O_2, an Object-Oriented Data Model

Christophe Lécluse
Philippe Richard
Fernando Velez

GIP Altaïr,
Domaine de Voluceau, B.P. 105,
78153, Le Chesnay Cedex, France.

Abstract

The *Altaïr* group is currently designing an object-oriented data base system called O_2. This paper presents a formal description of the object-oriented data model of this system. It proposes a type system defined in the framework of a set-and-tuple data model. It models the well known inheritance mechanism and enforces strong typing.

1. Introduction

One of the objectives of the *Altaïr* Group is to develop a new generation database system. The target applications are traditional business applications, transactional applications (excluding very high performance applications), office automation and multi-media applications.

The system we are designing is object-oriented. We briefly recall the main features of the object-oriented paradigm:

(i) Object identity. Objects have an existence which is independent of their value. Thus, two objects can be either identical, that is, they are the same object, or they can be equal, i.e., they have the same value.

(ii) The notion of *type*[1]. A type describes a set of objects with the same characteristics. It describes the structure of data carried by objects as well as the operations (*methods* in the object-oriented terminology) applied to these objects. Users of a type only see the interface of the type, that is, a list of methods together with their signatures (the type of the input parameters and the type of the result): this is called *encapsulation*.

(iii) The notion of *inheritance*: it allows objects of different structures to share methods related to their common part. Types are organized in an inheritance (or *subtype*) hierarchy which factorizes common structure and methods at the level at which the largest number of objects can share them[2].

(iv) *Overriding* and *late binding*. The body of a method in a given type may be redefined at any moment in any of its subtypes, yet keeping the same name. This frees the programmer from remembering the name of an overriden method in a given type, and therefore, the code is simpler and reusable because it is independent of the types that existed at the time the program was written. To offer this functionnality, the system has to bind method names to binary code at run time.

There is a clear interest in the database community for the object-oriented technology. First of all, types and inheritance are a powerful tool to model the real world. They also make systems extensible: by adding new types in a system, one can extend its capabilities. Object identity allows modeling object sharing and provides a natural semantics for object updates [Copeland & Khoshafian 86]. Second, this technology provides a framework to represent and manage both data and programs. It is a promising paradigm to solve the so-called *impedance mismatch*: the awkward communication

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and / or specific permission.

© 1988 ACM 0-89791-268-3/88/0006/0424 $1.50

[1] The term *class* is frequently used; however, in addition to the intensional notion of type, it contains an extensional aspect, as it denotes the set of all objects of the system which conforms to the type at a given time.

[2] Another mechanism allowing objects to share operations is called *delegation*. It is the basis of the so-called "Actor languages". We will not consider it in this paper.

between a query language and a programming language that results when developing applications with a database system. Third, it provides good software engineering tools that make the programming task much easier.

Object-oriented database systems are being currently built. Most of them are prototypes [Banerjee & al 87], [Zdonik 84], [Nixon & al 87], [Bancilhon et al 87] and few of them are commercial products [Copeland & Maier 84], [Andrews & Harris 87]. The overall objective of these systems is to integrate database technology (such as data sharing, data security, persistency, disk management and database query languages) with the object-oriented approach in a single system.

However, there is a lack of a strong theoretical framework for object-oriented systems. This paper is a step in this direction. It proposes the data model foundations for an object-oriented database system. The originality of this model, called O_2, is its type system defined in the framework of a set and tuple data model. We think that what makes our approach different from other object-oriented approaches is that we use set and tuple constructors to deal with arbitrary complex objects, and the type system enforces strong typing, yet overriding is allowed.

There already exist models dealing with inheritance such as [Bruce & Wegner 86] and [Cardelli 84]. In [Bruce and Wegner 86] types are modeled as many-sorted algebras. A type is a subtype of another if their exist suitable (not necessarily injective) "coercion" operators which behave as homomorphisms between the algebras. In [Cardelli 84], a safe, strongly typed system is proposed in which the semantics of subtyping for tuple-structured types corresponds to set inclusion between the corresponding type interpretations (this semantics is different from the previous one). Functions are typed and rules for subtyping among functional types are also given.

We have borrowed Cardelli's interpretation for tuple types, as it leads to an intuitive notion of subtyping of tuple structures. Our model is different to these proposals in that (i) we propose a different rule for inheritance of methods (for functional subtyping, in Cardelli's terms), (ii) set-structured objects are introduced, and objects may form a directed graph in which cycles are allowed, and (iii) methods can be directly attached to objects.

Our "tuple-and-set" construction of objects is similar to that of [Bancilhon and Khoshafian 86] and specially to that of [Kuper and Vardi 84] where identifiers (called addresses) are also introduced.

This paper is organized as follows. Section 2 gives an informal overview of our approach and exposes it through examples. Section 3 gives a definition of objects. Section 4 gives the semantics of types and inheritance relationship. Finally, the notion of database is introduced in Section 5. Section 6 contains some concluding remarks and open problems.

2. Informal Overview

Let us introduce some of the notions of this model using examples. Objects represent our (computer) world. They are made up of an object identifier (a name for the object) and a value. Values can be atomic (string, integers, reals,...), tuple-structured or set-structured.

(ob_1, <name: "Smith", age: 32>)
(ob_2, <name: "Doe", age: 29, salary: 9700>)
(ob_3, {ob_1, ob_2})

The first two objects are examples of tuple objects and the last one is a set object. Atomic objects here are ages and names (they actually also have identifiers, as shown later). Objects can, of course, reference other objects and this allows the definition of complex objects. We can have mutually referencing objects, as shown in the following example:

(ob_4, <name: "john", spouse: ob_5>)
(ob_5, <name: "mary", spouse: ob_4>)

This possibility makes our objects more general than the simple nested tuple-and-set-objects.

