

A Synthesis of Complex Objects and Object-Orientation

Marc H. Scholl and Hans-Jörg Schek

ETH Zürich, Institute of Information Systems
CH-8092 Zürich, Switzerland
e-mail: {scholl,schek}@inf.ethz.ch

Abstract

Complex Object models, semantic or knowledge representation models on the one side, and object-oriented models on the other side are currently considered candidates for future databases. Either of them have their particular strongpoints and weaknesses, such that up to now no single model could be identified to suit all needs. Database models provide limited structuring capabilities and too poor semantics and object-oriented approaches suffer from their navigational one-object-at-a-time style of operation. In this paper we show, how the approaches can be mixed into a single, coherent approach that eliminates all of these deficiencies and nevertheless preserves their advantages: flexibility through powerful structuring primitives, rich semantics, and encapsulation, and efficiency through optimizable descriptive, set-oriented query and update languages. We call this symbiosis of Complex Objects with the object-oriented paradigm “Complex-Object-Orientation”.

Keywords: Data Models, Complex Objects, Nested Relations, Object-Orientation, Query Language

1 Motivation

Reviewing recent database research, one can observe—among others—two main-stream developments: the first one is along the line from CODASYL network databases to Object-Oriented DBMS, the second one is the extension of the relational model to nested relations and further towards Complex Objects (aka ‘structural’ object-orientation). The need for a greater variety of structural abstraction mechanisms made database research also reconsider semantic models and knowledge representation schemes. Complex Objects are constructed by repeated application of tuple and set constructors. Nested Relations as a special case of Complex Objects have been studied in detail in their theoretical as well as practical issues during the past few years. Some prototype implementations of nested relational systems are already available. One of the major strong points of these “relational-style” models is their great potential for query optimization.

While Complex Objects cover (most of) the structuring aspects of new applications, the object-oriented approaches are oriented towards operational issues, often also called behavioural modeling [Dit86]. The idea here is not to operate on objects with general purpose operators (like the ones provided by a relational algebra, for instance), but rather to use type- (class-) specific operations. The advantages are twofold: first, the structure of objects is hidden as the user operates only by functions on them (encapsulation) and second, these functions can implement integrity checks that are specific to the object class. Thus, an object-oriented database system provides more modeling capabilities on the operational side, while on the other hand this typically leads to a navigational way of manipulation: a function is applied to an object (class), this function returns another (set of) object(s), again other functions can be applied and so on.

The notion of referencing an object from other objects (i.e. being able to identify an object) is a necessary prerequisite for *all* models providing network structures, i.e. sharing. In the CODASYL network data model [Oll78] the “database key” of a DBTG-record was the (implicit) identifier used as “currency indicators”, which however could be made explicit (stored in a programming language variable) by a special DB-operation (“ACCEPT”). Referencing to other objects is provided by CODASYL “Sets”. More recent models either have explicit object identifiers (surrogates) or use them implicitly and have operations that establish and follow references from one object to others.

Generalization as an abstraction mechanism originates from semantics data models or knowledge representation schemes. It can also be found in most object-oriented models. A generalization hierarchy gives rise to inheritance, i.e. functions/methods (attributes) of a general class are automatically available for all subclasses too (structural inheritance).¹ Besides the effect of a shorthand notation for type

¹Inheritance of values for ‘attributes’ of subobjects from superordinate objects (defaults) is a second way how generalization can be exploited. In this paper we restrict our concern to only the first one, viz. structural inheritance.

definitions it is useful to have a set of predefined functions/predicates available with generalization, e.g. to test whether an object of a superclass belongs to some of the subclasses.

Usually these two new directions of database evolution are considered competitive. Unifying the concepts of object-based models (identity, sharing, operating by type-specific functions, “pointer chasing”) and the ones from the relational world (being value-based, set-orientation, descriptive query languages) seems to be difficult, if not impossible, and even unwanted (“No Select-From-Where thinking any more!”²). This has been the starting point and challenge for our work. The problem we consider is the *integration* of the object-oriented paradigm with Complex Object databases, or the integration of data and object bases for short. We will show that it is rather easy to combine the essential features of both directions into a coherent framework. Set-oriented query languages *are* applicable in an object-based, and even object-oriented (encapsulated) setting. The essential idea is to view the object world as a set of *recursively nested relations*, or recursively defined Complex Objects, and to apply usual nested relational query languages.

Whereas (nested relational) retrieval operations on recursively nested relations have already been introduced in [SS88], the main focus of this paper is on update operations. They are the really crucial point as we have seen that the retrieval part is straight forward. For updates, however, it turns out that we have to carefully distinguish between two different environments: the encapsulated object world, i.e. the object base³, and the world of interpretable values. The latter one is necessary to communicate with the outside of the encapsulated world in both retrieval and update operations. One particularity about this approach is that while dynamic type construction (via projections) is very common in value-based models (i.e. relational ones), it is not in object-based models. Dynamically creating objects of new types as the result of (retrieval) operations is often impossible. This problem has specifically been addressed in [BKK88]. The claim was that the result of a query should match a predefined type, thus either all or only one “attribute” of objects may be projected. Our idea, however, is to dynamically extract Complex Objects (i.e. data structures) out of an object base when executing projections. Thus we avoid the problem of dynamically creating *object* types (classes).

The ORION query language described in [BKK88], the IRIS project [DKL88], the PROBE language [MD86], the EXTRA model of EXODUS [CDV88], the O_2 project [BBB⁺88, BLRV87], Larson’s proposal [Lar88], the MAD model [Mit87], and the HDBL language [PT86] have stimulated our approach. We found that none of the projects has offered a unified framework for objects *and* Complex Objects, which is our main interest. Neither did we see a sufficiently clear distinction between the notions of an *encapsulated object* and of a *data structure* used to communicate with a value-based environment. With respect to object-orientation we most closely follow the IRIS approach and the PROBE approach to structured objects [DMB⁺87]. Concerning Complex Objects we extended our work on nested relations [SS86, LS88, SS88] in a way that was influenced by the MAD model. Larson [Lar88] works on a similar extension of nested relations. The main contribution of our work consist in the utilization of recursively nested relations as the essential formalism for an object model. As these are an evolution from nested and these in turn from classical relations we smoothly integrate objects, complex objects and relations. With a minimum of effort the languages designed and studied for Complex Objects can be carried over to the object-oriented world, where others have also observed the need for an algebraic language [GM88]. In a more general setting, recursive type definitions have already been proposed in [Lam85] for the semantic representation of complex structures, but the style of operations is different from the one proposed here.

