Supporting multiple roles through class hierarchies [1]

Luca Cabibbo

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria, 113 — 00198 Roma, Italy.
e-mail: cabibbo@infokit.ing.uniroma1.it

Abstract

Object Oriented Database Systems should allow their objects to play multiple roles and to change roles in their lifetime. We describe a data model, based on classes, is-a hierarchies, and complex types, that allows objects to belong to several classes, so to play multiple roles. Then we propose a declarative query language to access objects from multiple perspectives, and an update language to describe dynamics of objects.

1 Introduction

An Object–Oriented database describes a set of *objects*, corresponding to entities in the real world. An object in our database may be a *Student* named "Joe", to which we can refer by means of its *object identifier* (or *oid*) *Joe*. At the same time, *Joe* may be a *TennisPlayer*, or we may need to view *Joe* simply as a *Person*. Thus, it is natural for *Joe* to play different *roles* in the database. Furthermore, the dynamic nature of the database must allow objects to change roles in their lifetime, without losing their identity. At a certain time, *Joe* will cease to be a *Student*, will become a *Worker*, and then a *Married* person.

We model roles by means of classes. We associate with each class a type, describing structural properties of the objects in the class. Moreover, we model is-a relationships by means of inheritance hierarchies. They express subtyping and subset relationships between classes. In this framework, we do not require (in contrast with other proposals, which are more restrictive, such as [1, 5, 8]), for each object, the existence of a "most specific class", because this would imply, for each pair of classes with a nonempty intersection, a class containing exactly this intersection. In our example, we need classes *Person, Student, Worker, TennisPlayer*, and *Married*, and not *MarriedTennisPlayer*, probably a nonempty but meaningless class.

Currently, there are some proposals of data models (and systems) to support such a scenario of objects playing "multiple roles": *hierarchies* in IsaLog [3] and LOGIDATA+ [4], from which this paper inherits many ideas; *data specialization* in Galileo [2], *aspects* in [9], *object specialization* in [10], and *object migration* in [11].

In this paper we propose a data model, along with the associated query and update languages. We allow values of objects to have a complex structure, obtained by means of the set and tuple constructors.

The query language can express simple queries, similar to the conjunctive (or select–project–join) queries for relational database. For example, we ask for the names of the wifes of all persons having name "Paul" writing the query

$$Married(\text{OID} : X, name : \text{"}Paul\text{"}, spouse : S), Person(\text{OID} : S, name : N).$$

[1] Work partially supported by Systems & Management S.p.A.

Note how this query is very similar to the body of a Datalog rule. Actually, the result of this query is a set of *substitutions* over the variables X, S, and N or, equivalently, a set of tuples over these three variables. As a matter of fact, the significance of such a simple language is due to the possibility of extend it to a full Datalog–like environment. Moreover, we will see how this declarative language lets us view objects from multiple *perspectives* [10].

The update language expresses simple transformations on the database, called *atomic transactions*. We have five kinds of updates, which let us *create* new objects, *delete* them from the database, *specialize* and *despecialize* objects (that is, gain and lose roles), and *modify* values associated with objects. In this paper, we do not care the capability to express complex transactions, such as updates subject to conditions, or involving more than one object. Also in this case, it is possible to extend the language to make it a general update language.

The significance of both languages, in spite of their simplicity, is related to the possibility to point out some questions about multiple roles and complex structures, first of all the possibility to manage them by means of a class–based data model, in particular within a declarative framework.

This paper is organized as follows. Section 2 introduces the data model. The query and the update languages are defined in Section 3 and Section 4, respectively.

2 The Data Model

The data model is based on a clear distinction between scheme (the intensional level) and instance (the extensional level). Many features in this section are inherited from the LOGI-DATA+ model of data [4]. Intuitively, a scheme describes a set of classes, with their structural properties[2]. These are of two kinds: the type associated with objects of a class, and is-a relationships between classes. An instance describes a set of objects, with their values and memberships in classes.

We fix a countable set $\mathcal{A}$ of *attribute names* and a finite set $\mathcal{B}$ of *base type names*; associated with each $B \in \mathcal{B}$ there is a set of *base values* $\mathrm{V}(B)$.

A *scheme* is a triple $\mathbf{S} = (\mathbf{C}, \textsc{typ}, \textsc{isa})$, where

- $\mathbf{C}$ is a finite set of symbols called *class names*;
- $\textsc{typ}$ is a function defined on $\mathbf{C}$ such that for each symbol $C \in \mathbf{C}$, $\textsc{typ}(C)$ is a tuple type descriptor (see below);
- $\textsc{isa}$ is a partial order over $\mathbf{C}$ (with some conditions, see below).

In order to define the components of a scheme, we need the auxiliary notions of type descriptor and subtyping.

The *types* of a scheme $\mathbf{S}$ are defined as follows:

1. if B is a base type name in $\mathcal{B}$, then B is a type of $\mathbf{S}$;
2. if C is a class name in $\mathbf{C}$, then C is a type of $\mathbf{S}$;
3. if τ is a type of $\mathbf{S}$, then $\{\tau\}$ is also a type of $\mathbf{S}$, called a *set* type;
4. if $\tau_1, \ldots, \tau_k$, with $k \geq 0$, are types of $\mathbf{S}$ and $A_1, \ldots, A_k$ are distinct attributes in $\mathcal{A}$, then $(A_1 : \tau_1, \ldots, A_k : \tau_k)$ is also a type of $\mathbf{S}$ (*tuple* type); since the attributes are distinct, the order of components is immaterial.

[2]In this paper we do not address behavioural capability of objects in classes.

Moreover, we say that a type τ is a *subtype* of a type τ' (written $\tau \preceq \tau'$) if and only if at least one of the following conditions holds:

1. $\tau = \tau'$;
2. $\tau, \tau' \in \mathbf{C}$ and $(\tau, \tau') \in \text{ISA}$;
3. $\tau = \{\tau_1\}$ and $\tau' = \{\tau'_1\}$, with $\tau_1 \preceq \tau'_1$;
4. $\tau = (A_1 : \tau_1, \ldots, A_k : \tau_k, \ldots, A_{k+p} : \tau_{k+p})$, $\tau' = (A_1 : \tau'_1, \ldots, A_k : \tau'_k)$, with $k \geq 0$, $p \geq 0$, and for $1 \leq i \leq k$, it is the case that $\tau_i \preceq \tau'_i$;

With a few technical extensions the notion of subtyping induces a lattice over the types of a scheme **S**. We will see in a following section how a notion of *refinement* over values is the natural counterpart of subtyping over types.

