTIONS

The fundamental features of object-oriented database languages have been presented. This new approach to the design of database languages should prove very promising in improving the application development process. These improvements come from the richness of the data model, and from the data abstraction and inheritance mechanisms provided by the object paradigm. Examples of object-oriented DBMSs are also beginning to appear on the market [GemStone 86] [Andrews 87], but there are many problems which still require more investigation.

The most important research issues are:

- Performance related studies;
- Designing a query model and query languages;
- Methodologies for schema and application design;
- Defining of formal basis for OODBPLs;
- Type theories for OODBPLs;
- Enhancements to the OO data model;
- Design of languages and systems allowing schema evolution;
- Studies on new features of transaction and concurrency mechanisms.
- Design of OODBPLs which fully support the database related operations;

Much of this research is still in the preliminary stage; much work still has to be done on the basic problems of performance and formal foundation. In the next subsections we comment on the research fields listed above.

Performance

The technology of OODBPLs will only become widely accepted if it reaches the same performance levels offered by current technology, without compromising the expressive power. OODBPLs pose many new problems in terms of performance, and many of the basic techniques used to improve the efficiency of traditional DBMSs cannot be immediately extended to these systems, due to the complex structure of objects and to the encapsulation mechanisms of the OO languages.

For example, in traditional DBMSs, it is possible: (a) to store the tuples of one class in contiguous pages; (b) to group tuples of different classes on the same page to speed up the retrieval of related data; and (c) to maintain data structures allowing associative access to data. In object-oriented DBMSs, storing objects in even class already creates problems, due to the complex and variably sized structure of objects. Grouping related objects creates new problems since in the object-oriented data model objects can be mutually related through structures of any complexity, and associations can also be mediated by methods. In object-oriented systems there is also the problem of the physical storage of objects which belong both to a subclass and a superclass.

Depending on the usage, it may be more convenient to store the whole object in the superclass, the whole object in the subclass, or to split it between the two classes. In this field, an important problem, which is still unsolved, is the development of a formal setting to compare different storage strategies, or at least the development of reasonable and standard benchmarks.

Finally, the usage of associative access structures in object-oriented databases is more complex than in relational databases for...
Schemas is essential for the success of this approach. This research objects and classes. By viewing them during queries, as non-flat relations, this kind of approach suffers from the problems of the object-oriented paradigm. The latter only exist after they have been created, and they cannot have the same identity. The relational algebra operators are closed, since they perform side-effects. Furthermore, a good query language should be based on operators which satisfy the closure property, i.e. operators which transform entities into entities of the same nature. The relational algebra operators are closed, since they transform relations in relations. By analogy, an object-oriented query language should transform classes of objects into classes of objects, but it is not clear how object identity could be managed. The identities of the resulting objects should be a subset of the identities of the queried classes; e.g., a query asking for all the Persons named John should return all the existing objects of class Persons with their identity, without creating new ones. On the other hand a relational-like query can return objects which may have more methods than the original ones, i.e. by performing a join operation, and can combine and split objects, so that the resulting objects cannot be the "same" original objects, i.e. they cannot have the same identity.

In this case, the basic problem is that, while the relational tuples are just "mathematical entities" which do not need to be created, objects are entities which have both a state and an identity. The latter only exist after they have been created, and they should only be created to model the creation of a new entity in the domain of discourse, and not to answer queries.

One approach to this problem is to define a mathematical view of objects and classes, by viewing them during queries, as non-flat tuples and sets of non-flat tuples respectively. Then queries can be performed by extended relational algebras, operating on these non-flat relations. This kind of approach suffers from the problems which are associated with views — some important information such as object identity is lost — and must be carefully studied to prevent problems due to the fact that data is implicitly converted from the object/class to the tuple/set of tuples format.

Methodologies for schema and application design
Defining methodologies to develop OODBPL applications and schemas is essential for the success of this approach. This research field could inherit much from the methodologies studied for entity relationship schema designs and from the development of object-oriented applications, but still there is much work to be done.