The paper is organized as follows: section 2 gives an introductory example of the way of operating with/on objects in our model. Subsequently, we present the underlying concepts in detail in section 3. We discuss some open issues and conclude with comparisons to related work in section 4.

2 From Nested Relations to Objects—Introductory Example

Generalizing the relational model by allowing repeated application of the “type constructor” *relation* (i.e. set-of-tuples) has enhanced the modeling power of relations, while preserving the essential feature of relational query languages—their descriptive, set-oriented nature. This very nature of relational query languages which makes them optimizable, which in turn is a necessary prerequisite for the database system to be efficient. The nested relational model is obtained from the flat relational one by allowing relations

²citation from the 2nd Workshop on Object-Oriented Databases, Bad Münster, 1988

³due to the distinction between an object-based and a value-based world we talk about an object base rather than a database

as values of attributes. Then we can apply relational operations (e.g. of an algebra) wherever we find relations. For instance, if tuples in a relation contain a relation-valued attribute, we can apply relational operations to it inside a projection or selection. As a result we have a nested relational query language derived from a flat one in a very similar way as nested relations were obtained from flat ones. Such nested relational algebra-, calculus- or SQL-style languages have been discussed in detail in several publications [JS82, FT83, SS83, AB84, SS86, RKB87, RKS88]. For the moment we concentrate on the structures and discuss operations later.

<pre> type Emptup = tuple eno: int, ename: chrstr, salary: real, dno: int end; Emprel = relation of Emptup key eno; Deptup = tuple dno: int, dname: chrstr, budget: real, end; Deprel = relation of Deptup key dno; var Empl: Emprrel, Dept: Deprel; </pre> <p style="text-align: center;">(a)</p>	<pre> type Deptup = tuple dno: int, dname: chrstr, budget: real, Empl: Emprrel end; Deprel = relation of Deptup key dno; Emptup = tuple eno: int, ename: chrstr, salary: real end; Emprrel = relation of Emptup key eno; var Dept: Deprel; </pre> <p style="text-align: center;">(b)</p>	<pre> type Emptup = tuple eno: int, ename: chrstr, salary: real, Belongs: Deptup end; Emprel = relation of Emptup Deptup = tuple dno: int, dname: chrstr, budget: real, Staff: Emprrel end; Deprel = relation of Deptup key dno; var Empl: Emprrel, Dept: Deprel; </pre> <p style="text-align: center;">(c)</p>
---	---	--

Figure 1: Relation as a type constructor: (a) flat, (b) nested, (c) recursively nested relations

Considering “relation” as a type constructor, we see an example of flat (a) and nested (b) relations in figure 1. As can be seen, the notions of types and variables are appropriate in the relational framework. Notice, that in order to obtain a nested relational structure the type definitions of relations must be non-recursive. Furthermore, if we allow a single relation type to occur within the definition of more than one superordinate relation the interpretation *must be* that the two share the *schema* of that subrelation, but the *values* of them are independant from each other (no subtuple sharing)! These restrictions guarantee a purely hierarchical model. While the model turned out to be efficiently implementable and is well-suited for the description of storage structures [BRS82, PSS⁺87, SPS87], obviously it is not general enough as a logical data model. This is due to the lack of modeling constructs for many-to-many or recursive relationships.

In order to describe such network structures using a nested relational approach we allow recursion in the relational type definitions (cf. figure 1(c)). For the moment we are not interested in details of such a recursive definition, but we observe that this “nested relational” kind of view represents a network. Notice, now we would assume that a subrelation occurring in two relations actually means sharing. What seems to be just a nice exercise when we restrict our interest on structures turns out to be the key to a powerful manipulation language when we look at operations later.

Let us now present a small, but rather complete example of an object base defined according to our model and how we operate on it. We use the notion of “type” and “variable” as known in programming languages. However we are not interested in related storage structure layouts which are derived from type definitions. We use only the semantic part of type definitions. As usually we assume a collection of given basic types like integer, character strings, boolean, . . . , and possibly also types like polygon. This means that a set of functions and operations is given such as equality, comparison, or arithmetic operations. In case of polygon—which may be an example for a user defined basic type—we would assume given functions like intersect which takes two polygons and returns true if the polygons intersect and false otherwise. In addition to such functions representing the (operational) semantics of types we require a convention for each basic type on how to represent objects for input and output operations (**in** and **out** functions, see below).

Given a set of basic types we allow the construction of new types by two type constructors: the (composite) object constructor and the set constructor. The object constructor roughly corresponds to a

tuple constructor or to the record type in Pascal. However, in our interpretation, an object constructor gives names to the functions (methods) which are applicable to a variable of this type (in this respect, object-oriented models are based on [Shi81]). We will use the following type definitions in our example “toy” object base throughout the paper.

```

type DEPARTMENTS = setof DEPARTMENT,
    DEPARTMENT = object
        (DNO:      int,
         DNAME:   chrstr,
         BUDGET:  int,
         MANAGER: EMPLOYEE inverse MANAGES,
         STAFF:   setof EMPLOYEE inverse BELONGS),

    EMPLOYEES = setof EMPLOYEE,
    EMPLOYEE = object
        (ENO:      int,
         SAL:      int,
         CHILDREN:setof PERSON,
         WORKSIN: setof PROJECT inverse MEMBERS,
         BELONGS: DEPARTMENT inverse STAFF,
         MANAGES: setof DEPARTMENT inverse MANAGER)
        inherits PERSON,

    PERSONS = setof PERSON,
    PERSON = object
        (NAME:   chrstr,
         BDATE:  date),

    STUDENTS = setof STUDENT,
    STUDENT = object
        (SNO:      int,
         FACULTY: chrstr)
        inherits PERSON,

    EMPSTUDENTS = setof EMPSTUDENT,
    EMPSTUDENT = object
        (PERIOD:  int)
        inherits (STUDENT, EMPLOYEE),

    PROJECTS = setof PROJECT,
    PROJECT = object
        (PNO:      int,
         MEMBERS: setof EMPLOYEE inverse WORKSIN);