The function TYP associates a *type* with each class name, indicating the set of possible values; the function VAL, to be defined later, will associate with each type descriptor the corresponding set of values:

1. *base types*, which denote predefined sets of values, such as integers, reals, or strings, are obviously types;
2. *classes* are types[3], because we want to be able to reference their elements (this is implemented by means of oid's);
3. 4. *set* and *tuple* types can be built from other types (their *components*), and their values are sets and tuples, respectively, of the component types.

As we said, the partial order ISA of a scheme **S** is subject to conditions, as follows:

1. if $(C_1, C_2) \in \text{ISA}$ (often written in infix notation, C_1 ISA C_2, and read "C_1 *is a subclass of* $\mathbf{C}_2$"), then TYP(C_1) is a subtype of TYP(C_2);
2. if C' and C'' have a *common ancestor* (that is, a class C such that C'ISAC and C''ISAC), and a *common attribute* A, then there is a common ancestor C_1 of C' and C'' such that A is an attribute of C_1;
3. if there are $C, C', C'' \in \mathbf{C}$ such that C ISA C' and C ISA C'', then C' and C'' have a common ancestor in $\mathbf{C}$; that is, *multiple inheritance* is allowed only beneath a common ancestor.

The partial order ISA has the usual role of *is-a relationship.* The condition of subtyping is imposed in order to guarantee that the elements of a subclass have a type "compatible" with that of the superclass. The definition of subtyping for tuple types expresses the idea that a tuple t belongs to a tuple type τ if it has *at least* the components of τ, and possibly some more. The condition about common attributes insures that each attribute, within a hierarchy, has a unique uppermost class defining it and its "upper type". In this way, possible redefinitions of the type of an attribute are forced to happen in a type–compatible fashion[4]. The condition concerning multiple inheritance implies that each class belongs to a unique hierarchy, in such a way that each distint hierarchy corresponds to a taxonomy of the real world.

[3]This means that classes can be used in building other types, which may therefore refer to the objects in the classes.

[4]This hypothesis is not too restrictive. In fact it states that each attribute in a hierarchy has a unique meaning, disallowing, for example, to have in the *Person-Student-Worker* hierarchy, an attribute *dept* in both *Student* (meaning the thesis department) and *Worker* (meaning the working seat).

We say that a class $C \in \mathbf{C}$ is a *root class* if there is no class $C' \in \mathbf{C}$ (different from C) such that C ISA C'. In other words, the root classes of a scheme are those classes without ancestors. It is apparent that each root class identifies a distinct *hierarchy*, that is, the root class and its subclasses.

Furthermore, if we suppose that, for each class C of a scheme $\mathbf{S} = (\mathbf{C}, \text{TYP}, \text{ISA})$ there is no an attribute which appears more than once in $\text{TYP}(C)$[5], then we can define a function A-TYP associated with TYP, from attributes and classes to type descriptors, such that $\text{A-TYP}(A, C)$ is the type descriptor of component A in $\text{TYP}(C)$.

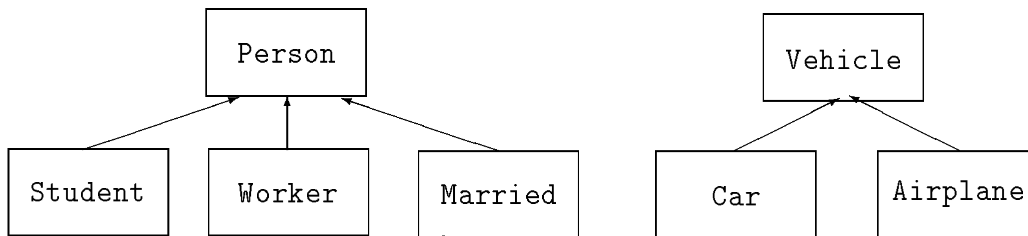

Figure 1

Example 1 Figure 1 shows two class hierarchies of a scheme, with roots *Person* and *Vehicle*. For this scheme, considering only the *Person* hierarchy, we have:

- $\text{TYP}(Person) = (name : string)$;
- $\text{TYP}(Student) = (name : string, school : string)$;
- $\text{TYP}(Worker) = (name : string, salary : integer)$;
- $\text{TYP}(Married) = (name : string, spouse : Married)$.

ISA is the reflexive and transitive closure of the relation that contains the pairs (*Student, Person*), (*Worker, Person*), and (*Married, Person*).

This scheme satisfies the various conditions.

As in every other data model, the *scheme* gives the structure of the possible *instances* of the database. Instances over schemes are built, by means of the set and tuple constructors, from the elementary values, which come from the value-sets associated with the base types, and *object identifiers (oid's)*, which are used as indirect references to elements of classes.

Now, we devote our attention to the definition of instance. The first step is the definition of the value-sets associated with types. We assume the existence of a countable set $\mathcal{O}$ of oid's. With each type descriptor τ we can associate the set $\text{VAL}(\tau)$ of its possible values, called its *value-set* :

1. if $\tau = B \in \mathcal{B}$, then $\text{VAL}(\tau) = \text{V}(B)$;
2. if τ is a class name $C \in \mathbf{C}$, then its value-set is the set of the oid's $\mathcal{O}$;
3. if τ is a set type, that is, it has the form $\{\tau'\}$, then its value-set is the set of the finite subsets of $\text{VAL}(\tau')$;
4. if τ is a tuple type $(A_1 : \tau_1, \ldots, A_k : \tau_k)$, then

$$\text{VAL}(\tau) = \{t : \{A_1, \ldots, A_k\} \rightarrow \cup_{i=1}^{k} \text{VAL}(\tau_i) \mid t(A_i) \in \text{VAL}(\tau_i), \text{for } 1 \leq i \leq k\}$$

that is, the set of all possible tuples over $A_1, \ldots, A_k$ of the correct type.

