The NO^2 Data Model

Andreas Geppert
Klaus R. Dittrich
Vera Goebel
Stefan Scherrer

Technical Report 93.09
April 1993

Institut für Informatik
Universität Zürich
Winterthurerstr. 190
CH - 8057 Zürich
Switzerland

Email: {geppert, dittrich, goebel, scherrer}@ifi.unizh.ch

Abstract

This report describes NO[2] (***N***ew ***O***bject-***O***riented data model), the data model of CoOMS[1]. CoOMS is a structurally object-oriented database system currently under implementation at SNI[2]. It is intended to serve as both, as a self-contained full-fledged database management system and as the database component of the ITHACA[3] kernel.

In this report, we first describe the data modelling facilities of NO[2]. NO[2] distinguishes between objects and values, it supports complex objects (*is-part-of* relationships), and complex values can be created using a rich set of orthogonal constructors. Additionally, NO[2] supports type hierarchies, i.e., (multiple) inheritance and specialization of object types.

In addition to the data definition facilities, we also introduce a declarative data manipulation language (*Quod*) supporting queries and update operations. The style of the query language is similar to SQL. However, beside the typical operations for selection *Quod* also supports recursive queries.
Finally, an algebraic formalization for both, data structures and queries, is given.

1. ***Co***mbined ***O***bject ***M***anagement ***S***ystem
2. Siemens-Nixdorf Informationssysteme
3. ***I***ntegrated ***T***oolkit for ***H***ighly ***A***dvanced ***C***omputer ***A***pplications. ITHACA has been an ESPRIT-project (No. 2705). The work of A. Geppert, V. Goebel, and S. Scherrer for ITHACA has been supported by the "Kommission zur wissenschaftlichen Förderung" (KWF, Commission for the Support of Scientific Research).

Contents

1 Introduction

While requirements of standard application domains like banking are satisfied by traditional data models (e.g., the relational model [Date 83]) rather well, this is not the case for advanced applications [Bernstein 87, Dittrich 85, Lockemann 85]. Applications like computer aided design (CAD), computer aided software engineering (CASE), office automation systems (OIS), geographic information system (GIS) and so forth typically need richer capabilities for the modelling of data structures as just tables of flat values. They can also benefit from the possibility to define behaviour for data entities (i.e., methods). Therefore, object-oriented data models [Atkinson 89, Dittrich 90b] have been proposed for such so-called "non-standard" applications.

Another case for object-oriented database systems is the broad gap between the relational model and the type system of programming languages. While object-oriented programming languages offer powerful type systems, the relational type system is rather primitive. Hence, integrating traditional database systems with (object-oriented) programming languages leads to the so-called "impedance mismatch" (i.e., an awkward, often user-specified conversion of programming language objects into tuples). Hence, the promise of an object-oriented data model is the minimization of the impedance mismatch in that the respective type systems integrate better or are even equal.

In this report, we present NO^2, the data model of the database system CoOMS [Dumm 91] that has been developed for three distinct purposes:

- as the underlying database system for the software production environment developed in the ITHACA project,
- as the persistent object store for applications realized by means of this software production environment,
- as a stand-alone database system for advanced applications.

NO^2 is a structurally object-oriented data model [Dittrich 90b], since it supports complex objects (and a rich, orthogonal set of value constructors). Full object-orientation can be achieved through the integration with an object-oriented programming language, e.g., CooL[4] [Schröer 91] or C++.

The remainder of the report[5] is organized as follows: in Part 1, we describe the data definition language of NO^2. While Part 2 describes *Quod*, the query language we have developed for NO^2, Part 3 contains the NO^2 algebra. Finally, the grammar of the entire database language (also called *Quod*) is given in the appendix.

4. ***C***ombined ***O***bject-***O***riented ***L***anguage, an object-oriented language developed at SNI.
5. This report comprises the following previous ITHACA project reports: data definition language [Dittrich 90b], query language [Geppert 90a], algebra [Geppert 90b], and (revised) syntax of the entire language *Quod* [Geppert 93]

Part 1: The Data Definition Language of NO^2

At least for the traditional style of database systems, a data model consists of three components [Date 83]:

- a collection of basic types and constructors for further types,
- a collection of operators to create and deal with instances of those types,
- a collection of inherent integrity rules.

In NO^2, types are actually *object* types. Objects are a means to model the real world entities of interest, whatever their complexity and structure. Objects are distinct from *values*, which are used to describe properties of objects. Therefore, object types are the basic building blocks of the data model. The logical structure of any database that conforms to the model has to be built entirely from objects of those types. Operators provide a means of manipulating a database that is composed of valid instances of the object types. Integrity rules constrain the set of valid database states.
In a nutshell, NO^2 can be summarized as follows:

- it contains a collection of basic value sets,
- composite value sets are defined by constructors for tuples, sets, lists, and arrays,
- object types are defined on the basis of value sets,
- complex object structures can be defined either by means of including objects as components into other objects, or by referencing them,
- object types may be organized in a specialization/generalization hierarchy which allows for (multiple) inheritance of (structural) properties.

This part is organized as follows: in the next section, we describe our understanding of (structured) objects and values. Section 3 introduces the concepts of schemas and logical databases, while section 4 introduces a running example schema.

2 Objects

2.1 Objects and Values

Following [Khoshafian 86], there are at least two aspects of information stored in a data base (irrespective whether it is modelled as tuples, records, or objects) which have to be distinguished:

- the identity aspect and the
- data aspect.

Identity is a property of an object distinguishing it from all other objects. The data aspect of an object describes further properties, like values of attributes, etc. Although, e.g., in conventional relational systems tuples may be identified through user defined key attributes, both aspects should be strictly separated.

Like O_2 [Lecluse 89a], NO^2 distinguishes between the data and the identity aspect. The data aspect is described by values. Thus, saying that an object has a value means to talk about its data aspect. Objects are uniquely identified by so called object identifiers (or equivalently, surrogates, oid's). This identifier is defined by the system, cannot be altered by users and is not reused even if the corresponding object does not exist any more. An object is a pair (surrogate, value). The existence of identity for objects but not for values has some implications. Obviously, values cannot be identical, they can only be equal (in the usual sense). We have to distinguish between equality and identity when objects are concerned: two objects are equal if they have the same value (but potentially different surrogates), they are identical if they have the same identifier (but maybe —when looking at them at different times— have different values). Next, objects can be shared (in their capacity of acting as subobjects, see section 2.3), values not. Finally, objects exist in their own right, while values do not.