A type has a name and it contains a structure and a set of methods, applying to these objects. A structure will be either a basic structure (String, Integer, Real for example), tuple structures and set structures. The following example of type structures will be used throughout the paper:

Person = <name: String, age: Integer, sex: String>
Employee = <name: String, age: Integer, sex: String, salary: Integer>
Male = <name: String, age: Integer, sex: "male">
Persons = {Person}
Employees = {Employee}
Married-Person = <name: String, spouse: Married-Person, children: Persons>

The type structure of "Person" represents the set of all tuple objects having a name field which is a string, and an age field which is an integer. The type structure of "Male" is as "Person" except that the sex field is restricted to contain the string "male". The type structure of "Persons" represents all objects which are sets of persons. Given a set of objects Θ, we shall call the interpretation of a type structure (say "Person") the set of all objects of this set having the corresponding structure. If Θ is the set of all objects ob_1 to ob_5, then the interpretation of "Persons" will be the object ob_3, whereas the interpretation of "Person" will be the two objects ob_1 and ob_2. Indeed, these two objects have name and age fields with the corresponding structures (string and integers). Notice that we allow the objects to have addi-

tional fields (the object ob_2 also have a salary field). In the same manner, the interpretation of the "Employee" structure is the set containing only the ob_2 object. So the interpretation of "Employee" is included in the interpretation of type structure "Person". This is an intuitive result, because we want to say that every employee is a person. This "is-a" relationship between type structures is what is called inheritance in the object-oriented terminology.

The notion of inheritance also deals with methods. As employees are persons, a method defined for every person can be applied to an employee. Moreover, if a method (say "name") is defined for both persons and employees, then we shall put some constraint on these methods, in order to make them "compatible". Such a compatibility is necessary to be able to perform type checking.

3. Objects

In this section, we define the notion of objects. We suppose given:

- A finite set of *domains* $D_1, ..., D_n$, $n \geq 1$ (for example, the set Z of all integers is one such domain). We note **D** the union of all domains $D_1, ..., D_n$. We suppose that the domains are pairwise disjoint.
- A countably infinite set **A** of symbols called *attributes*. Intuitively, the elements of **A** are names for structure fields as we shall see later.
- A countably infinite set **ID** of symbols called *identifiers*. The elements of **ID** will be used as identifiers for objects.

Let us now define the notion of *value*.

Definition 1: Values

(i) The special symbol *nil* is a value, called a *basic value*.

(ii) Every element v of **D** is a value, called a *basic value*.

(iii) Every finite subset of **ID** is a value, called a *set-value*. Set-values are denoted in the usual way using brackets.

(iv) Every finite partial function from **A** into **ID** is a value, called a *tuple-value*. We denote by $<a_1 : i_1 , ..., a_p : i_p>$ the partial function t defined on $\{a_1, ..., a_p\}$ such that $t(a_k)=i_k$ for all k.

We denote by V the set of all values. □

We can now define the notion of object.

Definition 2: Objects

(i) An *object* is a pair o = (i, v), where i is an element of **ID** (an identifier) and v is a value.

(ii) We define, in an obvious way, the notion of *basic* objects, *set-structured* objects and *tuple-structured* objects.

(iii) O is the set of all objects, that is $O = \mathbf{ID} \times V$. □

This "tuple-and-set" construction of objects is similar to that of [Bancilhon and Khoshafian 86] and specially to that of [Kuper and Vardi 84] where identifiers (called addresses) were also introduced.

In the following, we need some technical notations: If o=(i,v) is an object then *ident*(o) denotes the identifier i and *value*(o) denotes the value v. We will denote by *ref* the function from O in $2^{\mathbf{ID}}$ which associates to an object the set of all the identifiers appearing in its value, i.e., thode referenced by the object. We can use a graphical representation for objects as follows :

Definition 3: Object graph

If Θ is a set of objects, then the graph *graph(Θ)* is defined as follows:

(i) If o is a basic object of Θ then the graph contains a vertex with no outgoing edge. The vertex is labeled with the value of o.

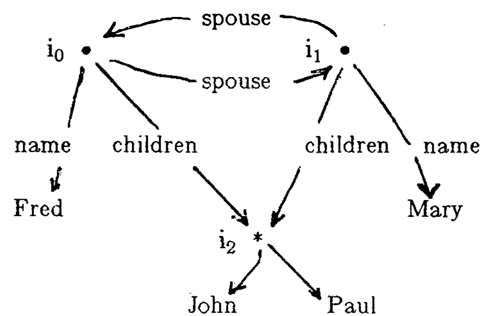
(ii) If o is a tuple-structured object .br $(i, <a_1:i_1,..., a_p:i_p>)$, the graph of o contains a vertex, say v, represented by a dot (•) and labeled with i, and p outgoing edges from v labeled with $a_1, ..., a_p$ leading respectively to the vertex corresponding to objects $o_1, ..., o_p$, where o_k is an object identified by i_k (if such objects exist).