[5]That is, we disallow situations like $\text{TYP}(C) = (A : (A : \ldots))$.

With respect to classes, it is important to note that their value-sets only contain oid's; the actual values of the elements of the classes are defined by another function that associates a value (of the correct type) with each oid; in this way, it is possible to implement indirect references to objects and other features such as object sharing. Also, for each class, the value-set is the set of *all* possible oid's: essentially, we can say that oid's are not typed, and so they allow the identification of an object regardless of its type; oid's become typed when they belong to classes.

As we will see later, there is a need to handle *incomplete information*, allowing for values to be undefined. In this framework we assume the existence of a polymorphic *null value* $\perp$, denoting *unknown values*, and extend all value-sets with this null value.

An *instance* $\mathbf{s}$ of a scheme $\mathbf{S} = (\mathbf{C}, \textsc{typ}, \textsc{isa})$, is a pair $\mathbf{s} = (\mathbf{c}, \mathbf{o})$, where:

- $\mathbf{c}$ is a function that associates with each class name $C \in \mathbf{C}$ a finite set of oid's: $\mathbf{c}(C) \subseteq \mathcal{O}$, with the following conditions:

  1. if $C_1 \textsc{isa} C_2$, then $\mathbf{c}(C_1) \subseteq \mathbf{c}(C_2)$;
  2. if $\mathbf{c}(C_1) \cap \mathbf{c}(C_2) \neq \emptyset$ then C_1 and C_2 have a common ancestor;

- $\mathbf{o}$ is a (partial) function that associates oid's in classes with tuples, as follows. For each $o \in \mathcal{O}$, let us consider the *role set* $\textsc{classes}(o)$ of o, which is the set of classes containing o: $\textsc{classes}(o) = \{C \mid C \in \mathbf{C}, o \in \mathbf{c}(C)\}$. Then, for each o, if $\textsc{classes}(o)$ is empty, then $\mathbf{o}(o)$ is undefined, otherwise it is a value from the value set of the tuple type that is the greatest lower bound (according to the lattice induced by subtyping) of the types of the classes in $\textsc{classes}(o)$;

- if a tuple has an attribute A whose type is a class $C \in \mathbf{C}$, then the value of the tuple over A is an oid in $\mathbf{c}(C)$ (this condition avoids "dangling references");

- for each class C, for each attribute A of C such that $\textsc{a-typ}(A, C)$ is in the form $\{(\ldots)\}$ (that is, a set of tuples, corresponding to a *relation*), let us consider the class C' which is the uppermost class in C hierarchy defining A. We require that the set of attributes appearing in $\textsc{a-typ}(A, C')$ forms a *key* (in the sense of relational database theory) for the set of tuples A of each object in $\mathbf{c}(C)$[6].

Therefore, an instance is defined by means of the functions $\mathbf{c}$ and $\mathbf{o}$.

The function $\mathbf{c}$ associates with each class name a set of oid's, with the conditions that the partial order $\textsc{isa}$ corresponds to subset relations between the involved classes, and that two classes have a nonempty intersection only if they have a common ancestor in the $\textsc{isa}$ hierarchy; in this way, for each object, there is a "most general class", that is, the root class of the hierarchy to which it belongs. This completes, at instance level, the multiple inheritance condition at scheme level: objects, as well as classes, are partitioned into distinct taxonomies. This is useful in many applications, because the same happens for real world entities. On the contrary, as opposed to what happens in other models [1, 5, 8], we do not require, for each object, the existence of a "most specific class", because, in many cases, this could lead to a proliferation of almost meaningless classes. Note that the function $\mathbf{c}$ over classes is equivalently represented by means of the function $\textsc{classes}$ over oid's, which associates with each object its *role set* [11].

The function $\mathbf{o}$ associates values with oid's belonging to classes, with the condition that, for each class C, for each oid o in the instantiation $\mathbf{c}(C)$ of C, the value $\mathbf{o}(o)$ has a type that

[6]That is, for each $o \in \mathbf{c}(C)$, the component A of $\mathbf{o}(o)$ contains no different tuples t' and t'' having the same values over the attributes occurring in $\textsc{a-typ}(A, C')$.

is a subtype of the type $\text{TYP}(C)$ of the class. In this way we allow objects to play different roles having, at the same time, values of a well-defined type.

The condition about keys needs additional comments. We will give them in Section 4, after the introduction of the query and the update languages. Intuitively, this condition is imposed in order to guarantee well–definedness of sets of tuples for components of objects playing multiple roles.

Example 2 An instance $\mathbf{s} = (\mathbf{c}, \mathbf{o})$ over the scheme as in Example 1, could be defined as follows:

- $\mathbf{c}$ is the function
  - $\mathbf{c}(Person) = \{Joe, Meg, Paul\}$;
  - $\mathbf{c}(Student) = \{Joe, Paul\}$;
  - $\mathbf{c}(Worker) = \{Paul\}$;
  - $\mathbf{c}(Married) = \{Joe, Meg\}$;
- $\mathbf{o}$ is the function
  - $\mathbf{o}(Joe) = (name : \text{“}Joe\text{”}, school : \text{“}Medicine\text{”}, spouse : Meg)$;
  - $\mathbf{o}(Meg) = (name : \text{“}Margaret\text{”}, spouse : Joe)$;
  - $\mathbf{o}(Paul) = (name : \text{“}Paul\text{”}, salary : 100K)$.

Let us note that symbolic names for oid's (*Joe, Paul,* and *Meg* in the example above) carry no information beyond object identity, being unrelated to values of corresponding objects. Moreover, usually oid's are considered not visible to the users. We use symbol constants only for the sake of presentation, ignoring details concerning oid's implementation or their internal representation.