The most important problems come from the fact that, in OODBPLs, procedural information is present in the schema; this greater expressive power cannot be dealt with by traditional methodologies. The development of these methodologies would greatly benefit from the development of better languages. The following linguistic features would be helpful:

- The possibility to separate the definition of the signature of an object type from its implementation would provide a better support of methodologies where interface and implementation design are separated.
- An expressive type system where most aspects of the domain of discourse could be directly modelled would avoid representation problems.
- A strong type system which would directly implement the systematic error prevention activities which are often encoded in methodologies. It would be useful to be able to describe in the language the kind of information which must be described in the documentation of the code, such as that a variable is mutable and immutable, that a function could perform side effects, etc.

Defining a formal basis for OODBPLs
The lack of a formal understanding of the basic features of the object-oriented paradigm is the basic problems of this approach. The object-oriented paradigm is based on notions such as objects, identity, messages, encapsulation and inheritance which are defined through analogies with the "real world", rather than formally. A general agreement about the innumerable interactions between all these different mechanisms is therefore impossible. A formal description of the object-oriented mechanisms through a minimal set of well-defined elementary notions would be needed for the following reasons:

- To define simpler programming and query languages, without sacrificing expressivity;
- To verify the correctness of the implementations of the OODBPLs;
- To allow query optimization, and in general to allow the definition of CASE tools;
- To reach an agreement about the features of object-oriented languages, to allow the definition of standards and the exchange of data between different systems;
- To prove properties of programs.

Many researchers are trying to solve this problem by embedding some elementary operators, which realize the features of the object-oriented mechanism, in typed functional languages. This approach is very promising, since the object-oriented and the functional paradigms are similar enough to make this approach feasible, and furthermore, both would gain from an integration of the two paradigms. The most important semantic similarity is that both approaches deal with "active entities": functions and objects in the two cases. Functional languages could gain much expressive power if they were integrated with the functionalities of object-oriented languages. Similarly, an object-oriented language would gain clear semantics and a rich type system if it could be described as an extension of a typed functional language.

Much of the research on this subject is being performed in the field of type theory (see, for example, [Cook 90] and [Mitchell 90]).
Type theories for OODBPLs

OODBPLs need a strong and expressive type theory. A type theory for a language is defined by a language of types plus a set of rules which specify which terms of the language are associated with which types. Any type codifies some properties of the terms with that type (like the property of being objects with a given set of methods). In a strong type system the evaluation of a term which is typed never fails: for any type the associated property implies the property of never causing run-time failures.

Type theories are badly needed for long-lived systems since they provide a useful, though limited, way to describe properties of expressions which can be certified by the system. Besides this, types describe the structure of the values which they are associated with, so if a type system is expressive enough, the schema of a database can be described by a set of types.

Database programming languages require expressive type systems, since their types must be able to model the structure of the entities of the domain of discourse in a natural way, and OODBPLs require even more expressive power since the notion of inheritance supported by these languages requires that a parallel notion of subtyping be supported by the type system.

OODBPLs also require the ability to define recursive types and values. Recursive values are structured values which recursively refer to themselves. They are used to build mutually related objects like an object a which refers to b which refers to a. Recursive types are a familiar notion, but when they are combined with parametric polymorphism and type inclusion they create some interesting problems. One important problem is the definition of an algorithm to decide whether one recursive type is a subtype of another one.

Enhancements to the OO data model

The object-oriented data model is based on the notions of objects and classes. Objects model entities of the domain of discourse, classes model sets of homogeneous entities, and an association between two objects is codified by having a method in each of them which returns the other one; one-to-many associations are modelled in a similar way. The advantage of this approach is that cardinality constraints on the associations can be enforced using types, and that associations are dealt with exactly like object attributes.

Nevertheless, this approach can be criticized for the following reasons:
- associations are conceptually a higher level abstract notion, whose implementation should be decided by the DBMS, while attributes force the programmer to choose a specific implementation for them;
- associations are symmetric, while in the object model the association semantics is split between different objects;
- the enforcement of the inverse relation constraint, i.e. of the fact that the two methods modeling the associations are the opposite of each other, is left to the programmer;
- associations are not necessarily binary, and can have their own attributes; these aspects can only be modeled indirectly by means of attributes;
- associations relate objects which exist independently, and it should be possible to define them incrementally without redefining the structure of existing objects;
- operations on relationships as a whole are not possible in a straightforward way.