(iii) if o is a set-structured object .br $(i, \{i_1,..., i_p\})$, the graph of o contains a vertex, say v, represented by a star (*) and labeled by i, and p unlabeled outgoing arcs from v leading respectively to the vertex corresponding to objects $o_1,..., o_p$, where o_k is an object identified by i_k (if such objects exist). □

We illustrate this definition with an example. Let Θ be the set consisting of the following objects:

$o_0 = (i_0, <spouse : i_1, name : i_3, children : i_2>)$
$o_1 = (i_1, <spouse : i_0, name : i_4, children : i_2>)$
$o_2 = (i_2, \{i_5, i_6\})$, $o_3 = (i_3, "Fred")$, $o_4 = (i_4, "Mary")$
$o_5 = (i_5, "John")$, $o_6 = (i_6, "Paul")$

Θ is represented by the following graph:



It is important to note that, referring to Definition 3, we cannot build the graph representation of any set of objects. For example, if an identifier i appears in a value, there must be an object identified by it. Intuitively speaking, identifiers are pointers on objects and there must be no dangling pointers in our set of objects. This leads us to introduce the notion of consistency for a set of objects.

Definition 4: Consistent set of objects

A set Θ of objects is consistent iff :

(i) Θ is finite,

(ii) The *ident* function is injective on Θ (i.e, there is no pair of objects with the same identifier),

(iii) for all $o \in \Theta$, *ref*(o) $\subseteq$ *ident*(Θ) (i.e every referenced identifier corresponds to an object of Θ). □

In the following, we denote by $\Theta(i)$ the value v such that the object (i, v) is in Θ.

In value-based systems (i.e, in systems where no object identity exists, such as relational systems) there is no need to distinguish between identical objects and equal objects since the two notions are the same. On the contrary, object-oriented systems need to distinguish them as there is a sharp distinction between values and objects.

Definition 5: Equalities

(i) *0-equality*: two objects o and o' are 0-equal (or *identical*) iff o=o' (in the sense of mathematical pair equality),

(ii) *1-equality*: two objects o and o' are 1-equal (or simply *equal*) iff value(o) = value(o'),

(iii) *ω-equality*: two objects o and o' are ω-equal (or *value-equal*) iff span-tree(o) = span-tree(o') where span-tree(o) is the tree obtained from o by recursively replacing an identifier i (in a value) by the value of the object identified by i. □

Equality implies value-equality, but the converse is not true since many distinct objects may have the same span tree. These definitions of equality correspond to identity, shallow-equality and deep-equality of Smalltalk 80 [Goldberg and Robson 83]. We must notice that the span-tree build from an object may be infinite (in the case of cyclic objects). So, this construction cannot be used (directly) as a decision procedure for testing value-equality.

4. Types

A type is an abstraction that allows the user to encapsulate in the same structure data and operations. In our model, the static component of a type is called a type structure. As we shall see later, our notion of type bears some similarity to abstract data types. Users of a type only see its abstract part, that is, the interface of its methods, whereas the programmer of the type is concerned with the implementation. However, a type has only one implementation.

In what follows, we shall decompose the process of defining the syntax, semantics and subtype relationship among types in two steps. First, in section 4.1, we shall define the syntax of type structures as well as the notion of schema. Then, we shall give the semantics of a schema with respect to a consistent set of objects. A partial order among type structures will be defined using this semantics. Second, in section 4.2, the same treatment will be given to methods. We shall bring up pieces together in section 4.3 with the notion of "type systems". We begin by defining the set of type names.

Definition 6: Type names

Bnames is the set of names for basic types containing :

(i) the special symbols *Any* and *Nil*,

(ii) a symbol d_i for each domain D_i. We shall note D_i=dom(d_i),

(iii) a symbol 'x for every value x of D.

Cnames is a set of names for constructed types which is countably infinite and disjoint with *Bnames*. *Tnames* is the union of *Bnames* and *Cnames* and it is the set of all names for types. □

In order to define types, we assume that there is a finite set *MT* whose elements are called methods and which shall play the role of operations on our data structures. For the moment, we can think of the elements of *MT* as uninterpreted symbols. We shall define them in section 4.2

Definition 7: Types

Basic type (Btypes) : a basic type is a pair (n,m) where n is an element of *Bnames* and m a subset of *MT*[3].
Constructed types (Ctypes) : A constructed type is either

(i) a triple (s,t,m) where s is an element of Cnames, t is an element of Tnames, m a subset of *MT*. We shall denote such a type by (s = t,m).

(ii) a triple (s, t, m) where s is an element of Cnames and t is a finite partial function from A to Tnames and m a subset of *MT*. We shall denote such a type by (s $=<a_1:s_1, \ldots, a_n:s_n>$, m) where $t(a_k)=s_k$ and call it a *tuple structured type*,

(iii) a triple (s, s', m) where s is an element of Cnames and s' an element of Tnames and m a finite subset of *MT*. We shall denote such a type by (s={s'}, m) and call it a *set structured type*.

[3] The link between basic types and domains D_i will be given in the following section, in the definition of interpretations.

A *type* is either a basic or a constructed type. The set of all types is denoted by T. □

4.1. Type structures

In this subsection we are interested in the static part of a type, i.e. in its structure.

4.1.1. Definitions

Definition 8: Type structures

Basic type structure: let t=(n,m) be a basic type, we call n the *basic type structure* associated to t.

Constructed type structure: let t=(s=x,m) be a constructed type, we call "s=x" the *constructed type structure* associated to t.

Given a type t, its structure part will be denoted by *struct*(t) and its methods part by *Methods*(t). Intuitively, a type structure is a type in which the methods part is hidden, i.e it is the data part of the type. Note that recursion (or transitive recursion) is allowed in type definitions, that is, one of the s_i may be s. The type structure "Married-Person" is an example of recursively defined type.