Example 3 Let us consider another example, more complex than the previous one. It will be useful in the following. The scheme $\mathbf{S}_R$ contains three classes, *Reader, A-Reader*, and *P-Reader*, where *A-Reader* ISA *Reader* and *P-Reader* ISA *Reader*. Each *Reader* has an associated set of *books*, described by means of a *title*. Moreover, an *A-Reader* knows *authors* of its books as well, whereas a *P-Reader* knows their *publishers*. Thus, we have in our scheme:

- $\text{TYP}(\textit{Reader}) = (books : \{(title : string)\})$;
- $\text{TYP}(\textit{A-Reader}) = (books : \{(title : string, author : string)\})$;
- $\text{TYP}(\textit{P-Reader}) = (books : \{(title : string, publisher : string)\})$.

The scheme satisfies the subtyping condition.

We have in this scheme a set of tuples, corresponding to the component *books*, having *Reader* as the uppermost class defining it. Thus, we require *title*, the only attribute appearing in $\text{A-TYP}(\textit{books}, \textit{Reader}) = \{(title : string)\}$, to be a key for each *relation* bound with *books*.

An instance $\mathbf{s}_R = (\mathbf{c}, \mathbf{o})$ over $\mathbf{S}_R$ could be the following:

- $\mathbf{c}(\textit{Reader}) = \{r_1, r_2, r_3\}$;
- $\mathbf{c}(\textit{A-Reader}) = \{r_2, r_3\}$;
- $\mathbf{c}(\textit{P-Reader}) = \{r_2\}$.
- $\mathbf{o}(r_1) = (books : \{(title : \text{“}Siddharta\text{”}), (title : \text{“}The\ Prophet\text{”}\})$;
- $\mathbf{o}(r_2) = (books : \{(title : \text{“}Momo\text{”}, author : \text{“}Ende\text{”}, publisher : \text{“}P.Press\text{”})\})$;
- $\mathbf{o}(r_3) = (books : \{(title : \text{“}Ulysses\text{”}, author : \text{“}Joyce\text{”})\})$.

3 The Query Language

In this section we propose the syntax and semantics of a query language for the data model proposed in the previous section. This language is declarative, allowing for queries having a

structure similar to the body of a Datalog rule. In this way we can express conjunctive queries in a simple way. The semantics of a query is given from the (typed) substitutions over the occurring variables occurring that let the query be satisfied by the actual instance.

We present the language briefly defining all needed concepts, from term to atom and query, from typed substitution to satisfaction, as usual in logic programming frameworks.

Let a scheme $\mathbf{S} = (\mathbf{C}, \text{TYP}, \text{ISA})$ be fixed. Also, let V be a countable set of *variables*.

The terms of the language are obtained from constants, oid's, variables, and recursive applications of set and tuple constructors. The type of a term depends on its form and, when it contains variables or oid's, on the context (that is, the literal) in which it occurs. We will see how, for variables and oid's, we can obtain a type from their use in atoms. The *terms* of the language, with the corresponding types[7], are

1. if $v \in V$ is a variable of type τ, then v is a (variable) term of type τ;
2. if $d \in \text{VAL}(B)$ (with $B \in \mathbf{B}$) is a constant of type B, then d is a (constant) term of type B;
3. if $t_1, \ldots, t_k$ are terms of type τ, then $\{t_1, \ldots, t_k\}$ is a (set) term of type $\{\tau\}$;
4. if $t_1, \ldots, t_k$ are terms of type $\tau_1, \ldots, \tau_k$, and $A_1, \ldots, A_k$ are distinct attribute names in $\mathcal{A}$, then $(A_1 : t_1, \ldots, A_k : t_k)$ is a (tuple) term of type $(A_1 : \tau_1, \ldots, A_k : \tau_k)$;
5. if o is an oid in $\mathcal{O}$, of type τ, then o is an (oid) term of type τ.

Terms of the language are used to build expressions, with set theoretic operators on sets, and the *dot* operator on tuples[8]. The *expressions* of the language may have the following forms:

1. each term t of type τ is an expression of type τ;
2. if E and E' are set expressions of type $\{\tau\}$, then $E \cup E'$, $E - E'$ and $E \cap E'$ are expressions of the same type $\{\tau\}$;
3. if E is a tuple expression of type $(A_1 : \tau_1, \ldots, A_k : \tau_k)$, then, for each $1 \leq i \leq k$, $E.A_i$ is an expression of type τ_i.

Atoms of the language are used to express (atomic) conditions on objects in the database and relationships between terms and expressions. The *atoms* of the language may have the following forms:

1. *class atoms*: $C(\text{OID} : t_0, A_1 : t_1, \ldots, A_k : t_k)$ where C is a class name in $\mathbf{C}$, with $\text{TYP}(C) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$, t_0 is a term of a type C, and for each $1 \leq i \leq k$, t_i is a term of type τ_i;
2. *equality atoms*: $t = E$, with t term of type τ and E expression of the same type τ;
3. *membership atoms*: $t \in E$, with t term of type τ and E expression of type $\{\tau\}$.

We say that a variable x is *range restricted* in an atom A if it satisfies one of the following conditions:

- A is a class atom in which x occurs as a term;

---

[7] The type of variable and oid terms has to be defined later.

[8] It is possible to extend the expressions including other built–in operators, such arithmetic ones on integers, and so on.

- A is either an equality or a membership atom in which x occurs as the term in the left side.

A variable is *range restricted* in a set of atoms if it is range restricted in at least one of these atoms.

A *query* Q is a nonempty set of atoms $A_1, \ldots, A_k$, containing only range restricted variables. The set of variables that occurr in a query Q is called the *domain* of Q.

In the above definition of atoms, each term is required to have a correct, specified type. For value–based terms, these conditions are constraints; instead, for occurrences of variable and oid terms, we have their type characterization just as a consequence of the above conditions. This is the justification of the asymmetry between terms and expressions in equality and membership atoms. Moreover, this happens without loss of generality.

Example 4 Let us consider the following query over the same scheme as in Example 3:

$$\textit{A-Reader}(\text{OID} : R, \textit{books} : B), A \in B, \text{``}Hesse\text{''} = A.author.$$

This query retrieves all *A-Reader* having at least a book written from Hesse. All variables in the query are typed:

- R is bound with the oid term of the class atom *A-Reader*, so R has type *A-Reader*;
- B is bound with attribute *books* of *A-Reader*, so B has type $\{(\textit{title} : \textit{string}, \textit{author} : \textit{string})\}$;
- A is bound within a membership atom, so it has the type of the component of the (set) expression B, that is, A has type $(\textit{title} : \textit{string}, \textit{author} : \textit{string})$;
- moreover, in the equality atom, the term *"Hesse"* has type *string*, as well as expression $A.author$.