Another fundamental problem with an object-oriented data model is enforcing the constraint that the extension of a class (the set of its elements) coincides with the set of all the elements of the related type. This "extension coincidence constraint" is used in some OODBPL to enforce the referential constraint as follows: when an attribute p of an object is used to model an association with objects which must be in class classB, this referential constraint is enforced by defining the type of the attribute p to be the type typeB of the elements of class classB. Since all objects with typeB belong to classB, this type constraint ensures also that the object returned by method p always belongs to classB, enforcing the referential constraint.

But the enforcement of the extension coincidence constraint when objects are removed from classes creates problems. When an object is removed from a class, by extension coincidence, this operation is only allowed if the object can no longer be reached as a value with the associated type. In any object-oriented language this condition cannot be decided, and the implementation of any reasonable approximation of this condition requires either a system-controlled implementation of associations or an extremely costly operation. A common alternative approach is to mark any object removed from a class as "killed", which allows raising a failure when that object is successively accessed. This is slightly better than leaving dangling references, but cannot be regarded as an enforcement of the referential constraint.

To overcome these problems, extending the object-oriented data model with an association construct was proposed [Albano 91]. Associations are just subsets of the cartesian product of some types which can be used to represent the existence of associations between objects; constraints can be associated with these associations. This approach has many advantages:
- The actual implementation of associations is decided by the system, while in the other approach the user was required to implement the methods which model associations. One important advantage of this is that the system can maintain associative access structures for these associations and exploit them in an optimization phase.
- This notion of association is symmetric, and n-ary associations and associations with attributes can be represented naturally.
- It also allows the system to maintain the referential constraint with no need to enforce the "extension coincidence constraint". The programmer declares which classes are associated by an association, and when an element is removed from a class the system only has to check that it is no longer referred by that association, while in the object-oriented data model the system must check that no expression of the language can access that object any more. Checking the referential constraint in this approach is much easier for two reasons: (a) the programmer can specify more information, by distinguishing methods and associations, while in the object-oriented approach the system does not know which methods represent associations; (b) whereas methods are described procedurally, and cannot be inspected by the system, associations are defined declaratively, and the system has full knowledge of them.

The following are other possible enhancements to the object-oriented data model:
- The addition of operators to change the type of an object, either by extending its type to a subtype or by contracting it, without affecting its identity. These operators are needed to model the evolution of entities of the domain of discourse, such as persons which become students and later on workers. It should be possible to specify which type transitions are allowed and which are not. These operators cannot be statically type checked, and dynamic type checking also poses some difficult problems.
The possibility to model composite objects. In literature, this term is used with different meanings. For example, if a is a component of a composite object abc, this may mean either that when the abc is destroyed then a must be destroyed too, or that when abc is removed from a given class then a is removed from another one, or that a can only be a component of abc, or any combination of these three properties.

The possibility to define declaratively static or dynamic integrity constraints.

The possibility to define operations which are automatically executed when certain conditions are satisfied.

**Design of languages and systems allowing schema evolution**

The ability to modify the schema without losing any data stored and without jeopardizing the safety of any already compiled application is an important requirement which can be fulfilled at many different levels of completeness, and with many different mechanisms.

OODBPLs are well suitable in this case since the subclass mechanism helps to solve some of the problems in this field. For example, in order to add one attribute to the objects of one class with a given type, a subtype is defined by inheritance, which adds that attribute, and then all the objects of the old class are extended to the new type and moved in the new class. All the applications which refer to the old type can be still used, with no need for recompilation, thanks to the subtyping mechanism. All the data structures which refer to the extended objects still refer to them, since extension does not affect identity, and they refer them in a type-safe way, thanks to the subtyping mechanism.

The view mechanism would be another powerful tool for schema evolution: when the structure of a class is changed, a view mechanism can hide this fact from the old applications. Unfortunately, OODBPLs do not support a general view mechanism. This is due to the same kind of problems about object identity which were discussed in the section about query languages.

**Studies on new features of transaction and concurrency mechanisms**

According to the object-oriented paradigm, any object is an independent entity with an internal state which only cooperates with the other entities by exchanging messages. It is thus very natural to allow concurrency between objects. In this way any single application would be a concurrent program, where only the methods of a single object are necessarily executed sequentially, and where message passing is the synchronization and communication mechanism.