Type structures are analogous to GALILEO's concrete types [Albano & al 85] except that type structures only exist within types.

For the same reasons as in section 2, we need a notion of consistency for a set of expressions defining type structures. In order to define it formally, we need some technical notations:

(1) If t is a type, then *name*(t) is the name of the type, that is the first component in its definition.

(2) If st is a type structure associated to the type t, we call "name of the type structure" st, the name of t and we note *name(st)*=name(t).

(3) If st is a type structure (associated to a type t), we call "set of types references for st" and denote by ***refer(st)*** the set of all types names appearing in the structure st.

Definition 9: Schemas

A set Δ of constructed type structures is a *schema* iff

(i) Δ is a finite set,

(ii) *name* is injective on Δ (only one type structure for a given name),

(iii) for all st ∈ Δ, *refer*(st) ∩ Cnames ⊆ *name*(Δ) (i.e. there are no dangling identifiers). □

Nota Bene : In a schema, we can identify a *type name* of name(Δ) with the corresponding type structure in Δ, and we shall use this convention in the sequel of the paper.

We illustrate the notion of a schema with two examples : Let Δ be the set consisting of the following type structures :

age = integer,
person = <name : string, age : age>

Δ is a schema. If we take off the type structure "age" from Δ, it is no longer a schema. On the other hand, the following set of type structures is also a schema:

person = human
human = person

This set of type structures may be not useful but it is well defined and has an interpretation as we shall see in the next section.

4.1.2. Interpretation

This section deals with the definition of the semantics of the type structure system presented above. It will be given by a particular function which associates subsets of a consistent set of objects to type structure names.

Definition 10: Interpretations

Let Δ be a schema and Θ be a consistent subset of the universe of objects O. An *interpretation* I of Δ in Θ is a function from Tnames in $2^{ident(\Theta)}$, satisfying the following properties:

Basic Type Names

1) I(Nil) ⊆ {i ∈ ident(Θ) [4] / (i, Nil) ∈ Θ}
2) $I(d_i)$ ⊆ { id ∈ ident(Θ) / Θ(id) ∈ D_i } ∪ I(Nil)
3) I('x) ⊆ { id ∈ ident(Θ) / Θ(id) = x } ∪ I(Nil)

Constructed Type Names

4) if $s = <a_1 : s_1 , \ldots , a_n : s_n>$ is in Δ then
I(s) ⊆ { id ∈ ident(Θ) / Θ(id) is a tuple structured value defined (*at least*) on $a_1,\ldots,a_n$ and Θ(id) (a_k) ∈ I(s_k) for all k} ∪ I(Nil)
5) if s = { s' } is in Δ then
I(s) ⊆ { id ∈ ident(Θ) / Θ(id) ⊆ I(s') } ∪ I(Nil)
6) if s = t is in Δ then I(s) ⊆ I(t)

Undefined Type Names

7) if s is neither a name of basic type nor a name of the schema Δ, then I(s) ⊆ I(Nil) □

Definition 11: Model of a schema

(i) *Partial order on interpretations*: An interpretation I is *smaller* than an interpretation I' iff for all s ∈ Tnames , I(s) ⊆ I'(s)

(ii) *Model*: Let Δ be a schema and Θ be a consistent set of objects. The model M of Δ in Θ is the greatest interpretation of Δ in Θ. □

As we shall show later, this definition is well founded. Some important remarks are in order at this point. Intuitively, the model M(s) of a constructed type structure of

[4] Recall that ident(Θ) denotes the set of the identifiers of all objects of Θ and that Θ(id) denotes the (only) value v such that (id, v) is in Θ.

name s is the set consisting of all objects (identifiers of objects) having this structure. For example, if Θ is the set

$\{(i_0,\text{Nil}),(i_1,\{i_2,i_3\}),(i_2,1),(i_3,4),(i_4,<a:i_2>),(i_5,<a:i_2,b:i_3>)\}$ and Δ is the schema

$\{s_1 = <a:\text{Integer}>,\ s_2 = <a:\text{Integer},\ b:\text{Integer}>,\ s_3=\{\text{Integer}\}\}$

then $M(s_1) = \{ i_0, i_4, i_5\}$

$M(s_2) = \{ i_0, i_5\}$

$M(\text{Integer}) = \{i_0, i_2, i_3\}$

$M(s_3) = \{ i_0, i_1 \}$

We can notice that the value of an interpretation of a *Basic type name* does not depend on Δ which is an intuitive result. Moreover, if (the identifier of) an object belongs to the model M(s) of a tuple structure, then it also belongs to the models of tuple type structures which are sub-structures of s. In the example below, i_5 is in $M(s_2)$ but also in $M(s_1)$. This property will allow us to give a simple set-inclusion semantics for the sub-typing relation among type structures defined in the following subsection. This interpretation is derived from an interpretation which was originally proposed in [Cardelli 84]. Note that any attribute could be added to a tuple-structured object and the latter would still have a well-defined type. Such "added" attributes will be referred to as "exceptional", and their manipulation is considered in section 4.4.

We now have to prove that our definition of the model of a schema is well founded. Given a schema Δ and a consistent set of objects Θ, there is a finite number of interpretations of Δ defined on Θ. Therefore, in order to prove that the greatest interpretation exists, we just have to prove that the union of two interpretations is an interpretation.