It is apparent that a variable x may occurr several times within a query Q. Moreover, each occurrence x^i of x in Q may be with a different type τ^i. The *type of a variable x within a query Q* is defined as the greatest lower bound of the types of the occurrences of x in Q.

Example 5 Let Q be the query over the same scheme as in Example 3:

$$\textit{A-Reader}(\text{OID} : R, \textit{books} : B), \textit{P-Reader}(\text{OID} : R, \textit{books} : B), A \in B.$$

This query retrieves information about books of readers that are both *A-Reader* and *P-Reader*. All variables in the query are typed:

- The first occurrence of B is as a term of type $\{(\textit{title} : \textit{string}, \textit{author} : \textit{string})\}$;
- The second occurrence of B is as a term of type $\{(\textit{title} : \textit{string}, \textit{publisher} : \textit{string})\}$;
- The third occurrence of B is free. Thus, the type of B within Q is the greatest lower bound of $\{(\textit{title} : \textit{string}, \textit{author} : \textit{string})\}$ and $\{(\textit{title} : \textit{string}, \textit{publisher} : \textit{string})\}$, that is, $\{(\textit{title} : \textit{string}, \textit{publisher} : \textit{string}, \textit{author} : \textit{string})\}$;
- Finally, the type of A is $(\textit{title} : \textit{string}, \textit{publisher} : \textit{string}, \textit{author} : \textit{string})$.

Intuitively, the result of a query is the set of ground typed substitutions over its domain, that makes true all its atoms. Thus, to define the semantics of a query, we need the important concept of satisfaction of a ground atom — that is, an atom containing no variables. To introduce it, we need some other auxiliary, preliminary notions: refinement, evaluation of an expression, and typed substitution.

The notion of subtyping, defined over types, has *refinement* as a natural counterpart over values (that is, instances of types). With respect to values of base types and oid's (which are *atomic* values) refinement coincides with equality, so the definition is really significant with respect to sets and tuples (which are *structured* values). We say that a value t of type τ is a refinement of a value t' of type τ' (in symbols $t \preceq t'$) if and only if at least one of the following conditions holds:

1. τ and τ' are base types in $\mathcal{B}$, $\tau \preceq \tau'$, and $t = t'$ are the same constant;

2. τ and τ' are class names in $\mathbf{C}$, $\tau \preceq \tau'$, and $t = t'$ are the same oid;

3. τ and τ' are set types, $\tau \preceq \tau'$, for each component t_1 of t there exists a component t'_1 of t' such that $t_1 \preceq t'_1$, and for each component t'_1 of t' there exists a component t_1 of t such that $t_1 \preceq t'_1$;

4. τ and τ' are tuple types, $\tau \preceq \tau'$, with $\tau = (A_1 : \tau_1, \ldots, A_k : \tau_k, \ldots, A_{k+p} : \tau_{k+p})$, $\tau' = (A_1 : \tau'_1, \ldots, A_k : \tau'_k)$, $t = (A_1 : t_1, \ldots, A_k : t_k, \ldots, A_{k+p} : t_{k+p})$, $t' = (A_1 : t'_1, \ldots, A_k : t'_k)$, and for each $1 \leq i \leq k$, it is the case that $t_i \preceq t'_i$.

Informally, a value t is a refinement of a value t' if their types are in a subtyping relationship, and the "restriction" of t to the type of t' equals t'. Also in this case, we can extend the set of values allowed in the language, so to have a lattice induced by the notion of refinement. In this way we are able to talk about the greatest lower bound of a set of values.

The notion of evaluation of an expression is defined by means of the function *eval*, from (ground) expressions to (ground) terms, which gives a meaning to the various built–in functions introduced. Given a (ground) expression E, we define $eval(E)$ as follows:

1. if E is a term t, $eval(E) = t$;

2. if E_1 and E_2 are expressions, and $\Phi \in \{\cup, \cap, -\}$, then $eval(E_1 \Phi E_2) = eval(E_1) \Phi eval(E_2)$, assuming for Φ the usual set theoretic meaning;

3. if $eval(t) = (A_1 : t_1, \ldots, A_k : t_k)$, then, for each $1 \leq i \leq k$, $eval(t.A_i) = t_i$.

Let a query Q be fixed. A *typed substitution* θ is a function from variables to terms that maps variables in V (with type τ within Q) to ground terms of the corresponding type τ. The notion of typed substitution is extended in the natural way to atoms and sets of atoms, and then to queries.

We say that an instance $\mathbf{s} = (\mathbf{c}, \mathbf{o})$ *satisfies* a ground atom A if:

1. A is a class atom $C(\text{OID} : o, A_1 : t_1, \ldots, A_k : t_k)$, $o \in \mathbf{c}(C)$, and $\mathbf{o}(o) \preceq (A_1 : t_1, \ldots, A_k : t_k)$;

2. A is an equality atom $t = t'$, and $t \preceq eval(t')$;

3. A is a membership atom $t \in t'$, and exists an element $t'' \in eval(t')$ such that $t \preceq t''$.

Let us note how the definition of satisfaction for class atoms refers to a weak requirement on values, refinement rather than equality, because, in general, we do not (need to) know the exact type of an object within a hierarchy. In this way, class atoms act as "cast" operators in C: variables occurring in them give rise to substitutions with values of a type which is the type of the corresponding component in the class. For example, if we suppose that Q is the atom $C(\text{OID} : o, a : A, b : B)$, where o is an oid in a class C, with $\mathbf{o}(o)$ equals $(a : 1, b : (b_1 : 11, b_2 : 12))$, and $\text{TYP}(C)$ equals $(a : integer, b : (b_1 : integer))$, then the types of the variables A and B within Q are *integer* and $(b_1 : integer)$, respectively, and Q is satisfied from a typed substitution θ such that $\theta(A) = 1$ and $\theta(B) = (b_1 : 11)$[9].