Since value serve to describe the data aspects of objects, they can only exist in contributing to the values of objects. Objects having the same attributes and structure are grouped together to object types. Object types are specified by a name and a appropriate value set.

2.2 Value Sets

While in conventional (e.g. relational) systems only tuples of atomic types are allowed, NO^2 provides for multiple and arbitrarily nested value constructors. Users thus may define value sets from existing ones. We assume some basic (atomic) value sets as given. Actually most of these value sets might be left out or others might be added without influencing the other data model decisions.

We assume basic value sets INTEGER, REAL, FLOAT, STRING, CHARACTER, BOOLEAN with the usual meaning. Furthermore, there is a basic value set LONG FIELD whose elements are unstructured byte sequences of unrestricted length. The value sets SUBRANGE and ENUMERATION are comparable to respective types in Pascal.

Complex value sets can be constructed from existing ones, which in turn may be basic or complex, using the following constructors:

- LIST,
- SET,
- TUPLE, and
- ARRAY.

Lists consist of a (variable) number of members, they are ordered and may contain duplicates. Sets have the common set theoretic meaning (i.e. they are no bags or multi-sets). Tuples consist of *attributes*, which in turn have *names* and *values*. Tuples are comparable to tuples in SQL or records in Pascal; in contrast to SQL, the domains of the attribute values need not be atomic. Arrays may be multi-dimensional and must be of fixed length (i.e., static arrays).

2.3 Complex Object Structures

Up to now, we are able to define complex value sets and to assign them to object types. However, this is neither very interesting nor particularly powerful, because so far there are no means to express relationships between distinct objects. This subsection describes facilities to define complex object structures.

First, there is the concept of *general references*. An object may reference other objects by including "*REF(oid)*" into its value. This kind of reference is called "general" in that the data model does not attribute any specific semantics to these references. Any kind of relationship between objects can be expressed by establishing general references. In DDL-definitions, general references are expressed by specifying "*REF(object type name)*" at any place where a value set is allowed.

Second, *subobject* (or synonymously, is-part-of) *relationships* can be modelled. Semantically, this relationship expresses that objects (beyond having arbitrary other values) consist of other objects and thus are called *structured objects* (or equivalently, complex objects, composite objects). Subobject relationships are expressed in schema definitions by just writing an object type's name wherever a value set name is allowed.

It is important to distinguish subobject relationships from other kinds of references. A structurally object-oriented database system has to provide facilities to manipulate objects of any desired structure "as a whole". The system has to distinguish the subobject relationships from general relationships, and to provide for operations manipulating structured objects. The benefit of differentiating between subobject and general references is that

- more real-world semantics can be expressed in object type definitions and

- operators provided by the data manipulation language treat these two kinds of references differently ("cascading" semantics for operations on structured objects).

We say that an object o_2 is a subobject of another object o_1 if

- o_1 has value v and
- v is represented by a sequence of list, set, tuple, or array constructors applied to o_2 and possibly some other values.

While general object references may be cyclic, this is not allowed for subobject references. Intuitively, it makes sense to allow cycles on the type level (objects may contain objects of the same type as subobjects), but not on the object level: this would imply that an object (transitively) contains itself as a subobject.

The transitive closure ob subobject references is defined as usual. *Shared objects* of o_1 and o_2 are objects which are in the transitive closure of the subobject relationship of o_1 as well as of o_2; o_1 and o_2 are said to *overlap*.
Depending on the real world situation of interest, shared objects may exist or not. If not, specific objects have to be assigned to parent objects exclusively. Furthermore, the existence of subobjects may be desired to depend on the existence of a parent object. If a parent object is removed, dependent subobjects have to be removed, too. Independent subobjects can continue to exist "stand alone" when the parent object is removed. In consequence, four kinds of subobjects are obtained by combination [Kim 89a]:

- sharable, independent objects,
- sharable, dependent objects,
- exclusive, independent objects, and
- exclusive, dependent objects.

The DDL allows to specify the desired case as an implicit consistency constraint. Figure 1 summarizes facilities to define value sets and complex object structures. Bold arrows represent construction of object types, the plain arrows represent value set constructors.

2.4 Type Hierarchies, Inheritance and Unions

NO^2 provides for the concept of *type hierarchies*. Type hierarchies are built from *specialization/generalization* relationships between object types. Every time an instance of an object type (say, *ot*) is required, an instance of any type being a specialization of *ot* is allowed as well. Thus, by means of specialization/generalization, the *is-a* relationship can be expressed (e.g., each *employee* is a *person*).

Specialization of object types is possible in two (not necessarily exclusive) ways:

- object types can inherit value sets from other object types and restrict (parts of) those value sets (*value restriction*), or