Let I_1 and I_2 be two interpretations, and I be the function defined by $I(s) = I_1(s) \cup I_2(s)$, for every type name s. This function I clearly verifies properties 1, 2 and 3 of the Definition of interpretations. If $s=<a_1:s_1,...,a_n:a_n>$ and id is an element of I(s), (for example, an element of $I_1(s)$), then $\Theta(id)(a_k)$ is in $I_1(s_k)$ for all k, because I_1 is an interpretation. So $\Theta(id)(a_k)$ is in $I(s_k)$ for all k, and I verifies the property 4) of Definition 10. We can show in the same manner that I also verifies properties 5 and 6.

In conclusion, there is a greatest interpretation M, and we have:

$$M(s) = \bigcup_{I \in INT(\Delta)} I(s),$$

for every type name s, where $INT(\Delta)$ denotes the set of all interpretations of Δ (in Θ).

4.1.3. Partial Order Among Type Structures

Definition 12: Partial order $\leq_{st}$

Let s and s' be two type structures of a schema Δ. We say that s is a substructure of s' (denoted by $s \leq_{st} s'$) iff $M(s) \subseteq M(s')$ for all consistent set Θ. □

For example, if Δ consists of the following type structures :

$s_1 = <a:\text{Integer}>$,

$s_2 = <a:\text{Integer},\ b:\text{Integer}>$,

$s_3 = <c:s_1>$, $s_4 = <c:s_2>$,

$s_5 = \{s_1\}$, $s_6 = \{s_2\}$, $s_7 = <a:'1>$

then the following relationships holds among these structures :

$s_2 \leq_{st} s_1 \qquad s_4 \leq_{st} s_3$

$s_7 \leq_{st} s_1 \qquad s_6 \leq_{st} s_5$

The first relationship ($s_2 \leq_{st} s_1$) comes from the interpretation of tuple type structures. Let us establish the second one ($s_4 \leq_{st} s_3$). Let id be the (identifier of an) object belonging to $I(s_4)$. We know from the definition that $\Theta(id)(c)$ belongs to $I(s_2)$ and so to $I(s_1)$ because we have $s_2 \leq_{st} s_1$. We conclude that id belongs to $I(s_3)$ and so $I(s_4) \subseteq I(s_3)$. The inequality $s_6 \leq_{st} s_5$ can be established in the same manner and the relation $s_7 \leq_{st} s_1$ is obviously true.

Definition 12 gives a semantic definition for the subtyping relationship $\leq_{st}$. The following theorem gives a syntactic characterization of it.

Theorem 1:

Let s and s' be two type structures of a schema Δ. s is a substructure of s' ($s \leq_{st} s'$) iff

(i) either s and s' are tuple structures s = t and s' = t', such that t is more defined than t' and for every attribute "a" such that t' is defined, we have $t(a) \leq_{st} t'(a)$.

(ii) or s and s' are set structures $s = \{s_1\}$ and $s' = \{s'_1\}$ and we have $s_1 \leq_{st} s'_1$.

(iii) or s = 'x, s' is a basic type structure and x is in dom(s'). □

Proof:

The validity of this characterization can be easily established by induction. The completeness can be established with a case study, inspecting successively tuple structured types, set structured types and basic types. □

This theorem gives a syntactical means for checking type structure subtyping.

4.2. Methods

In Section 4.1, we have presented the syntax and semantics of type structures. In this subsection, we define, in the same way, the syntax and semantics of operations, which we call methods in this context. These operations

will consist of (first order) functions.

4.2.1. Definition

We assume that we have a countable set Mnames of symbols that will be used as names for methods.

Definition 13: Signatures

Let Δ be a schema. A ***signature*** over Δ is an expression of the form:

$$s_1 \times s_2 \times \ldots \times s_n \rightarrow s$$

where $s_1, s_2, \ldots, s_n$, and s are types names corresponding to type structures in Δ or basic types names.

A *method* m is a pair m=(n,σ) where n is a method name (an element of Mnames) and σ is a signature. We shall denote by *name*(m) the name of the method m and by *sign*(m) the signature of the method m. □

In the object-oriented formalism, methods are related to types (or type structures) using the first argument of their signature, so we have:

Definition 14: Methods

Let m=(n, $s_1 \times \ldots \times s_n \rightarrow s$) be a method. We say that m *is defined on* s_1. □

4.2.2. Interpretation

In this subsection, we define the model of a signature σ

Definition 15: Model of a signature

Let Δ be a schema and σ a signature over Δ ($\sigma = s_1 \times \ldots \times s_n \rightarrow s$). If Θ is a consistent set of objects, then the *model of σ in Θ* is the set of all **partial** functions from $M(s_1) \times \ldots \times M(s_n)$ into M(s) where $M(s_k)$ is the model in Θ of the structure of Δ identified by s_k. □

Let us illustrate these definitions by an example. Let Δ be the schema introduced in section 2 restricted to the type structures "Person", "Persons", "Employee", "Employees" and "Male". We consider now the following signatures:

σ_1 = Persons × Person → Boolean
σ_2 = Employees × Employee → Boolean
σ_3 = Person → Person
σ_4 = Male → Employee
σ_5 = Employee → Integer

We shall take the following set of objects Θ as interpretation domain:

(i_0,nil), (i_1, <name: i_{16}, age:i_7 sex:i_8>),
(i_2, <name: i_{17}, age:i_9, sex:i_{10}>)
(i_3, <name: i_{18}, age:i_9, sex:i_8, salary: i_{11}>),
(i_4, <name: i_{19}, age:i_{13}, sex:i_{12}, salary:i_{11}>)
(i_5, {i_1, i_2}), (i_6, {i_3, i_4}), (i_7, 20), (i_9, 25), (i_{11}, 130000),
(i_{13}, 35), (i_8, "male"), (i_{10}, "varying"), (i_{12}, "female"),
(i_{14}, false), (i_{15}, true), (i_{16}, "Smith")
(i_{17}, "Blake"), (i_{18}, "Jones"), (i_{19}, "Nash")

Using definition 11 in the previous subsection, we can build the models of the type structures defined in Δ:

M(Person) = {i_0, i_1, i_2, i_3, i_4}
M(Persons) = {i_0, i_5, i_6}
M(Employee) = {i_0, i_3, i_4}
M(Employees) = {i_0, i_6}
M(Male) = {i_0, i_1, i_3}
M('"male") = {i_0, i_8}
M(String) = {i_0, i_8, i_{10}, i_{12}, i_{16}, i_{17}, i_{18}, i_{19}}
M(Integer) = {i_0, i_7, i_9, i_{11}, i_{13}}
M(Boolean) = {i_0, i_{14}, i_{15}}

The model of the signature σ_1 is the set of all **partial** functions from {i_0, i_5, i_6} × {i_0, i_1, i_2, i_3, i_4} into {i_0, i_{14}, i_{15}}. Intuitively, the model of the signature σ_1 is the set of functions assigning a boolean object to some pairs (i,j) where i is (the identifier of) a set of person objects and j is (the identifier of) a person object.

We shall use this interpretation of signatures in the following subsection which introduces an ordering among signatures.

4.2.3. Partial order among signatures.

Definition 16: Partial order $\leq_m$

Let Δ be a schema and f and g two signatures over Δ. We say that f is smaller than g (or that *f refines g*) iff $M(f) \subseteq M(g)$ for all consistent set Δ. This ordering will be denoted by $\leq_m$. □

Looking at the schema of the previous example , we can see that the following inequalities hold:

$$\sigma_2 \leq_m \sigma_1 \quad \text{and} \quad \sigma_4 \leq_m \sigma_3$$

In fact, let Θ be any consistent set of objects and f be a partial function in M(σ_2). f is a (partial) function from M(employees) × M(employee) in M(Boolean). We have seen in subsection 4.1.3 that employees $\leq_{st}$ persons and employee $\leq_{st}$ person, and hence, M(employees) $\subseteq$ M(persons) and M(employee) $\subseteq$ M(person). So f is also a partial function from M(persons) × M(person) in M(boolean), so f is in M(σ_1). A similar proof can be constructed for the inequality $\sigma_4 \leq_m \sigma_3$.

Intuitively, $\sigma \leq_m \sigma'$ means that we can use a method of signature σ' "in place of" a method of signature σ. In the example above, we can apply a method of signature σ_1 to a set of employees and an employee because, employees are persons. This partial order models inheritance of methods, just as the ordering $\leq_{st}$ models inheritance of data structures. In the following section, we put data structures and methods together to define type systems and we use the ordering $\leq_{st}$ and $\leq_m$ to define inheritance of types. The following theorem gives an easy syntactical equivalence to the definition of the partial order $\leq_m$ among signatures.

Theorem 2

Let f and g be two signatures over a schema Δ. Then, f

$\leq_m$ g iff:

$f = s_1 \times \ldots \times s_n \rightarrow s$
and $g = s'_1 \times \ldots \times s'_n \rightarrow s'$
and $s_k \leq_{st} s'_k$ for k=1,2,...,n
and $s \leq_{st} s'$. □

Proof :

In order to clarify the proof, we assume, without loss of generality, that the methods signatures are of the form: $\sigma = s_1 \rightarrow s$, and $\sigma' = s'_1 \rightarrow s'$. Suppose that $\sigma \leq_m \sigma'$. Every partial function from $M(s_1)$ to M(s) is then a partial function from $M(s'_1)$ to M(s'). So, we necessarily have $M(s_1) \subseteq M(s'_1)$ and $M(s) \subseteq M(s')$.

Conversely, if these two inclusions hold, then every partial function from $M(s_1)$ to M(s) is clearly also a partial function from $M(s'_1)$ to M(s'). □

In the part concerning methods, our definition differs from the "classical" definitions of data type theory [Bruce & Wegner 86], [Cardelli 84], [Albano & al 85]. In these settings, functional types may be constructed: the type $r \rightarrow s$ has as instances functions having r as domain and s as co-domain. The general rule of subtyping among functional types can be expressed as follows:

$$\text{if } r' \leq r \text{ and } s \leq s' \text{ then } r \rightarrow s \leq r' \rightarrow s'$$

This means that a function with domain r and codomain s can always be considered as a function from some smaller domain r' to some larger codomain s'. This is a necessary condition for the type system to be *safe*, that is, to guarantee that a run-time error will never be caused by a syntactically well-typed expression. If we had adopted this rule in our framework, the subtype relation would be inverted between the right-hand sides of the signatures in theorem 2, i.e., $s' \leq s$ instead of $s \geq s'$.

Our choice leads to a less restrictive type system, but we give up safety. There are three main reasons for such a choice. First, we want to be able to inherit an "add" method defined on sets. Suppose there are two types S and T such that $S \leq_{st} T$ then we have $\{S\} \leq_{st} \{T\}$. Let add_S be a method of signature $\{S\} \times S \rightarrow \{S\}$, and add_T be a method of signature $\{T\} \times T \rightarrow \{T\}$, then we want $add_S \leq_m add_T$, which is a necessary condition to have $\{S\} \leq \{T\}$ (see definition 18). Second, this model is intended to be a foundation for an object-oriented layer on top of C which itself is not type safe. Third, we shall implement run-time checking for the cases where static type checking is not sufficient.