Similarly, we say that an instance $\mathbf{s}$ *satisfies* a set of ground atoms $A_1, \ldots, A_k$ if for each $1 \leq i \leq k$, $\mathbf{s}$ satisfies A_i.

[9]A substitution that gives a value $(b_2 : 12)$ for B seems to satisfy the atom as well. But we cannot consider it, because it is not *typed*.

Then, the *semantics* of a query $Q = A_1, \ldots, A_k$ is a function QUERY$[Q]$ from instances to sets of typed substitutions over the domain of Q, defined as follows:

$$\text{QUERY}[Q](\mathbf{s}) = \{\theta \text{ typed substitution over the domain of } Q \mid \mathbf{s} \text{ satisfies } \theta(Q)\}.$$

The query language proposed needs some additional comments: we propose them resorting to an example. Given the query

$$Married(\text{OID} : X, name : \text{"}Paul\text{"}, spouse : S), Person(\text{OID} : S, name : N).$$

we can think of it as the body of a Datalog rule. An atom expresses a condition over objects in the corresponding class, an attribute bound with a constant means selection, occurrences of the same variable in different places mean a join condition, and a comma means conjunction of conditions. Actually, the result of this query is a set of typed substitutions over the variables X, S, and N or, equivalently, a set of tuples over these three variables. As a matter of fact, the significance of such a simple language is due to the possibility of extend it to a full Datalog-like environment [3, 4]. Because of the presence of complex objects and object identity, the extension is not straightforward. Intuitively, the computation of a query in this language is assimilable to the determination of the *valuation domain* associated with a rule within a computational step of the semantics of an IQL–like program [1].

Furthermore, this language lets us view objects from multiple *perspectives* [10]. In fact it is possible to test membership of an object in several classes, checking for its role set. For example, if we want all objects that are both student and worker, we can simply query:

$$Student(\text{OID} : X, school : S, \ldots), Worker(\text{OID} : X, salary : W \ldots).$$

In this way, a substitution satisfies the query only if the oid associated with X is in both the required classes. At the same time, we can obtain (from substitutions) values for all the needed components of an object, even if they are represented in different classes, such as *school* and *salary* in example above. Moreover, because of the use of typed substitutions and refinement, information split among classes in hierarchies can be rejoined, as in Example 5.

4 The Update Language

We now propose the skeleton of an update language for this data model. The main idea is to have, at least, a set of "operations" able to manipulate single objects, performing on them the various possible trasformations over values and roles. These are called atomic updates. Some concepts in this section are similar in spirit to those in [11], where the subject concerns on *flat* objects, while we are dealing with hierarchies of *complex* objects.

In this paper we do not care the ability to express complex transactions, such as conditional updates, sequencing and iterations. Because of the requirement to act over single objects, with specified values, we refer in this section only to *ground* terms. Also in this case, we must resort to *symbol constants* to represent oid's.

Given a scheme $\mathbf{S} = (\mathbf{C}, \text{TYP}, \text{ISA})$, suppose that C is a class name in $\mathbf{C}$, with $\text{TYP}(C) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$, t_0 is an oid term, and for each $1 \leq i \leq k$, t_i is a ground term of type τ_i. The *atomic updates* of the language may have the following forms:

1. *create* $C(\text{OID} : t_0, A_1 : t_1, \ldots, A_k : t_k)$;
2. *specialize* $C(\text{OID} : t_0, A_1 : t_1, \ldots, A_k : t_k)$;
3. *delete* t_0;

4. *despecialize* $C(\text{OID} : t_0)$[10];

5. *modify* $C(\text{OID} : t_0, A_j : t_j)$, with $1 \leq j \leq k$.

We now informally describe effects of atomic updates. The semantics of each update U is a partial function EFFECT$[U]$ from instances to instances. It is partial because of the possibility to express invalid updates[11], such as creations of objects already in the database, or incorrect specializations. Let us consider an instance $\mathbf{s} = (\mathbf{c}, \mathbf{o})$.

1. EFFECT[*create* $C(\text{OID} : o, A_1 : t_1, \ldots, A_k : t_k)$]$(\mathbf{s})$ is defined iff o is an oid not used in $\mathbf{s}$. The resulting instance contains a new object, with oid o, such that o belongs to C as well as to each of its superclasses, and $\mathbf{o}(o)$ equals $(A_1 : t_1, \ldots, A_k : t_k)$;

2. EFFECT[*specialize* $C(\text{OID} : o, A_1 : t_1, \ldots, A_k : t_k)$]$(\mathbf{s})$ is defined iff o is an oid in the hierachy which C belongs to, and $(A_1 : t_1, \ldots, A_k : t_k)$ is a refinement of the restriction of $\mathbf{o}(o)$ to the type which is the greatest lower bound of the types of all superclasses of C. In the resulting instance, o belongs to C and each of its superclasses, with a value that is the greatest lower bound, according to the lattice induced by refinement, of $(A_1 : t_1, \ldots, A_k : t_k)$ and $\mathbf{o}(o)$;

3. EFFECT[*delete* o]$(\mathbf{s})$ is defined iff the oid o corresponds to an object in $\mathbf{s}$, and there is no object in $\mathbf{s}$ referring to o. In the resulting instance, o is not used, meaning that o does not belong to any class, and $\mathbf{o}(o)$ is undefined;

4. EFFECT[*despecialize* $C(\text{OID} : o)$]$(\mathbf{s})$ is defined iff o is an oid in $\mathbf{c}(C)$, and there is no object in $\mathbf{s}$ referring to o as an object in neither C nor any of its subclasses. In the resulting instance, o belongs to the same classes as in $\mathbf{s}$, except for C and its subclasses. The new value for o is the restriction of $\mathbf{o}(o)$ to the type associated with the new role set of o;

5. EFFECT[*modify* $C(\text{OID} : t_0, A_j : t_j)$]$(\mathbf{s})$ is defined iff o is an oid in $\mathbf{c}(C)$. In the resulting instance, o belongs to the same classes as in $\mathbf{s}$, with the same value as in $\mathbf{s}$, except for the component A_j, whose value is changed according to t_j.[12]

Let us clarify the definitions by means of examples.