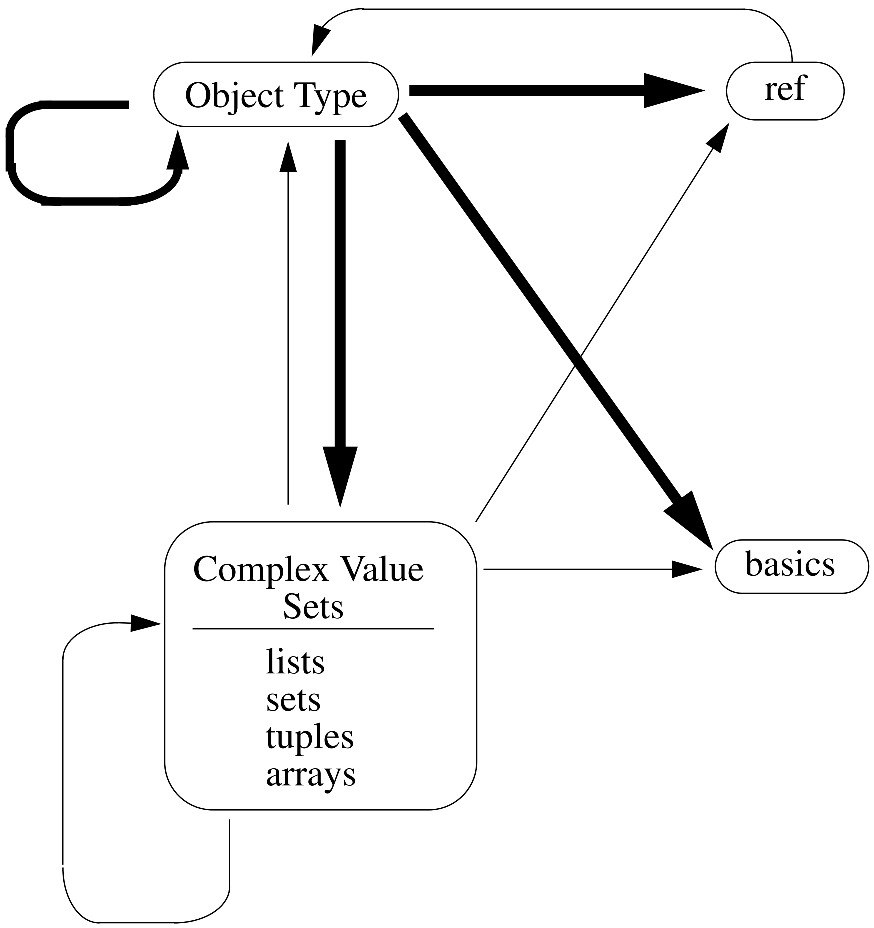

Figure 1 Value Set and Object Type Constructors

- object types can inherit from other object types and extend their value sets by further attributes (*structure extension*).

According to [Atkinson 89], the first case could be called *constraint inheritance*, the latter could be called *specialization inheritance*.

Restriction of value sets refers to the restriction of the inherited object type's value set to some subset of it. Value restriction may be applied recursively: e.g. integer value sets may be restricted to subranges of integers, subranges themselves may be restricted to smaller subranges, and enumeration sets may be restricted to subsets of the enumeration set. Complex value sets (lists, etc.) may be restricted by restricting their component value set.
Structure extension refers to adding attributes to tuple value sets. The extended value set then consists of tuples containing the inherited attributes plus the additionally defined ones. New attributes defined in this way must have new names, i.e., overriding is not al-

lowed. Structural extension is possible on any level of value set definitions. If a tuple to be extended is itself component of a tuple, a path to this tuple must be specified for uniqueness reasons. In order to specify this path it suffices to specify a sequence of tuple components (attribute names), since in the case of lists, arrays or sets no ambiguity may occur.
Object types may inherit from other types transitively. All the types an object type *ot* inherits from are called *supertypes* of *ot*, and *ot* is called a *subtype* of the types from which it inherits. As a consistency constraint, the transitive closure of the is-a relationship may not contain cycles. Intuitively, this does not make sense anyway neither in case of structure extension nor value restriction.

NO^2 supports *multiple* inheritance, i.e. object types may inherit from more than one object type. As soon as multiple inheritance is allowed, name conflicts may arise if different supertypes have identically named attributes. This problem is also present in fully object-oriented systems, and different approaches are possible to resolve these conflicts. We choose an approach similar to that of ORION [Banerjee 87] (and also EXTRA [Carey 88]). Conflicts have to be resolved in one of two different ways:

- if only one of the conflicting attributes is required, the user may specify from which object type the attribute should be inherited,
- if more than one of the conflicting attributes should be inherited, it is possible to *rename* the attributes. The user then specifies new attribute names and the supertypes from which the respective attribute should be inherited.

In contrast to the ORION approach, there is no default solution provided by NO^2.

A further problem arising from inheritance is how to assign types to objects. If an employee is also a person, is an instance of type employee also of type person? May objects have multiple types? Although we do not deal with data manipulation facilities here, it should be stated that it is desirable to include subtypes (automatically) in queries against supertypes (this implies that objects may have multiple types). Nevertheless, if an object has multiple types, the *most specific type* is always uniquely defined: if an object o has types T_i, there is exactly one that is a subtype of all others but not a supertype of any other type in $\{T_i\}$. This most specific type exists uniquely if and only if there are no cycles in the is-a relationship. Thus, when saying that objects have exactly one type, this refers to the most specific type in case of inheritance.

Union types are special cases of is-a relationships. Union types do not have instances of their own, they are just regarded as (disjoint) unions of some other (sub-) types. Furthermore, union types do not have value sets associated with them. Consequently, they cannot inherit from other types: because they do not have own instances, neither structure extension nor value restriction is applicable. Although union types do not possess own extensions, they are useful whenever instances of different types are allowed in a specific context. Whenever a union type is specified as type of the objects to be manipulated by DML-statements, instances of any subtype of the union are allowed as parameters.
Assume, for example, a university context where professors and assistants exist. In some cases, professors or assistants may occur (e.g. as heads of research groups or as teachers).

In other contexts, either professors or assistants may be required (e.g. the director of a department must be a professor). This situation could be easily captured in defining two object types (*professor* and *assistant*) and a union type *researcher*.

2.5 Uniqueness and Required Properties

Uniqueness properties are a further means to specify implicit consistency constraints. If an attribute or a combination of attributes is specified as unique for a given type, two different objects of that type must have different values associated with those attributes. As an example, assume an object type *department* whose instances represent the departments of a university. Attributes of the value of a *department* are *name* and *address*. An implicit consistency constraint may be that two departments always have different names. This fact could easily be expressed by defining the *name* attribute as *unique.*

If attributes are defined as unique, the system will not accept update or insert operations violating the uniqueness condition. In order to make necessary checks efficient, we restrict the set of attributes which may be specified as unique to those with *basic* domains. E.g., attributes having *integer* or *string* as domain may be declared as unique, while those having a set valued domain may not. Unique attributes may be components of (nested) tuples, but are not allowed to occur in sets, lists, or arrays. Combinations of attributes can be declared as unique, and multiple (unique) attribute combinations can be specified for an object type.