```

Using these type definitions, the declaration of an “object base schema” is similar to the declaration of variables in a usual programming environment. Of course, classes of objects defined within the object base are assumed to be persistent.

```

define objectbase TOY =

    OBclass Depts:    DEPARTMENTS,
                  Empls:  EMPLOYEES,
                  Pers:   PERSONS,
                  Studs:  STUDENTS,
                  Empstuds: EMPSTUDENTS,
                  Projs:  PROJECTS;

```

Obviously, the type definitions are recursive. An employee is described in terms of e.g. a function returning a department object and department is defined in terms of e.g. employee. This has been typical for network databases as it is now for object bases. We may also interpret the definition as the schemes of several nested relations like

$$\begin{aligned}
 &Empls \ (eno, \dots, Projs(\dots)) \\
 &Projs \ (pno, \dots, Empls(\dots))
 \end{aligned}$$

which are obviously recursive. The difference to the explicit type definitions above are merely role names.

Furthermore we recognize the keyword “inverse” which simply states that if, for instance, an employee belongs to the staff of a project, we require that this project must appear in the worksin role of the employee. The “inherits” keyword finally states that e.g. student is a person and therefore all functions defined for the person type are also applicable to student. The example of employed students (EMPSTUDENTS) shows that inheritance may also be multiple. The union of the related sets of functions, i.e. of student and of employee in our example, is applicable then. Function names may have to be changed in order to get reasonable type inheritance.

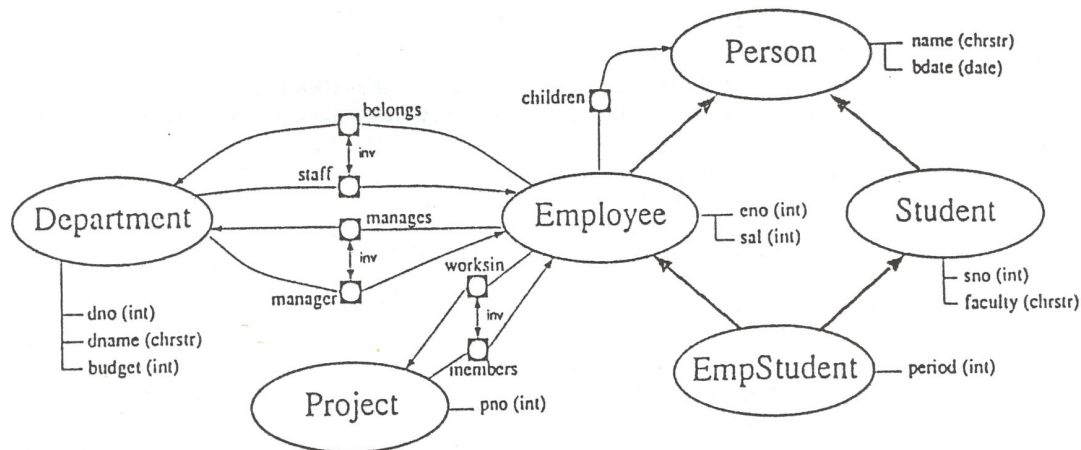


Figure 2: Example of a KL-ONE network

As a graphical representation of our example, we can consider KL-ONE knowledge representation networks (cf. figure 2). The choice of KL-ONE is somewhat arbitrary, however, the constructs provided by KL-ONE fit quite well with our model.

Objects are called “concepts” in KL-ONE (the ovals in the net) and they are related by “roles” (the arcs), e.g. “Employees” play the role “Staff” of a “Department”, the “Department” on the other hand plays the role “Belongs” of an “Employee” and so forth. Roles can be specified with a “number restriction”, e.g. an employee may have zero up to four departments to work for. We observe, that the two directions of a relationship are given separate (role) names, similar to the functions in our models. In this case the two roles are related by an “inverse” arc in the graph. KL-ONE provides a number of further constructs, especially some more restrictions on role inheritance and definitions of role differentiation [BS85], that we will not consider in this paper. Generalization (the **is-a** relationship) is denoted by the double arcs.

Let us now proceed with our example and assume that the above schema has been defined for the toy object base, but no objects have been ‘inserted’ yet, i.e. the object base is empty. We will use variables to temporarily refer to objects within sessions (programs operating on the object base).

Begin session 1.

```

objectbase TOY;

type .....

var d1,d2:    DEPARTMENT,
    e1,e2,e3: EMPLOYEE;

/* insert a new department with two employees */

d1 := insert into Depts
      (DNO := 25,
       DNAME := "NewDept",
       BUDGET := 100 000 );

```

```

e1 := insert into Empls
      (ENO := 1234,
       ENNAME := "John",
       BELONGS := d1);

e2 := insert into Empls
      (ENO := 4321,
       ENAME := "Mary",
       BELONGS := d1);

/* insert an employed student Max for a period of three months and insert another
   department, assign Mary and Max to it and let John manage this department */

e3 := insert into Empstuds
      (ENO := 9001,
       ENAME := "Max",
       PERIOD := 3)

d2 := insert into Depts
      (DNO := 125,
       DNAME := "NextDept",
       BUDGET := 10000,
       STAFF := (e2,e3),
       MANAGER := e1)

```

End session 1.

Note that by the insertions of **e1** and **e2**, in particular through the assignments to the **BELONGS** role the inverse roles (**STAFF**) are filled automatically. Further, the insertions of **e1**, **e2**, and **e3** are propagated into the superclasses (e.g. **PERSON**). Nevertheless there are still many values undefined. So let's have a next session, possibly after further insertions, but notice that the temporary names (variable bindings) are lost, of course.

Begin session 2.

```

objectbase TOY;

type ...

var dd: DEPARTMENTS,
     m: EMPLOYEE,
     ed: EMPLOYEEES;

/* delete the departments where the budget is less or
   equal 10000 and fire the persons who manage these */

dd := filter x in Depts with BUDGET(x) <= 10000;
ed := filter y in Empls with MANAGES(y) in dd;
delete e in ed;
delete d in dd;

/* assign employee Mary as a manager to the NewDept department(s) */

m := pick ( filter e in Empls with ENO(e) = 4321 );
dd := filter d in Depts with DNAME(d) = "NewDept";
update d in dd: MANAGER(d) := m;

```

End session 2.