4.3. Type systems

Definition 17: Type systems

A set of types Π is a *type system* iff

(i) the set of structures associated to Π is a schema ;

(ii) for all type t ∈ Π, and for all method m ∈ Methods(t)[5], m is defined on struct(t). □

[5] We recall that Methods(t) denotes the set of methods of the type t.

Now, given a type system Π, we must be able to compare two types t and t', with respect to their structures and to the methods they contain.

Definition 18: Subtyping

Let Π be a type system and t and t' two types of Π. We say that *t is a subtype of t'* and we note $t \leq t'$ iff

(i) $struct(t) \leq_{st} struct(t')$

(ii) for all m ∈ Methods(t), there exist m' ∈ Methods(t') such that name(m) = name(m') and $sign(m) \leq_m sign(m')$. □

We illustrate this by the following example : Let Π be the type system of the example of section 4.2.2. where

Methods(Person) = {(husband, σ_3)}
Methods(Persons) = {(parent, σ_1)}
Methods(Employee) = {(husband, σ_3), (salary, σ_5)}
Methods(Employees) = {(parent, σ_1),(manager,σ_2)}
Methods(Male) = {(hire, σ_4), (husband, σ_3)}

In this type system, the following subtype relationships hold : employee ≤ person, male ≤ person and employees ≤ persons

4.4. Objects revisited

We shall now extend the definition of an object in order to encapsulate in the same structure data and operations.

Definition 19: Objects revisited

An *object* o is a triple (i, v, m) where i and v are as in Definition 2 and m is a set of methods. The first component of the signature of every method of m is a type structure whose interpretation contains o. □

The set of methods of an object can be empty, and in this case, it will be manipulated through the methods of the type it possesses. This notion is useful in the following cases:

(i) When handling exceptions. For example, let us assume that we define in type "Employee" a method "increase salary" to compute the salary of an employee. Suppose that one of these employees is the CEO and that his salary has to be computed in different way than for regular employees. One could create a specific subtype of employee in order to override the "increase salary" method of type employee. This would be heavy and it is more natural to define a specific method for the CEO object.

(ii) "Exceptional" attribute handling (see section 4.1.2) As full encapsulation is preserved, the only way to access and/or modify an exceptional attribute of an object is via a method attached to the object.

(iii) Still another application is when representing the data model in terms of itself. This kind of self-representation is very frequent in object-oriented frameworks (the predefined classes "class" and

"metaclass" of Smalltalk 80 are a good example). Types could be represented as objects belonging to a predefined type "type" and "type methods" could be easily defined attaching them to these objects. An example of a type method for a type T is a customized method for instanciating instances of T.

5. Databases

In this section, we introduce the notion of database. Informally, a database is a type system together with a consistent set of objects representing the instances of the types at a given moment.

Definition 20: Databases

A database is a tuple $(\Pi, \Theta, <_{db}, \mathbf{ext}, \mathbf{impl})$ where

(i) Π is a type system, and Δ is the associated schema,

(ii) Θ is a consistent set of objects,

(iii) $<_{db}$ is a strict partial order among Π,

(iv) **ext** is an interpretation of Δ in Θ.

(v) **impl** is a function assigning a function to every method m of a type t.

Moreover, we impose that the following properties hold:

(1) $t <_{db} t'$ implies $t \leq t'$.

(2) If $t <_{db} t'$ and $t <_{db} t''$ then t' and t'' are comparable.

(3) $\Theta = \bigcup_{t \in \Pi} \mathrm{ext}(t)$.

(4) $\mathbf{ext}(t) \cap \mathbf{ext}(t') = \emptyset$ if t and t' are not comparable.

(5) If t is a type of Π and m a method of t having signature $t \times \ldots \times s_n \rightarrow s$, then **impl**(m) is a function defined at least from $\mathbf{ext}(t) \times \ldots \times \mathbf{ext}(s_n)$ in $\mathbf{ext}(s)$. □

This definition deserves some comments. The extension of a type is an interpretation but may not be a model. Indeed, a model contains all the possible objects which satisfy a given structure. For example, there may be two types of structure "integer" (say age and weight) in a data base. These types have the same model but a given extension as defined by an user will not contain the same objects. The $\leq$ ordering of definition 18 models the notion of subtyping. That is, two types t and t' are comparable using $\leq$ if one *can be* a subtype of the other. The ordering $<_{db}$ is the actual inheritance types hierarchy, as *defined* by the user. This ordering must satisfy property (1), that is, the user can declare that t is a subtype of t' ($t <_{db} t'$) only if it is allowed by the model ($t \leq t'$). For example, the type system may contain the types:

Age = (Integer, {+,-}) and
Weight = (Integer, {+,-})

with corresponding signatures for the methods + and -. We have the inequalities (Age $\leq$ Weight) and (Weight $\leq$ Age) but the user does not intend to consider an age as a weight nor a weight as an age, and Age and Weight will be incomparable for $<_{db}$. Property (2) says that we do not allow multiple inheritance. This is a constraint we introduced for the O_2 system because it is still an open problem to decide whether multiple inheritance is a useful modelization tool. In any case, our semantics would still be valid in the context of multiple inheritance. Property (3) says that Θ is the union of all type extensions. There are no database objects not belonging to any type extension. Property (4) says that an object o cannot belong to the extension of two types t and t' if they are incomparable for $<_{db}$. Consider the types Age and Weight above. The object $(i_1, 1)$ belongs to the model of Age and to the model of Weight, but if we allow this object to belong to both ext(Age) and ext(Weight), we violate the user intention which was to isolate these two types.