Furthermore, the relationship between inheritance and uniqueness attributes is clarified by the following rules:

- subtypes inherit uniqueness properties from their supertypes,
- subtypes are not allowed to specify inherited attributes as unique.

As an example, assume an object type *person* and a subtype *researcher* of *person*. If (say) the attribute *name* of *person* was defined as unique, *name* is also unique in *researcher.* This rule is consistent to the *is_a* semantics of inheritance: since each *researcher is_a person*, the attribute *name* has to be unique among *researchers* as well as among *persons.* Furthermore, the subtype *researcher* may specify additional uniqueness conditions for further attributes. According to the second rule, the set of possible unique attributes is restricted to those attributes defined for the subtype, but not for the supertype. As an example, assume that *name* was not specified as unique for *person*. Then it is not allowed to define *name* as unique for *researcher.* Otherwise, a set of objects might be consistent when regarded as *persons*, but inconsistent when regarded as a set of *researchers*.

Furthermore, especially when objects are created interactively it is in principle allowed to specify only some of the values required by the respective object type definition. For example, if one does not know the address of a person to be inserted in a database it is possible to omit the respective attribute value. Unspecified values are known from the relational model as *null values* and in principle should be avoided [Elmasri 89], since they imply problems for the semantics and processing of queries. Null values can be prohibited for all instances of a given type in that the value set definition is preceeded by the keyword *"required"*. For instance, if the attribute *name* of *person* is specified as required, each *person* has to have a *name* attached to it. Similar to uniqueness properties, required properties are inherited to subtypes.

3 Schemas and Databases

Different applications of a particular database system often need to work with (possibly disjoint) real world parts of interest. In this case, they should be concerned with their specific mini-world only, but not with those of other users. On the other hand, although there are several real world parts of interest in the context of one database system, there is a need for a global documentation of the whole real world covered by the particular database system. As a consequence, NO^2 - like other data models - provides for the concept of *global schemas* and *logical schemas*. In contrast to a schema which is used to specify the global view of the data managed by the database system, subschemas are used to realize user (or application) specific schemas.

The entire data representing the current state of the universe of discourse is said to be stored in the *global database*, whose structure is described by the *global schema*. In terms of NO^2, the global schema is a collection of type and value set definitions. The global database is further subdivided into *logical databases* (LDB's). Those are termed "logical" since they simply serve to collect objects (e.g. libraries, or data of specific users), but do not imply any physical characteristics. The structure of a logical database is specified by a *logical database schema* (LDB-schema)[6]. An LDB-schema essentially is a subset of the global schema, i.e. in order to specify an LDB-schema it is sufficient to enumerate the required object types. Multiple LDB-schemas need not necessarily be disjoint; if the same object type name appears in more than one LDB-schema, all those types are assumed to be identical. LDB-schemas may not be arbitrary subsets of the global schema, but have to be *closed* in the following sense: if an object type *ot* is part of an LDB-schema *s*, then

- all object types being more general than *ot* have to be contained in *s* as well,
- all object types *ot* references have to be mentioned in *s*,
- all subobject types of *ot* have to appear in *s*, too.

Obviously, these rules apply transitively.

As an example, assume a literature database of scientific publications. A group of researchers may share some information (say, surveys or text books). Beside the common information, each researcher may want to store information on literature of his special interests. Thus, the structure of objects representing documents and authors is specified in an LDB-schema (which, as usual, is a subset of the global schema). Furthermore, there can

6. Please note: the term "logical" may not be confused with the notion of "logical schema" in the sense of database design, where a logical schema results from the mapping of a conceptual schema to a concrete data model.

be one logical database for the group literature and one logical database for each of the researchers.

Given a collection of logical databases and attached schemas, different users or applications may view the objects of a (logical) database from different angles. This is supported by NO^2 by the notion of *derived types* [Geppert 92b]. A derived type is comparable to a relational view; it is virtual and can be used wherever an NO^2 type is expected. A derived type is virtual in that it has no own extension, rather its extension is derived and dependent from one or more other types (i.e., their extensions in the same logical database). A derived type is obtained in that specific operations like projections or selections are applied to existing types (which in turn may be derived). Furthermore, like in the relational model it is an open problem which update operations applied to an instance of a derived type are legal. Derived as well as normal types are grouped into *subschemas*, which specify the application or user view of a logical database. Subschemas always refer to (exactly) one LDB-schema, and —like LDB-schemas— have to be closed.

Continuing the above example, assume different researchers accessing the various literature databases. Each of them may desire to have a special view on her/his database. For instance, one may want to exclude some detailed information about authors, while others desire to view only those authors having published more than 10 articles. Thus, each one defines her/his subschema containing the respective derived types or reusing object types from the logical schema, respectively.

Figure 2 shows the relationships between schemas and databases. Recall that logical databases are not physical, and that the union of the logical databases equals the global database. This section concludes with the following table showing the correspondences of the various notions of "schema" among the various models, namely ANSI/SPARC [Lockemann 85], the relational model, and the network model [Codasyl 78].

NO^2	ANSI/SPARC	Relational Model	Codasyl
Global Schema	Conceptual Schema	Schema	Schema
Logical Schema	-	-	-
Derived Type	-	View	implicit
Subschema	External Schema	-	Subschema

Table 1: The Notion of Schemas in Various Models

Although we do not deal with these issues here, it should be noted that (LDB-)schemas and logical databases may be useful in combination with various other facilities like user authorization (the access of users can be restricted to specific logical databases), distribution, and design transactions (e.g. checkout/checkin mechanisms between logical databases).