We applied a filter on classes of the object base in order to define the set variables `ed` and `dd`, this corresponds to a selection in the relational algebra. It always returns a set of objects of the same type as the input set. The `delete` operation removes all objects in the set (`ed` and `dd` respectively) from the object base, the set variables have the value “empty set” afterwards. Notice, that we reuse one of these variables (`dd`) for the next operation. There we need to identify a single object from the object base (‘Mary’) in order to assign a value to the `MANAGER` function (which must be an `EMPLOYEE`, not a singleton set of type `EMPLOYEEES!`). However, the variable that identified this object in session 1 (`e2`) is no longer available. So we first apply a filter returning a singleton set⁴, and convert it into the object contained in it by the `pick` operation. This is somehow similar to the `unnest` operation in the nested relational model.

Up to now, we only discussed operations *inside* the object base, or to bring objects into it. All we did was insertions, updates, and deletions, i.e. applying functions on objects resulting in other objects. Now let us extract some information out of the object base. We will first exploit the fact that for a basic type we have an `out` function attached to it. As mentioned above, the idea how to extract *data* from the object base is to dynamically construct new (data) types as known from the relational projection operation. We first show this with a very simple ‘projection’, the next step then is, more generally, the dynamic construction of complex data structures using these primitives in a nested way.

Begin session 3.

```

objectbase TOY;

type ...

var dj: DEPARTMENTS;
    ....
output buffer E1,E2,E3;

/* let us extract the budget of all departments managed by Mary */

dj := filter d in Depts with NAME(MANAGER(d)) = "Mary";
E1 := extract BUDGET(x) from x in dj;

/* similarly we may extract the department name and budget of
   all departments having no student as employee */

E2 := extract DNAME(d), BUDGET(d) from d in
      ( filter d in Depts with
        0 = ( filter s in STAFF(d) with STUDENT(s) )
      ) ;

/* next, we extract a complex data structure which shows for each department
   its employees and for each employee children and projects she works in */

E3 := extract DNO(d), DNAME(d),
           extract ENO(e), ENAME(e), SAL(e),
           extract NAME(c)
           from c in CHILDREN(e),
           extract PNO(p)
           from p in WORKSIN(e)
           from e in STAFF(d)
           from d in DEPTS;

```

End session 3.

Notice, that we used output buffers to keep the result of an `extract` operation. The idea is, that these buffers are used to transfer sets of Complex Objects to the outside environment. The first extraction

⁴we assume that `ENO` serves as a user ‘key’

creates a very simple data structure: a set of numbers representing the budget of departments managed by Mary. Even if Mary manages only one department, we obtain a (singleton) set. The next extraction creates (like the first one) a usual flat relation, viz. a set of tuples containing name and budget values of departments. Here two things are of particular interest: first, we need a predicate testing the type of an employee (as **EMPLOYEE** is defined to have subtypes, i.e. other types have been defined inheriting structure from it). We assume a function “ $\langle Type_Name \rangle$ ” for every subtype. If applied to an object of a supertype, this function returns true, iff the object also belongs to the subtype. We used the **STUDENT** function, which is applicable to all **PERSONS**. Alternatively, we could have used the **EMPSTUDENT** function. Secondly, observe that we constructed a set of data tuples, consisting of **NAME** and **BUDGET** components. There is no predefined *object* type corresponding to this, rather we dynamically constructed this type as a *data* type. The tuples hold the values of the **out** functions applied to the result of **NAME** and **BUDGET** respectively. More precisely, the tuples produced are $\langle \mathbf{out}(\mathbf{DNAME}(d)), \mathbf{out}(\mathbf{BUDGET}(d)) \rangle$.

The result of the last extraction is a nested relation, i.e. a hierarchically organized data structure that has dynamically been constructed by repeated application of the extract operation. Extract serves as a constructor for data structures of the form set-of-tuple. Thus, we obtain interpretable values (i.e. data) from the encapsulated object base by using this operation. Notice, that only functions returning basic object types may appear in the extract list, because the others do not have a representation outside, i.e. no predefined **out** function.

3 Concepts of a Complex-Object-Oriented Model

In this section we will give the rationale and detailed explanations of the concepts that have been used in the example above.

3.1 Data Model vs. Object Model

The difference between the object-based and the value-based philosophy is that the latter defines *data structures* used to store data as given from the outside while the former takes an encapsulated point of view. Hence, we define classes of objects of corresponding types. What looks like the definition of a record type with component types is now interpreted as an *object type* definition, i.e. a set of functions (methods) are applicable to objects of this type. The result of such a function application is an object of the “component type”.

Thus, what has been called the schema of a relation is now the definition of an object class, what has been an attribute of a relation, is now a function (method) applicable to objects of the class. Interestingly, in the (nested) relational model tuples can be considered functions mapping attribute names to values of the underlying domains ($t \in \mathit{val}(R)$ is a function, $t(A)$ for $A \in \mathit{attr}(A)$ is the attribute value from $\mathit{dom}(A)$), while now we consider the methods as functions applied to the objects ($o \in O$ is an object of class O , then $A(o)$ is the (set of) object(s) returned by method A defined on class O). Notice, that we allow set-valued functions, this gives rise to Complex Objects or nested relations. If all methods are single-valued, we have an object class corresponding to a flat relation. If we restrict ourselves to the case where the only functions returning non-basic object types are set-valued, we end up with structures similar to nested relations.

Generalization is handled on the level of type (or class) definition too: if type *Sub* is a subtype of class *Sup* (“*Sub inherits Sup*”), then all functions applicable to elements of a class of type *Sup* can also be applied to elements of subclasses of type *Sub*. Furthermore, we assume with each definition of such an **is-a**-relationship the automatic definition of a function on the supertype to determine whether an object of a superclass belongs to the subclass. In our example, we used a function “**STUDENT**” defined on **PERSONS**.

The definition of **inverse** functions (or roles in the KL-ONE terminology) is kind of an integrity constraint: if the value of one of the functions is changed, the system automatically reflects this change in the inverse function. For instance, if an employee is added to the staff of a department, the **add** operation on the **STAFF** role ‘triggers’ the corresponding **add** of the department into the **BELONGS** role of the employee.