The most important feature of this kind of concurrency is that it is natural to use. Once an application has been described in terms of communicating objects, then it is easy to think of it in terms of communicating concurrent objects. Concurrent objects would be useful for three reasons: (a) some applications, such as graphical interfaces, can be programmed more easily with a concurrent style; (b) activities running in parallel in the domain of discourse can be modelled by concurrent long term processes; (c) concurrent languages enable the features of advanced computer architectures to be exploited better.

This kind of concurrency poses many new problems. The management of concurrency and the rollback in case of failures in DBPLs must be completely rethought within the context of concurrent objects. In traditional approaches, the failure of one transaction has no effect on concurrent transactions. In this new context, it is not clear what the atomicity unit is with regard to failure management, but in any case the failure of a transaction executed by an object could affect all the other objects which that object communicated with. This kind of research is not new in the world of databases, but no satisfactory solution has yet been found.

Another notion which changes in the context of OODBPLs, both with parallel or with sequential transactions, is serializability. Whereas in traditional databases serializability can be defined only in terms of the elementary operations of reading and writing, in the object-oriented approach the programmer could define weaker conditions of non-interference at the level of the methods which can be applied to the objects of one class. For example, if one object type implements a set, then two operations of element insertion in an object of that type can be freely inverted. This fact derives from the semantics of sets and is known to the programmer, but cannot be deduced from the usual serializability rules, since both the insertion operations write on the same data. So this kind of information would allow the system to accept wider classes of transaction schedules.

The interest of this kind of research is not limited to the OO approach, but extends to an approach where abstract data types can be defined by the programmer [Schwarz 84] [Weihl 89].

Another important field of research on OODBPLs, is the management of long-lived transactions. This requirement does not derive from the features of these languages, but rather from their wider application area, since OODBPLs can be used in contexts such as CASE and CAD where the lifetime of transactions spans from the traditional short time intervals to whole days. Also long term transactions are a well known field of research on DBMSs which acquires new interest with the development of OODBPLs.

Design of OODBPLs which fully support the data base related operations

Current OODBPLs often only offer a compromise between the functionalities of a full object-oriented programming language and those of a database programming language. This happens because the design of an uncompromised OODBPL entails solving the many open questions listed above. In our opinion, the most important of all of them is defining a formal framework for these languages, since it would allow building OODBPLs on a firm basis rather than by extending DBPLs with some OO features, or OOPs with some database features.

Most of the current approaches lack the facilities listed in the section above, many of them do not support a strong and rich type system, and often multiple inheritance is not completely supported. One possible solution would be to define an OODBPL from scratch, creating a strongly typed language with enough operators to support all the features of the OO approach and all the constructs of an enhanced object-oriented data model. It should offer:

- static and strong typing;
- a rich type system, with parametric and inclusion polymorphism, recursive types and type encapsulation mechanisms (parametric polymorphism is the ability to define functions which operate uniformly on all the types with a common structure);
- the features of an object-oriented language: object identity, state and method encapsulation, type inclusion, multiple inheritance;
- the constructs of an extended object-oriented data model: hierarchies of classes, hierarchies of associations with the possibility to define referential, cardinality and dependency constraints on them, operators to change the types of objects, operators to move objects among classes, integrity constraints, failure management and nested transactions;
- a module mechanism for structuring complex schemes and applications;
- an object mechanism with separation between interface and
implementation of an object type;
- a query language which is a subset of the full language.

References

[A1bano 85]

[A1bano 88]

[A1bano 91]

[Andrews 87]

[A1tkinson 83]

[A1tkinson 87]

[A1tkinson 89]

[Bancilhon 88]

[Banerjee 87]

[Cook 90]

[Dahl 66]

[Dittrich 90]

[GemStone 86]

[Goldberg 83]
A. Goldberg and D. Robson, Smalltalk–80: The Language and its Implementation, Addison–Wesley, 1983.

[Kim 90]

[Lécluse 88]

[Lécluse 89]

[Meyer 88]

[Mitchell 90]

[Mypoulos 80]

[Ontologic 87]

[Ontologic 89]
Ontologic Inc. ONTOS Object Database Documentation, Release 1.5 Brussels, April, 1989.

[Schmidt 77]

[Schmidt 90]

[Smith 81]

[Symbolics 88]

[Wegner 87a]

[Wegner 87b]

[Weihl 89]

[Zdonik 90]