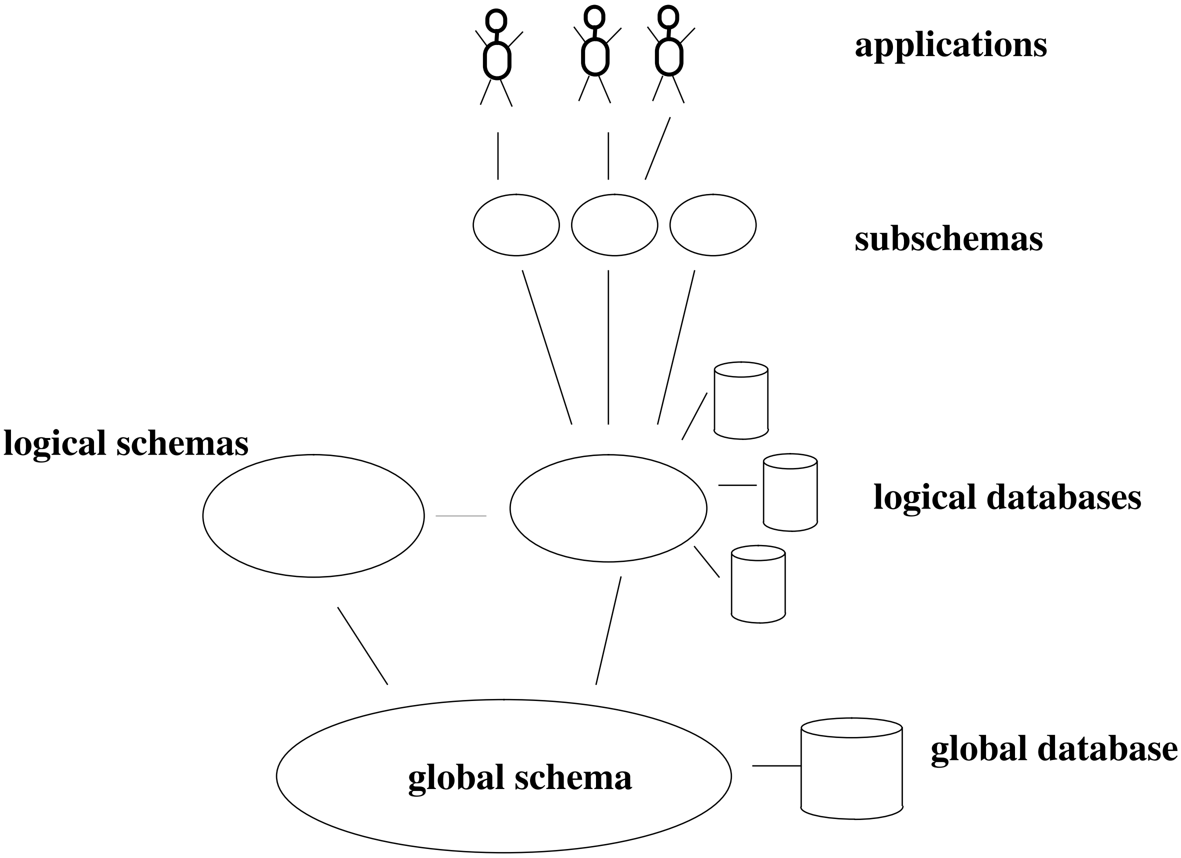

Figure 2 Schemas and Databases

4 An Example

In order to demonstrate the modelling facilities of NO^2, we model a part of a university. In our example, there are departments and persons. Departments may consist of several research groups. Each of these research groups can be split up into members (people) or further groups, which address more specific fields of the respective research area (and so on transitively). People working in a research group are called researchers, they are special persons. Researchers can be subdivided into professors and assistants. Figure 3 shows the sample schema. On this picture, rectangles represent object types, plain arrows represent subobject relationships, dashed arrows denote general references and bold dashed arrows represent generalization.