Before we switch to operations we have two important issues left for our discussion of the model: a network structured model is necessarily *object-based* as opposed to value-based models like the relational one, and secondly, we claimed to adopt the object-oriented paradigm of encapsulation.

The relational model is a value-based model: all information about the real world is represented as values inside the database. Attributes of tuples as well as relationships among tuples are values (of foreign keys in the latter case). This fact has been the key to the success of the relational model, because it made data manipulation languages simple and optimizable. On the other hand, a value-based model does per se not allow sharing: if a tuple relates to more than one tuple in another relation, then this can only be represented by introducing redundant storage of data (i.e. of the foreign key). There is no inherent referencing mechanism available in this model. Object-based models do provide such referencing mechanisms. Basically, all objects in an object base can be identified. Identifiers might be explicit or implicit. In our model we assign temporary names (variables) to objects. Such variables are used to refer to objects in other operations.

As the word encapsulation suggests, we have to distinguish two environments: an *inside* and an *outside*. This is of particular importance for update operations. Inside an object base we have objects that can be identified (referred to) and are manipulated by functions, but nothing inside the object base is a *data item* that can be given outside into a (value-based) programming language or displayed on a computer screen. Similarly, objects can not be brought into the object world from the outside in general. Of course, we assume that for basic object types, like numbers and strings, we have a standard representation in the value-based world, e.g. sequences of digits or quoted character strings. The principal idea is that there exist standard **in** and **out** functions to convert data of basic types into objects and vice versa. For non-basic, i.e. constructed types we do not a priori assume the existence such functions. Therefore, we might not be able to convert such objects into data structures as a whole, but rather have to construct data structures from (simpler) component object types. That is, how our retrieval operations work. It is interesting to notice that such conversion functions give rise to a different, but related, area of current database research: type extensibility [WHW88, WSSH88]. There they have observed the same need for **in** and **out** functions.

3.2 Operations on Objects

In the sequel we discuss the operations of our object manipulation language in an order that seems to be natural: first the operations for generating objects, then for the manipulation of objects and finally the retrieval operations.

3.2.1 Object Creation

In the relational context—or more generally in a value-based environment—we *store* tuples, i.e. collections of attribute values. This is just a matter of making certain data structures persistent, that is save them for further manipulation beyond the end of an application program (transaction). In an object-based model we *create representations* of objects (inside the object-base, by applying a system-provided function) and assign relationships to other objects. For instance, we create a new **EMPLOYEE** object and fill it into the **STAFF** role of an (existing or newly created) **DEPARTMENT** object: “**insert**” creates a new employee object, assigns some existing objects to functions defined on **EMPLOYEE**, and returns the new member of the object class as a result. Thus, an **insert** operation is used as the right-hand side of an assignment to an object variable. Functions that return basic object types can be given explicit values (i.e. a constant can be used in the **insert** statement to assign a value to them⁵), other functions can only be assigned values by using object (set) variables, or by applying subsequent update operations (see below).

The object variable defined by the **insert** statement can be used to refer to the new object throughout the rest of the program. Variables can be thought of as kind of a named pointer (currency indicator, reference) to the object. Object sharing comes for free with this concept: if we want to enroll the new employee in two distinct projects, we simply used the variable twice to assign values to the **MEMBER** function of the two projects. Our object-based paradigm guarantees that the employee exists only once within the object base.

To summarize the **insert** operation: the insert is applied to a class, i.e. a set of objects, the result is a new object of the member type of the class, and the class is extended to have the new object as an additional member. Values for the functions defined on the member type can be assigned using constants (for “component” types with defined **in** functions) or variables in a collection of assignments. We notice

⁵the meaning is that the corresponding **in** function is applied to the constant and the resulting basic object is assigned to the function

that `insert` is the operation which turns data from outside the object base into objects, i.e. an `insert` converts from the outside into the inside of an object base.

3.2.2 Object Filtering

By now we know how to bring objects into the object base and how to refer to them during the life time of a session: by object variables. However, once the insert session is finished the variables are no longer available to refer to objects. Thus, the next step is to identify a (set of) object(s) that should be affected by other operations (updates, deletions, or retrievals) in subsequent sessions. The typical way of operation in object-oriented programming systems is to *scan* an object class or to follow “references” to other objects (e.g. scan the set resulting from a function application on an object). In database terminology this means sequential, navigational access—pretty much the same way as in CODASYL network database languages—rather than descriptive specification of a *set* of objects for further manipulation (“pointer chasing”). However, it has been our claim to provide a *set-oriented, descriptive* language for our object-based model. Thus, what we need next is something similar to a relational selection: a mechanism to identify a set of objects (within a certain class) by a logical condition on their properties.

Notice that relational selection is a retrieval operation, while database languages like SQL use selection conditions in update operations too.⁶ Furthermore, the advantage of relational algebra is that the relational model is *closed* under the algebraic operations, that is the result of any algebraic operation again is a relation, which enables composition of complex algebraic expressions from simple operations. These two observations together with our distinction of a values’ and an object world lead to a slightly different interpretation of **object filters** in our model:

- The object world is closed under object filters, i.e. the result of a filter is a set of objects (rather than a set of values), i.e. dynamically creates a (temporarily defined) object class.
- Hence, retrieval operations as well as update operations can be applied to the result of a filter.

Thus filtering is not considered a retrieval operation in our model, since these are assumed to convert back out of the object world (see below).

Obviously, the most trivial filter is the one which selects all objects from a given class: `Q := filter o in O with true`, where `Q` is the variable and `O` is an existing object class. This filter is equivalent to the assignment `Q := O`, i.e. `Q` is assigned to be an object class with an identical type and set of objects as `O`. Notice, that here we also have object sharing: each object in `O` also belong to `Q` and vice versa⁷.

There is nothing special about filter formulae on objects, we can have logical connectives (\wedge , \vee , \neg) of predicates on role values and/or constants, exactly as in the relational setting for attribute values.