- *create* generates a new object: for example, *create Student(*OID*: Joe, name:"Joe", school:"Medicine")* specifies the creation of a new object, with oid *Joe*, in the classes *Student* and *Person* (which is the only *Student* superclass), with value *(name:"Joe", school:"Medicine")*.

- *specialize* gains new roles to an existing object, refining its value while preserving old values for previous roles: *specialize Worker(*OID*: Joe, name: "Joe", salary: 100K)* states that now *Joe* belongs to the class *Worker* as well, and its new value is *(name:"Joe", school:"Medicine", salary: 100K)*.

- *despecialize* loses roles, adapting values to a correct type: *despecialize Student (*OID*: Joe)* states that *Joe* is no more a *Student*, belonging now only to the classes *Person* and *Worker*, with value *(name:"Joe", salary: 100K)*.

---

[10]In [11] this update is called *generalize*.

[11]Otherwise, we can choose as the semantics of invalid updates the identity transformation.

[12]This definition leaves few points open. They will be clarified in the following.

- *modify* changes a value of an object: *modify Worker(*OID*: Joe, salary: 120K)* makes the value for *Joe* to be *(name:"Joe", salary: 120K).*

- *delete* removes an existing object: *delete Joe* totally removes the object *Joe* from the database. Note how this update is redundant, because it can be replaced from *despecialize Person(*OID*: Joe)*. At the same time, *delete* is more expressive than *despecialize*, since it does not require the specification of the root of the hierarchy which an object belongs to.

The various conditions for the existence of the semantics of atomic updates follow from similar requirements on instances.

- Each object, identified by an oid, has a unique value of the correct type. If o is already an oid of a *Student*, we cannot say *create Person(*OID*: o, ...)*, because we require, for the creation of a new object, the specification of an oid currently unused in the database.[13] Furthermore, if the actual *name* corresponding to o is *"Mark"*, we cannot say *specialize Worker(*OID*: o, name: "Joe",...)*, because in this way we are trying to modify an existing object value. This attempt would violate the refinement criterion, giving rise to an *incorrect specialization.*

- In an instance, there are no dangling references to objects not in the corresponding classes: if the School of Medicine object refers to *Joe* as one of its students, we can apply neither *delete* nor *despecialize Student* to the object *Joe.*

- Each object may belong to several classes, but all within the same hierarchy; thus we cannot say *specialize Car(*OID*: Joe, ...)*, because *Joe* is a *Person* (as a most general class) and not a *Vehicle.*

Before concluding this section, we propose two comments that are useful in order to justify the *key* condition imposed over instances in Section 2. The first is related to the refinement criterion for correct specializations, while the second concerns the semantics of *modify* over set–valued attributes.

Remark 1 Consider the scheme $\mathbf{S}_R$, and the instance $\mathbf{s}_R$ over it, in Example 3. In $\mathbf{s}_R$, we have an object r_3 belonging to *A-Reader*, but not to *P-Reader*, with value $(books : \{(title : \text{"}Ulysses\text{"}, author : \text{"}Joyce\text{"})\})$.

Let us now suppose we want to specialize r_3 to be an object in *P-Reader* as well. To do this, we must say *specialize P-Reader*$(\text{OID} : r_3, books : \ldots)$, specifying a value for the component *books*. This value must be of the correct type in *P-Reader*, that is $\{(title : string, publisher : string)\}$. We know that actually r_3 has a value for *books* containing only one element, which has value *"Ulysses"* for *title*. To specify a correct specialization, we must confirm these facts: a value like $(books : \{(title : \text{"}Ulysses\text{"}, publisher : \text{"}A.Print\text{"})\})$ achieves this condition, giving to r_3 the new value $(books : \{(title : \text{"}Ulysses\text{"}, publisher : \text{"}A.Print\text{"}, author : \text{"}Joyce\text{"})\})$, while $(books : \{(title : \text{"}King\ Lear\text{"}, publisher : \text{"}A.Print\text{"})\})$ does not, resulting in an incorrect specialization. Actually, the specialization of an object o in a class C has effect if and only if we specify values for o seen as an object in C that are coherent with its previous values. For simple values (constants and oid's), we must use the old ones; for tuples and sets, we must use refining values. In particular, for sets, it seems to be necessary to specify values having the same cardinality as the old ones.

This intuitive requirement over sets is achieved by means of the key condition over instances. In the proposed scheme, *books* is the only attribute involved in the condition, and we

[13] In this framework we do not address the matter of *invention* (or *distribution*) of new oid's.

have seen that *title* must be a key for objects in *Reader* hierarchy. As a consequence of this hypothesis, we cannot have an object r_4 in *Reader* and *P-Reader* with value ($books : \{(title : \text{``}Fables\text{''}, publisher : \text{``}A.Print\text{''}), (title : \text{``}Fables\text{''}, publisher : \text{``}P.Press\text{''})\}$). We have two motivations for this requirement. First, the value for r_4 looked as an object in *Reader* is ($books : \{(title : \text{``}Fables\text{''})\}$, that is, we "lose" a tuple from *books*, going against the intuition of cardinality invariance. Second, and more important, if we try to *specialize* $r_4$ to belong to *A-Reader* with a value ($books : \{(title : \text{``}Fables\text{''}, author : \text{``}Aesop\text{''})\}$), we are not able to understand if the value *"Aesop"* has to be associated with the *"A.Print"* tuple, the other one, or both of them. A similar uncertainty happens if we try to do a specialization with a value ($books : \{(title : \text{``}Fables\text{''}, author : \text{``}Aesop\text{''}), (title : \text{``}Fables\text{''}, author : \text{``}Grimm\text{''})\}$), because we could not infer relationships between authors and publishers. On the contrary, if we accept the key condition, we cannot encounter ambiguities like these.