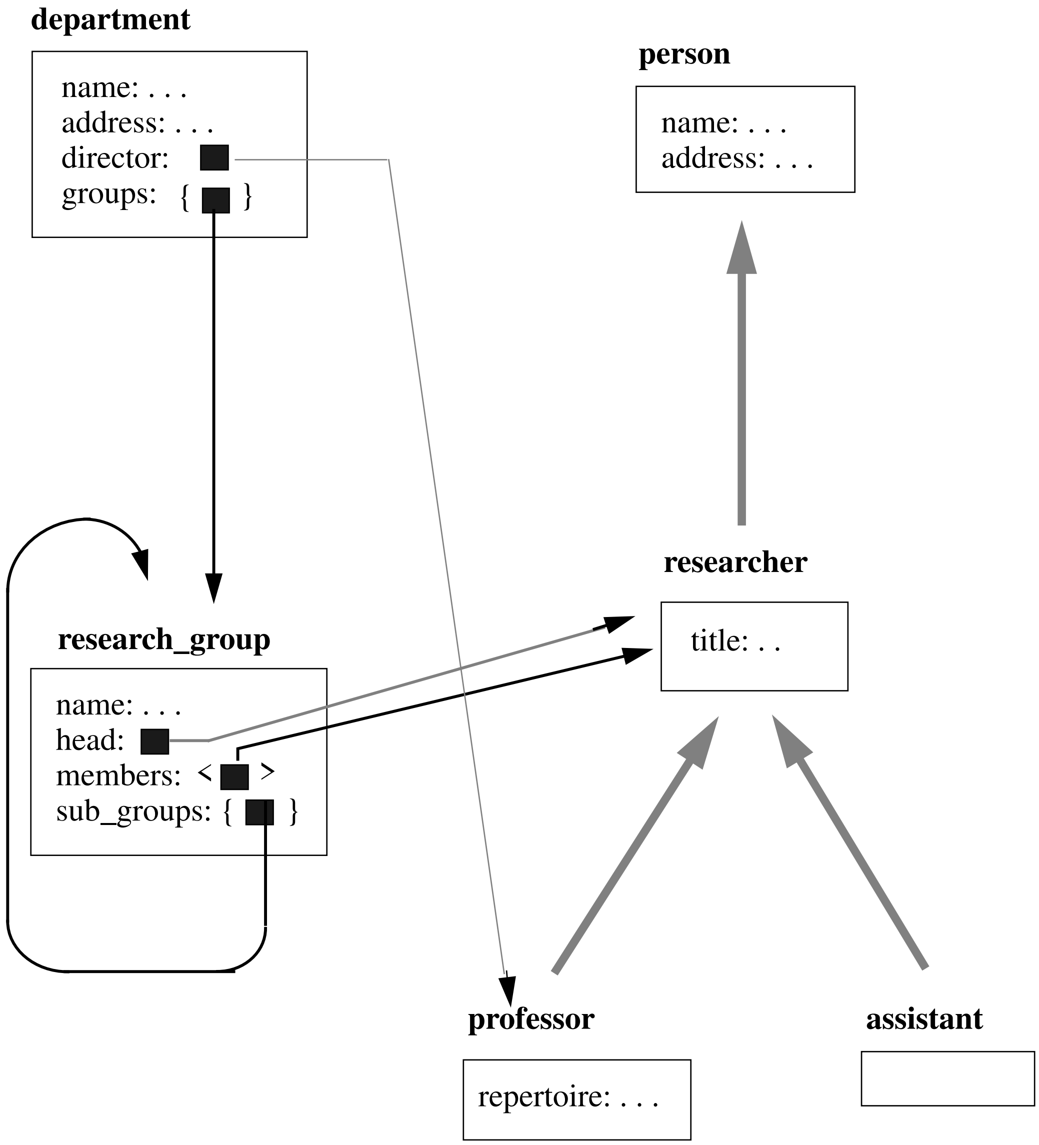

Figure 3: Sample Schema

4.1 The Schema

DEFINE LOGICAL SCHEMA peanuts_uni

DEFINE VALUE SET adress_vs =
TUPLE (street : STRING,
house# : INTEGER,
postcode: SUBRANGE [1000 .. 9999],
town: STRING).

```
DEFINE OBJECT TYPE person =
                    TUPLE ( name: REQUIRED STRING,
                     adress: adress_vs).

DEFINE VALUE SET title_vs = ENUM ( Prof, Dr, Dipl, PhD, Lic).

DEFINE OBJECT TYPE researcher
                    SUPERTYPE IS person
                    EXTENDS BY title: title_vs.
```

The object type *researcher* shows an example for structure extension. Instances of this type do not only have attributes denoting their title (as defined above), they also possess attributes defined for persons.

```
DEFINE OBJECT TYPE assistant
                    SUPERTYPE IS researcher.

DEFINE OBJECT TYPE professor
                    SUPERTYPE IS researcher
                    EXTENDS BY repertoire: SET ( STRING ).
```

Assistants and *professors* are further examples of inheritance. Both are subtypes of the object type *researcher*. Using these definitions, it is possible to subdivide researchers into assistants and professors (which are assumed to be two disjoint subsets). Thus, it makes sense to define types having no own attributes but inheriting attributes from other types. The semantics of the type assistant in our example is “all persons who are researchers, but not professors”.

```
DEFINE OBJECT TYPE research_group =
                TUPLE ( name: STRING,
                      head: REF( researcher ),
                      sub_groups:
                          SET (SHARED DEPENDENT research_group),
                      members:
                          LIST (SHARED DEPENDENT assistant) ).
```

The first thing to observe from the definition of *research_group* is the difference between general and subobject references. We modelled *sub_groups* (and also *members*) as subobjects because research groups are more closely related to their parent groups as could be expressed by general references. This becomes obvious when regarding possible DML-operations:

- If we had defined *sub_groups* as *SET (REF (research_group))*, research groups would exist independently of their parent group. However, we assume that cascading deletion semantics for departments, research groups, and members are more appropriate.

- General references are treated differently from subobject references in case of querying the database. E.g., it is be possible to read a whole structured object. The result then contains an entire object in the *sub_groups* attribute, while in the case of general references, only an object identifier would be returned.

Next, *research_group* shows the possibility to define recursive types, i.e. types whose instances may contain instances of the same type as subobjects (attribute *sub_groups*). Finally, one possibility to constrain subobject relationships is shown. Researchers may be members of multiple research groups, thus they are declared as sharable subobjects. On the other hand, assistants depend on the research groups they join. If they are no longer member in a group (e.g. because the last group they joined was removed), they will also be deleted (how hard is the life of an assistant!).

```
DEFINE VALUE SET department_vs =
                         TUPLE ( name: STRING,
                                 address: address_vs,
                                 head: REF ( professor ),
                                 groups:
                                   SET (EXCLUSIVE DEPENDENT research_group)).

DEFINE OBJECT TYPE department = department_vs.

END SCHEMA peanuts_uni.
```

The definition of *department* shows another possibility to restrict subobject relationships. In our example, research groups depend on the existence of the department they are associated with (thus, they are also declared as dependent). Furthermore, one research group can exist at only one place, it is therefore declared as exclusive.

Another point to observe is that it is just a matter of specification convenience whether an object type's value set is defined apart from the type (as in the *department* definition), or if it is defined with the object type. Furthermore, *department_vs* gives an example of the orthogonality of value set constructors: *department_vs* is a value set of tuples while the attribute *groups* is set-valued (and the members of those sets could again be tuples, lists, sets, or arrays).

An alternative model using a union type would be the following: one could argue that researchers cannot exist on their own. Either they are professors, or they are assistants and then have to work in a research group. But in the model given above, researchers that are neither professors nor assistants might exist. Furthermore, the title of professors is implicit, while assistants can have different titles. Thus, alternatively *title* might be defined as an attribute of *assistant* and *researcher* as a union type with subtypes *assistant* and *professor*:

```
DEFINE OBJECT TYPE assistant
                              SUPERTYPE IS person.
                              EXTENDS BY title: title_vs.
```

```
DEFINE OBJECT TYPE professor
                    SUPERTYPE IS person.
                    EXTENDS BY repertoire: SET ( STRING ).

DEFINE UNION TYPE researcher =
                    UNION (assistant, professor).
```

Note that *assistant* and *professor* cannot inherit (transitively) from *person* via *researcher* in this case; they have to inherit from *person* directly.

4.2 The Database

Finally, after having created the schema *peanuts_uni*, it may be assigned to a logical database called *peanuts_uni_db*.

```
DEFINE LOGICAL DATABASE peanuts_uni_db
                    peanuts_uni
END peanuts_uni_db.
```

Part 2: *Quod*, the NO^2 Query Language

Object-oriented database systems were initially intended to be mainly used by tools (i.e., application programs) rather than by human users. In consequence, procedural programming interfaces (in a one-record-at-a-time style) seemed to be the appropriate means of access. However, since programming interfaces are uncomfortable for interactive use by humans, and sometimes declarative access to databases is desired even in application programs, object-oriented database systems have to support query languages as well as programming interfaces. Therefore, support of query languages is one of the properties of an object-oriented database system as required by the definition of [Atkinson 89]).