One thing that almost comes for free with the object-oriented paradigm of encapsulation is that we may have arbitrary (user defined) predicates in filters. Any function defined on an object class returning a boolean value can be used as a filter predicate. Equality (identity) is probably available as a predefined predicate for all object classes, but typically some other predefined predicates will be applicable. The type checking predicates for all subtypes defined on a type are an example of predefined filters.

As we permit set-valued functions defined on objects it is consequent to allow set comparisons as filters too. Furthermore, this gives rise to nested filters as is the nested relational or Complex Object models: a filter can be applied to the result of such a function before comparing with another set in a filter predicate. We had such an example in section 2, where we filtered departments without students as employees:

```
... ( filter d in Depts with
      O = ( filter s in STAFF(d) with STUDENT(s) ) ...
```

here, `STAFF` is a function on departments that returns a set of employees. “`O =`” is a set comparison “equals the empty set”, and a nested filter is applied to `STAFF(d)` before testing this condition. Such combinations are correct since a set-valued function on an object identifies a class, and a filter applied to a class returns a subset of the objects in that class.

Thus, we have a very powerful filtering facility on objects derived from nested relational selections. Results of such filters are always *sets of objects*, i.e. filters can be used to define values of set variables.

⁶This is due to the fact that algebras usually do not provide any update operations at all!

⁷this has some effect on the semantics of update operations applied to the result of a filter, see below

However, as we already saw in the example in section 2, we may also need to refer to *single objects* in order to assign them to single-valued functions, for instance. The set-oriented paradigm of closed (nested) relational query languages, however, does not permit operations to return single objects. So we introduced a new operation, `pick`, which can be applied to a singleton set and returns the element contained in it. Hence, in order to identify a single object from within the object base we have to apply a unique filter, i.e. one that matches only one object, and pick the object out of the resulting (singleton) set with `pick`. Thus we have to assume that users can identify objects via filters or via relationships to other objects, if they want to `pick` them individually.

3.2.3 Object Updates

We know how to insert objects, so the next step is how to modify them. Again our intention is to achieve the full power of descriptive update facilities known from value-based models. To change values that have been assigned to functions defined on an object when it was inserted, we use an `update` statement. It contains a list of assignments to functions, just like the `insert` operation. Updates can be applied to sets of objects (permanent classes, filters, set variables: `update x in X: ...`) or individual objects (object variables: `update x: ...`). For instance, to give a 10% raise to the staff of the ‘NewDept’ department(s):

```
objectbase TOY;
var nd: DEPARTMENTS;
.....
nd := filter d in Depts with DNAME(d) = "NewDept";
update e in ( filter e in Empls with BELONGS(e) in nd ):
    SAL = SAL(e) * 1.1;
```

Notice that updates are an entry point for nesting: we can modify an object by modifying subobjects, e.g. the set of objects identified by a set-valued function. Thus modification operations can appear nested inside an update statement. For instance, the following nested update is equivalent to the previous sequence of operations (except the fact that no variable `nd` is defined):

```
objectbase TOY;
.....
update d in ( filter d in Depts with DNAME(d) = "NewDept" ):
    update e in STAFF(d):
        SAL = SAL(e) * 1.1;
```

Obviously, updates can be nested more than one level deep.

3.2.4 Object Deletion

Finally, objects can be deleted. Deletion is a very simple operation once we know how to identify (filter) objects. Given a class `O` of objects we can delete all objects contained in that class by `delete o in O`.

Notice that `O` can either be an object class permanently defined in the object base (i.e. an element of the ‘schema’ of the object base), or a dynamically defined object class (e.g. a variable holding the result of a filter). In any case, the objects contained in `O` are deleted from all classes (permanent or dynamic). This includes all classes resulting from the application of set-valued functions to other objects. Particularly, we stress that a deletion applied to a filter also deletes the objects from the original class, not only from the (temporary) result of the filter! Therefore, the sequence “`Q := O; delete q in Q;`” is identical to “`delete o in O; Q := O`”: both `O` and `Q` are empty object classes after executing any of them.

We can also apply `delete` to object sets identified as the result of set-valued functions defined on other objects, i.e. apply `delete` nested within an update. In order to delete all employees named ‘Tim’ working in the ‘NewDept’ department(s), for instance, we use

```
objectbase TOY;
.....
update d in ( filter d in Depts with DNAME(d) = "NewDept" ):
    delete e in ( filter e in STAFF(d) with NAME(e) = "Tim" );
```

Notice the semantics: the objects are deleted from the object base, i.e. they are no longer existent in the `STAFF` role, nor in the `Empls` class or any other class afterwards.

3.2.5 Adding and Removing Objects

The `insert` and `delete` operations affect both: the objects and the classes they belong to. Thus, we need an additional pair of operations to manipulate the relationship between objects and classes without an effect on the existence of objects: `add` and `remove`. If we just want to drop the assignment of employees named ‘Tim’ to the ‘NewDept’ department(s) and keep them as objects we use `remove` instead of `delete` inside the `update` statement shown above:

```
objectbase TOY;
.....
update d in ( filter d in Depts with DNAME(d) = "NewDept" ):
    remove e in ( filter e in STAFF(d) with NAME(e) = "Tim" )
        from STAFF(d);
```

here we drop the employees from the staff, but keep them anywhere else in the object base, e.g. in the person class `Pers`.

Up to now, we applied `delete` and `remove` to sets X of objects using the syntax “... x in X ”. The operations can be used on single objects as well: if A is a variable identifying a single object, `delete A` or `remove A from ...` are valid operations, too. Obviously, the `add` statement is inverse to `remove`, i.e. `add x in X to A` or `add x to A` inserts existing objects into a class A .