6. Concluding Remarks

The main contribution of this paper is to propose a data model for an object-oriented data base system. The model includes the following features:

(i) Objects may have a tuple or set structure (or be atomic). They form a directed graph in which cycles may appear. Consistent sets of objects are used as interpretation domains for type structures and method signatures.

(ii) Types consist of a type structure and a set of methods. Their structure may be recursively defined. The interpretation of tuple-structured types is unusual in the database world and follows the original proposal of [Cardelli 84]. It allows to give a simple set inclusion semantics to the partial order among type structures ($\leq_{st}$). Methods are defined as a name together with a signature and are interpreted as a function. The interpretation of method signatures allows again a simple set inclusion semantics for the partial order among signatures ($\leq_m$). The subtyping relationship is defined using the ordering $\leq_{st}$ and $\leq_m$. Notice that although our database definition restricts inheritance to simple inheritance, our model deals with multiple inheritance.

(iii) The $\leq_m$ relation differs from other proposals ([Cardelli 84], [Bruce & Wegner 86]) in that it is less restrictive, but the type system is no longer safe (that is, run-time errors may be caused by a syntactically well-typed expression). This decision was mainly motivated because of its increased flexibility and by the fact that we are not building a new language, but rather an object layer according to this model on top of existing programming languages with unsafe type systems, such as C and Lisp.

(iv) The notion of "database" is introduced. A database is a type system, together with a consistent set of objects (database instances) and a subtyping relationship satisfying some constraints.

We are currently working on some extensions of the model. The first one concerns object naming. Up to here, the only handle that a programmer has on a object is through the name of one of its types. So, to retrieve an object of the database, the programmer has to send a message to the extension of the type with some key as argument. Such a problem is introduced by persistency: in standard programming languages, we name objects using temporary variable names. Object names seem to be needed, and they have to be introduced in the model.

A second extension concerns the introduction of variables in the construction of types in order to model genericity (also called "parametric polymorphism") of types and methods. Genericity can be simulated with inheritance [Meyer 86], but in a heavy and non-intuitive way.

A third extension is to increase the modeling power of the model: a list constructor should be included in order to model ordered collections of data (it could be implemented as a recursive tuple type, but we would lose expressiveness). Finally, in this model, we made the simplifying assumption that the methods are not objects of the model. So methods have to be modeled as first order functions. It should be interesting to extend the model to treat methods as objects and to allow higher order methods.

Acknowledgements

Most of the ideas presented here were generated with F. Bancilhon. This paper also benefits from the careful reading of S. Abiteboul and our colleagues from *Altaïr*, in particular D. Excoffier. Thanks also go to P. Buneman, A. Borgida and D. DeWitt for the fruitful discussions we had on this model.

References

[Albano & al 85], "GALILEO: A Strongly Typed, Interactive Conceptual Language", A. Albano, L. Cardelli and R. Orsini, *ACM TODS, Vol 10 No. 2, March 85.*

[Andrews & Harris 87], "Combining Language and Database Advances in an Object-oriented Development Environment", T. Andrews and C. Harris, *Proc. OOPSLA, 1987.*

[Bancilhon and Khoshafian 86], "A Calculus for Complex Objects", F. Bancilhon, S. Khoshafian, *ACM PODS Conference, 1986*

[Bancilhon & al 87b], "The O_2 Object Manager Architecture", F. Bancilhon, V. Benzaken, C. Delobel, F. Velez, *Altaïr Technical Report,14/87, Nov, 87.*

[Banerjee & al 87], "Data Model Issues for Object-Oriented Applications", J. Banerjee & al., ACM TOOIS, Vol 5, No 1, Jan 1987.

[Bruce & Wegner 86], "An Algebraic Model of Subtypes in Object-Oriented Languages", K., B. Bruce, P. Wegner, *SIGPLAN notices V21 #40, October 86.*

[Cardelli 84], "A Semantics of Multiple Inheritance", L. Cardelli, *in Semantics of Data Types, Lecture notes in Computer Science, Vol 173 pp. 51-67, Springer Verlag, 84*

[Copeland & Maier 84], "Making Smalltalk a Database System", G. Copeland and D. Maier, *ACM-SIGMOD, 1984.*

[Copeland and Khoshafian 86], G. Copeland and S. Khoshafian, "Object Identity", *OOPSLA 86, Portland, Oregon, Sept 86.*

[Goldberg and Robson 83], "Smalltalk 80: The Language and its Implementation", A. Goldberg, D. Robson, *Addison-Wesley, Reading, Mass., 83*

[Kuper and Vardi 84], "A new Approach to Database Logic", G. M. Kuper, M. Y. Vardi, *ACM PODS Conference, Waterloo, Canada, 84*

[Meyer 86], "Genericity versus inheritance", B. Meyer, *OOPSLA 86, Portland, Sept 86.*

[Nixon & al 87], "Implementation of a Compiler for a Semantic Data Model: Experience with Taxis", B. Nixon et al, *ACM SIGMOD 1987.*

[Zdonik 84], "Object Management System Concepts,", S. Zdonik, *Proc. ACM SIGOA on Office Systems, Toronto, 1984.*