This part describes a query language for NO^2. The overall requirement is to support associative and declarative access to a CoOMS database. As a stringent consequence, the query language has to be adequate with respect to the NO^2 data definition language, i.e., data modelling facilities have to be reflected by corresponding constructs for querying and manipulating the defined data. Especially,

- the query language has to support access to objects, their values, and structures among objects,
- it has to cope with the value set system of NO^2, which has been constructed to be completely orthogonal,
- the retrieval or manipulation of *is-part-of* hierarchies reflecting levels of abstractions has to be supported.

Quod treats objects similarly to tuples in other approaches. To a large extent, operators dealing with values are borrowed from HDBL[7] [Pistor 85, Pistor 86], where list, tuple and set constructors are completely orthogonal like in NO^2. In order to satisfy the third requirement, *Quod* provides for *recursive queries*, similarly to those described for some other structurally object-oriented database systems [Schiefer 89, Schoening 89].

7. **H**eidelberg **D**ata**b**ase **L**anguage

Last but not least, *Quod* is purposely close (as far as possible and meaningful) to SQL [Date 83], but obviously has to be beyond it. Although query languages are sometimes regarded as a matter of taste and SQL is said to be “ugly and clumsy” [Bancilhon 89], it is more or less accepted as a standard. Since many prospective users already know (and to some extent like) SQL, the adaption overhead when switching to NO^2 from a relational system can be kept smaller than otherwise.

This part is organized as follows. The next section describes query facilities of *Quod*, i.e., their overall structure, operators for objects and values, and built-in operators. Recursive queries are introduced in section 6. Update facilities are described in section 7. Finally, the last section motivates and introduces the usage of *Quod* as a full-fledged database languages, which besides DML operations also provides for data definition operations, access control statements, and so forth.

Throughout the paper we will use the running example representing a university mini-world. We refer to the sample schema in Figure 3 and a sample database in Figure 4. In the latter, subobject relationships are expressed by the nesting of boxes (which in turn represent objects).

5 Queries

5.1 Structure of Queries

This section describes the overall structure of queries.

Like in SQL and many other query languages intended for object-oriented data models [e.g., Bancilhon 90, Beech 88], NO^2 queries are structured in *select-from-where* blocks, where the *where* clause is optional.

A *select* clause specifies which parts of the objects or values identified by the *from* and *where* clause are to be extracted. In SQL, the result of a query is itself always a relation. However, in object-oriented query languages there are multiple possibilities for the result of a query. Some languages (e.g. [Bancilhon 90]) allow for the retrieval of existing objects or their values. Other languages [e.g. Kim 89a, Shaw 90] construct new objects as result of a query. Since the underlying data models provide for class lattices, too, problems arise where to place the result in those class lattices. We therefore follow the approach of [Bancilhon 90]: the result of a query may be existing objects or (parts of) values of objects. If we are interested in creating new objects in the context of a query, this has to be specified explicitly by nesting a *select* statement in an *insert* statement.

Since the result of an SQL query is always a relation again, the structure of results is obvious as soon as tables and desired attributes are specified. Since NO^2 provides for a richer system of value sets, the user is allowed not only to extract, but also to *restructure* or *recombine* objects and values. Thus, it must be possible to define the structure of results. This is also done in the *select* clause. In this case a *select* clause describes the structure (or