Besides the manipulation of relationships among objects, we can use these operations to manipulate the type hierarchy. We stated above that we assume (boolean) functions on supertypes determining whether an object of the supertype also belongs to one or more of its subtypes. Consequently, we can use the `add` operation to make an object of a superclass belong to a subclass, and `remove` to drop it. For instance, we can make all employees of the “NewDept” department(s) students, i.e. `EMPSTUDENTS`, we use

```
objectbase TOY;
.....
ES := filter e in Empls with DNAME(BELONGS(e)) = "NewDept";
add e in ES to Empstuds;
```

The `add` statement puts all employees into the class `Empstuds`, i.e. any one of them now **is-an** `EMPSTUDENT` (i.e. a `STUDENT` and an `EMPLOYEE`, and a `PERSON`, of course). Conversely, if an employed student ‘Jerry’ does no longer work in the ‘NewDept’ department, but concentrates on his studies, i.e. we want to drop him from the department and make him a student only:

```
objectbase TOY;
.....
ES := filter e in Empls with NAME(e) = "Jerry"
    and DNAME(BELONGS(e)) = "NewDept";
remove e in ES from Empls;
```

as we removed Jerry from the `Empls` class, he no longer **is-an** `EMPLOYEE` and thus neither an `EMPSTUDENT`. As a consequence, he can no longer be in the staff of the department and gets removed from this set automatically.⁸

3.3 Extracting Data from the Object Base

In the (nested) relational context *projection* is the operation that dynamically constructs new (data) structures from the predefined relations. Our idea how to operate on an object base is the following: we use (nested) `extract` operations to dynamically construct complex data structures out of the (static) object network. So two aspects are important to remember: first, we want *dynamic* construction of new structures—this reflects a significant part of the power of relational query languages—, and second, we construct *data structures* from objects. That is, `extract` is the operation that brings us back out of the encapsulated object world.

Let us discuss the concept using an example first. The obvious way of getting back into the world of values is to apply the `out` functions to primitive objects, e.g. if `E1` is an object variable whose value is an employee object we can retrieve the employee number by `outnumbers(ENO(E1))`. Thus, we know how to retrieve single, primitive values from single objects.

⁸This is one particular choice of how to interpret such an operation. We gave the shortest, but also most tricky one intentionally. A simpler solution would be, to first `remove` Jerry from `STAFF(d)` explicitly.

In order to retrieve *structured* data we obviously have to introduce a retrieval operation that *constructs* certain data structures from simple values obtained from the object base as described above. There is, of course, a variety of choices for such constructors. We could, for instance, define a generic **out** function for *sets* of primitive objects as $\mathbf{out}_{\text{set_of_}X}(XS) := \{\mathbf{out}_X(x) | x \in XS\}$, and the like. In this paper we focus on retrieval functions that construct Complex Objects, or even more restrictive Nested Relations, as the output structure. Thus, we need set and tuple construction. Sets are already supported by the object model, tuples are constructed with the **extract** operation (basically a projection).

If we want to retrieve the set of all employee numbers⁹, we use an extraction applied to the class: $Q := \mathbf{extract} \text{ ENO from } e \text{ in } \mathbf{Empls}$. This looks identical to the relational query $Q := \pi[eno](Empls)$, and in fact the result is the same, just the way how this result is obtained is slightly different. We create a unary relation of employee numbers (i.e. values thereof) by collecting the results of applying the ENO function and the corresponding **out** function to all objects in the class **Empls**:

$$Q = \{\langle e \rangle | e = \mathbf{out}_{\text{numbers}}(\text{ENO}(E)) \wedge E \in \mathbf{Empls}\}.$$

That is, Q in fact is a relation (i.e. a data structure) with schema $Q(eno)$. In order to extract multi-attribute relations out of the object base, we simply list several functions in the **extract** list. Nested relations are constructed by nested **extract** statements as shown in session 3 in the example of section 2. There we also observe that extraction can be combined with filters.

One remark about extraction is important: as we construct data structures (relations) out of objects, we have to provide **out** functions for the ‘components’ mentioned in an **extract** list. While we assumed such functions available for primitive object types, for complex object types we either have to provide such functions, or we need to specify what (primitive) components of such objects shall be included in the data structure. For example, in

```
objectbase TOY;
.....
Q := extract DNO,DNAME,BUDGET,STAFF from d in Depts;
```

we know how to obtain data structures (numbers and strings) from the DNO, DNAME, and BUDGET components of a department, but how to make a set of employee objects (which is the value of the STAFF function) a data structure is not clear. Thus, the above projection is only legal if we have a predefined output function for sets of employee objects. Otherwise, we have to reformulate the operation, e.g.

```
objectbase TOY;
.....
Q := extract DNO,DNAME,BUDGET,
           extract ENO,ENAME from s in STAFF(d)
           from d in Depts;
```

i.e. we explicitly specify what data structure to create, viz. a (sub-) relation containing tuples with three components *eno*, *ename* and *salary*. Any of these is obtained from a primitive object type, hence we now have a legal extraction. Generally, unless we have a predefined output function for an object class (be it a default function or a user defined one) we have to specify how to construct data from the objects of that class. **Extract** is one of these default functions. It can be applied to all sets (classes) of objects and constructs a relation of tuples. The components of these tuples are constructed according to the specifications in the **extract** list.

We used the notion of *output buffers* that are supposed to keep the result of an **extract** operation. Obviously, we need some mechanism to transfer the (possibly large) result set of an extraction to the outside environment (application program or display system). In order to exploit set-orientation we transfer sets (cf. “portals” in [SR84] and “transfer areas” in [PSS⁺87]) rather than individual data items.

4 Conclusion

In this paper we presented a unifying approach towards Complex-Object-Oriented, i.e. the integration of Complex Objects and Object-orientation. In [GM88] they stated the need for a powerful algebra of object-oriented models, in order to make queries optimizable and thus object-oriented DBMSs efficient. We show that such languages already exist: algebras or other languages defined on Complex Objects

⁹what we actually retrieve is a set of 1-tuples containing the numbers

can be applied in models supporting object sharing too. Thus optimization techniques and other results obtained so far can be exploited. The key to this approach consists in viewing object networks as recursively nested relations and operating on them as such. For insertions and data extraction we distinguished two environments: the encapsulated object world and a value-based outside. Structured objects can be extracted from the object base into the value-based outside as complex data structures (Complex Objects) by applying nested extract operations (projections). Inside the object base we can apply powerful filters in a way similar to nested relational selections.

The set-oriented (‘relational’) operations we discussed can be viewed as default methods applicable to all classes of objects. Of course, to make use of the object-oriented paradigm, we allow type specific operations. These, however, are not different from any other object-oriented approach. The important point in this paper is that such general purpose set-oriented operations as known from classical database models can also be used.