Remark 2 When we try to define the semantics of the *modify* update, we may have a problem, due to the possibility of changing a value for a component of an object in a class, while the object is playing multiple roles, so to define the type of that component in multiple ways. Also in this case, the main problem is related to sets (of tuples). Let us consider again the same scheme as in Example 3, and an instance **s** defining an object r_5 in class *A-Reader*, with value ($books : \{(title : \text{``}Siddharta\text{''}, author : \text{``}Hesse\text{''}), (title : \text{``}The\ Prophet\text{''}, author : \text{``}Gibran\text{''})\}$). We are going to discuss the semantics of the three following updates, all applied to **s**, concerning on modifications of component *books* of r_5 thought as an object in *Reader*:

1. $modify\ Reader(\text{OID} : r_5, books : \{(title : \text{``}Siddharta\text{''}), (title : \text{``}The\ Prophet\text{''}), (title : \text{``}Fables\text{''})\})$.
2. $modify\ Reader(\text{OID} : r_5, books : \{(title : \text{``}Siddharta\text{''})\})$.
3. $modify\ Reader(\text{OID} : r_5, books : \{(title : \text{``}Siddharta\text{''}), (title : \text{``}Fables\text{''})\})$.

What is the semantics of these updates ? If we look at r_5 as an object in *Reader*, we see that the set *books* associated with r_5, in the input instance, contains two elements, namely ($title : \text{``}Siddharta\text{''}$) and ($title : \text{``}The\ Prophet\text{''}$).

1. The first update states that, in the output instance, the set *books* of r_5 has to contain three elements. Furthermore, it seems that two of these elements in the input instance appear in the output one, as well. Thus, the semantics for the first update is to insert a new tuple in *books*, having value *"Fables"* for *title*. But we know that r_5 belongs to *A-Reader*, and we need a new value for r_5 as an object in *A-Reader*. A possible solution, in this case, is to extend the value of new components in a set with null values, when required. Following this idea, in the output instance, we would have for r_5 a new value ($books : \{(title : \text{``}Siddharta\text{''}, author : \text{``}Hesse\text{''}), (title : \text{``}The\ Prophet\text{''}, author : \text{``}Gibran\text{''})(title : \text{``}Fables\text{''}, author : \perp)\}$).
2. We can think to a similar approach when we remove an element from a set. In the second update, we understand that the tuple with value *"The Prophet"* has to disappear from component *books* of r_5. Thus, we would have a value ($books : \{(title : \text{``}Siddharta\text{''}, author : \text{``}Hesse\text{''})\}$).
3. An effect of the third update is to let the cardinality of *books* be an invariant. Does it mean that we want to modify the *title* in tuple ($title : \text{``}The\ Prophet\text{''}, author : \text{``}Gibran\text{''}$) from *"The Prophet"* to *"Fables"* ? This would be a rather strange choice, and we do not consider it. A better interpretation is indeed composed of the deletion of tuple ($title : \text{``}The\ Prophet\text{''}$, $author : \text{``}Gibran\text{''}$) from *books*, followed by the insertion

of a new tuple $(title : \text{"}Fables\text{"}, author : \bot)$. Thus, the result of this update gives to r_5 a value $(books : \{(title : \text{"}Siddharta\text{"}, author : \text{"}Hesse\text{"}), (title : \text{"}Fables\text{"}, author : \bot)\})$.

Let us note how the above semantics for the three proposed updates, which are representative examples for modifications of set–values, are justifiable resorting to the key hypothesis: within a modify update referring to a set of tuples, we identify two tuples to be the same if they have the same values for the key defined as above. Without key condition we could not be able to give (justifying it) a semantics for the three updates.

As a final comment, let us outline other difficulties related to updates in this data model.

Consider a scheme with classes *Reader, A-Reader, Book*, and *A-Book*, where *A-Reader* ISA *Reader* and *A-Book* ISA *Book*. The types of classes *Reader* and *A-Reader* are *(books :* {*Book*}*)* and *(books :* {*A-Book*}*)*, respectively. This scheme models a reality similar to that in Example 3. Let us consider a *modify* update applied to an *A-Reader* r, thought as a *Reader*, that adds r a *Book* b. If b is not an *A-Book* as well, we are trying to specify an inconsistent update, because the resulting instance would have a dangling reference.

Update anomalies like the one outlined in the above example cannot be detected at compile-time. Other languages, such as Galileo [2], are able to prevent type-anomalies at compile-time, and therefore are strongly-typed. The trade-off to be paid for strongly-typing concerns limitations on the data model, e.g., the type of modifiable values cannot be redefined (by means of subtyping) throughout hierarchies. A solution for a more liberal data model, like the one presented in this paper, is to check legality of instances obtained as results of update operations, that is, we must resort to run-time type-checking.

Acknowledgements

The author thanks Paolo Atzeni, Giansalvatore Mecca, and Letizia Tanca for the fruitful discussions on the subject of this paper.

References

[1] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD International Conf. on Management of Data*, pages 159–173, 1989.

[2] A. Albano, L. Cardelli, and R. Orsini. Galileo: a strongly typed interactive conceptual language. *ACM Trans. on Database Syst.*, 10(2), June 1985.

[3] P. Atzeni, L. Cabibbo, and G. Mecca. ISALOG: A declarative language for complex objects with hierarchies. In *Ninth IEEE International Conference on Data Engineering, Vienna*, 1993.

[4] P. Atzeni and L. Tanca. The LOGIDATA+ model and language. In *Next Generation Information Systems Technology, Lecture Notes in Computer Science 504*. Springer-Verlag, 1991.

[5] J. Banerjee et al. Data model issues for object–oriented applications. *ACM Trans. on Off. Inf. Syst.*, 5(1):3–26, January 1987.

[6] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2):138–164, 1988.

[7] S. Khoshafian and G. Copeland. Object identity. In *ACM Symp. on Object Oriented Programming Systems, Languages and Applications*, 1986.

[8] J. Mylopoulos, P.A. Bernstein, and E. Wong. A language facility for designing database-intensive applications. *ACM Trans. on Database Syst.*, 5(2):185–207, June 1980.

[9] J. Richardson and P. Schwarz. Aspects: Extending objects to support multiple, independent roles. In *ACM SIGMOD International Conf. on Management of Data*, pages 298–307, 1991.

[10] E. Sciore. Object specialization. *ACM Trans. on Database Syst.*, 7(2):103–122, April 1989.

[11] J. Su. Dynamic constraints and object migration. In *Seventeenth International Conference on Very Large Data Bases, Barcelona*, pages 233–242, 1991.