We gave only an informal, example-driven overview of our model in this paper, a formal definition will be contained in a technical report that is in preparation. Concerning implementation we pursue two directions: first we are currently building a prototype on top of a relational system for experimentation purposes. Second, we will build a Complex-Object-Oriented frontend to the DASDBS nested relational DBMS kernel [PSS⁺87] in the COCOON project.¹⁰

Acknowledgement

The work presented in this paper has been discussed with and influenced by the following people: Catriel Beeri, Peter Brössler, Theo Härder, Bin Jiang, Raimond Lorie, and several others. Their help and constructive criticism is gratefully acknowledged.

References

- [AB84] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 191–200, Waterloo, 1984. ACM, New York.
- [BBB⁺88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gemerman, C. Lécluse, P. Pfeffer, P. Richard, and F. Velez. The design and implementation of *O₂*, an object-oriented database system. In K. R. Dittrich, editor, *Proc. Int. Workshop on Object-Oriented Database Systems*, pages 1–22, Bad Münster, September 1988.
- [BKK88] J. Banerjee, W. Kim, and K.-C. Kim. Queries in object-oriented databases. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 31–38, Los Angeles, CA, February 1988.
- [BLRV87] G. Barbedette, C. Lécluse, P. Richard, and F. Velez. Connecting the *o₂* data model to programming languages, version v0. Technical Report 13-87, Gip Altair, 1987.
- [BRS82] F. Bancilhon, P. Richard, and M. Scholl. On line processing of compacted relations. In *Proc. Int. Conf. on Very Large Databases*, pages 263–269, Mexico, 1982.
- [BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.
- [CDV88] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 413–423, Chicago, IL, May 1988. ACM, New York.
- [Dit86] K. R. Dittrich. Object-oriented database systems: The notions and the issues. In *Proc. Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.
- [DKL88] N. Derrett, W. Kent, and P. Lyngbaek. Some aspects of operations in an object-oriented database. *IEEE Database Engineering Bulletin*, pages 66–74, December 1988.
- [DMB⁺87] U. Dayal, F. Manola, A. Buchmann, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, and A. Rosenthal. Simplifying complex objects: The PROBE approach to modelling and querying them. In H.-J. Schek and G. Schlageter, editors, *Proc. GI Conf. on Database Systems for Office, Engineering and Scientific Applications*, pages 17–37, Darmstadt, April 1987. Springer IFB 136, Heidelberg.
- [FT83] P. C. Fischer and S. J. Thomas. Operators for non-first-normal-form relations. In *Proc. IEEE Computer Software and Applications Conf.*, pages 464–475, 1983.

¹⁰COCOON stands for “COupling Complex-Object-Orientation and Nested relations”, or—giving a hint on how it works—: a recursive acronym “COCOON = COcoon: Complex-Object-Orientation through Nested relations”

- [GM88] G. Graefe and D. Maier. Query optimization in object-oriented database systems. In K. R. Dittrich, editor, *Proc. Int. Workshop on Object-Oriented Database Systems*, pages 358–363, Bad Münster, September 1988. Springer LNCS 334, Heidelberg.
- [JS82] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non-first-normal-form relations. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 124–138, Los Angeles, March 1982. ACM, New York.
- [Lam85] W. Lamersdorf. *Semantische Repräsentation komplexer Objektstrukturen*. Springer IFB 100, Heidelberg, 1985.
- [Lar88] P.-Å. Larson. The data model and query language of laurel. Unpublished manuscript, University of Waterloo, Canada, June 1988.
- [LS88] R. Lorie and H.-J. Schek. On dynamically defined complex objects and SQL. In *Proc. 2nd Int. Workshop on Object-Oriented Database Systems*, Bad Münster, September 1988. (to appear).
- [MD86] F. A. Manola and U. Dayal. PDM: An object-oriented data model. In *Proc. Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, 1986.
- [Mit87] B. Mitschang. The Molecule-Atom data model. In H.-J. Schek, editor, *Proc. GI Conf. on Database Systems for Office, Engineering and Scientific Applications*, Darmstadt, April 1987. Springer IFB 136, Heidelberg. (in German).
- [Oll78] T. W. Olle. *The CODASYL Approach to Data Base Management*. J. Wiley & Sons, Chichester, 1978.
- [PSS⁺87] H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, and U. Deppisch. Architecture and implementation of the Darmstadt database kernel system. In *Proc. ACM SIGMOD Conf. on Management of Data*, San Francisco, 1987. ACM, New York.
- [PT86] P. Pistor and R. Traummüller. A data base language for sets, lists, and tables. *Information Systems*, 11(4):323–336, December 1986.
- [RKB87] M. A. Roth, H. F. Korth, and D. S. Batory. SQL/NF: A query language for \neg 1NF relational databases. *Information Systems*, 12(1):99–114, March 1987.
- [RKS88] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.
- [Shi81] D. W. Shipman. The functional data model and the language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.
- [SPS87] M. H. Scholl, H.-B. Paul, and H.-J. Schek. Supporting flat relations by a nested relational kernel. In *Proc. Int. Conf. on Very Large Databases*, pages 137–146, Brighton, September 1987. Morgan Kaufmann, Los Altos, CA.
- [SR84] M. R. Stonebraker and L. A. Rowe. Database portals: A new application program interface. In *Proc. Int. Conf. on Very Large Databases*, Singapore, 1984.
- [SS83] H.-J. Schek and M. H. Scholl. The NF² relational algebra for a uniform manipulation of external, conceptual, and internal data structures. In J.W. Schmidt, editor, *Sprachen für Datenbanken*, pages 113–133. IFB 72, Springer, Berlin, Heidelberg, 1983. (in German).
- [SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, June 1986.
- [SS88] H.-J. Schek and M. H. Scholl. The two roles of nested relations in the DASDBS project. In S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects*. Springer LNCS, Heidelberg, 1988. (to appear).
- [WHW88] W. Waterfeld, D. U. Horn, and A. Wolf. How to make spatial access methods extensible. In *Proc. Spatial Data Handling Conf.*, Sydney, August 1988.
- [WSSH88] P. F. Wilms, P. M. Schwartz, H.-J. Schek, and L. M. Haas. Incorporating data types in an extensible database architecture. In C. Beeri and U. Dayal, editors, *Int. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness*, Jerusalem, June 1988. Morgan Kaufmann, Los Altos, CA.