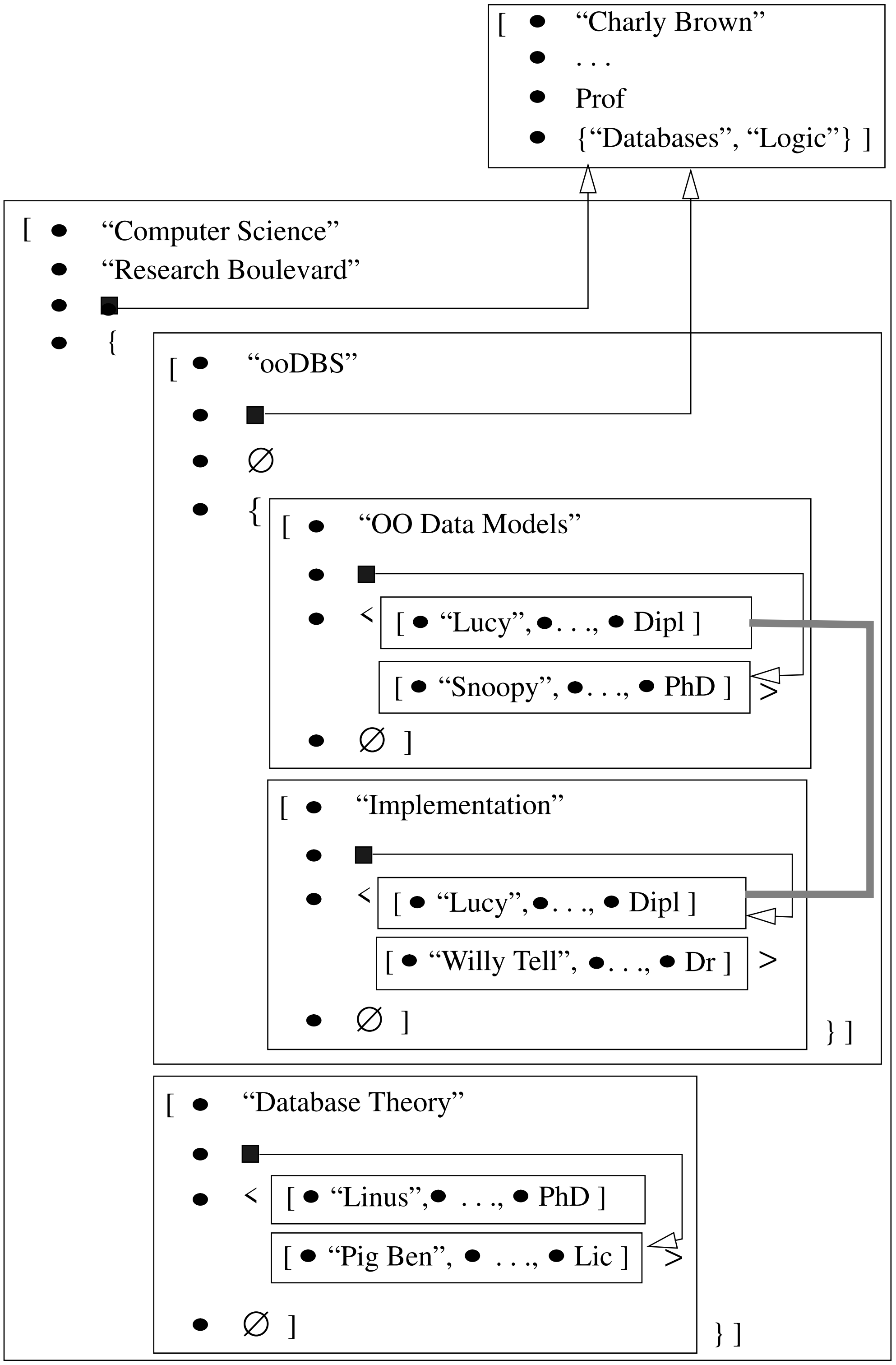

Figure 4: Sample Database

the type) of a result as well as how to extract the corresponding values from the data specified by the *from* clause. *Select* clauses in turn may contain queries, e.g., when the result of a query is desired to be a tuple consisting of two components, whose values are again results of a query, respectively.

In SQL, *from* clauses are simply used to name the tables to be examined. Additionally, variables can be declared, e.g. to distinguish between two different roles of a table. *Quod from* clauses specify extensions of object types. Furthermore, paths to (partial) values can be defined and assigned to variables. Thus, if the same path occurs more than once, variables serve for abbreviation. *Select* and *where* clauses as well as other variables may refer to variables. Of course, one restriction on variables is the absence of cycles in variable definitions.

The use of variable definitions also determines the type of the result of a query. Usually, queries map sets (extensions) into other sets. For example,

select d.groups
from d *in* department

returns a set of sets. This query extracts the set-valued *groups* of all departments (which in turn build a set), thus resulting in a powerset value. In contrast,

select g
from g in department.groups

results in a set of *groups* instead of a powerset.

Finally, *where* clauses serve to specify restrictions which have to be fulfilled by the resulting objects and values specified by *select* and *where* clauses.
Example:

select [name: department.name
director: department.director.name]
from department
where department.address =
"Research Boulevard".

This query returns a set of tuples. For each *department* located at "Research Boulevard", there is one tuple with the name of the *department* and the name of its *director.*

5.2 Accessing Objects

This chapter defines the constructs provided for accessing objects.

Objects (implicitly) consist of two components: their *object identifier* and their *value*. Dot notation is used to access these parts. Thus, if *o* is an object, *o.oid* refers to the identifier and *o.value* extracts the value of *o*. If just *o* is written, the entire object is returned (i.e. its object identifier as well as its value).

Values of objects again may be structured and composed of other values or objects. If only a part of the entire value of an object *o* is to be retrieved (which is assumed to occur rather frequently), it would be uncomfortable to write *o.value.<part>* every time. Thus, it is allowed to omit the keyword *value*, i.e. if the value of *o* is defined as a tuple and a component *a* has to be extracted, one may write *o.a* to obtain the component *a* of the value of *o*. In order to avoid ambiguities, we assume *oid* and *value* as reserved names which are not allowed to be used as attribute names.

The dot notation using *oid* and *value*, respectively, is sufficient to "navigate" in complex objects. If a part of a complex object is "reached" in a query, it may be accessed further in the way described above.

A further question is how to deal with general references. Usually, references are represented by the object identifiers of referenced objects. Nevertheless, if references serve as intermediate results (i.e. if the user "navigates" along a general reference), one is interested in the referenced object and its value, not in the object identifier. Thus, we use dot notation for references as well, and a referenced object (its value, respectively) can be accessed in the way described above in the case of objects.
In our example, *department.name* accesses the values of the *department* objects, while *department.director.name* accesses the general reference *director* and then the values of the referenced objects (*director.name*).

5.3 Accessing Values

To a large extent, operators accessing (complex) values are taken from HDBL, most of which are also found in the O_2 query language [Bancilhon 90] in a similar form. The following subsections define operators for various kinds of value sets. Although most of the operators are illustrated by simple examples using atomic values only, they are also applicable to complex values or to the results of subqueries.

5.3.1 Set Operators

Set Construction

The set constructor specifies a constant set, e.g. in comparisons. It is denoted by set brackets ("{" and "}"). Thus, {0,1,2,3} constructs the set of the first four natural numbers.

Union, Intersection, Difference

The three set theoretic operators take two sets as an argument and result in a new set. *Union* computes the set theoretic union of two sets, e.g. {1,2,3} *union* {2,5} results in the set {1,2,3,5}.

Intersect takes two sets as arguments, too. It returns a set consisting of those elements contained in both input sets. Thus, {1,2,3} *intersect* {2,5} returns {2}.

Minus returns a set combined of elements of the first input set that are not elements of the second input set as well. For example, {1,2,3} *minus* {2,5} = {1,3}.

Although at first glance set theoretic operators look rather simple, there is one problem related to them. All three operators are based on a membership test (or equality test, to be more precise). In the case of *union*, e.g., the 2's in both sets are assumed to be equal. In consequence, the result contains the 2 only once. The situation gets less trivial when complex values or objects are regarded. In these cases, different kinds of equality may be used for the comparison of set elements (namely, deep or shallow equality in the case of complex values, additionally identity in the case of objects). We assume that identity is used in the case of objects (i.e. when dealing with sets of objects) and shallow equality in the case of complex values. For the different kinds of equality/identity see section 5.4.1 below.

Cardinality

The *card* operator determines the number of elements contained in a set. Thus, *card*({1,2,3}) returns 3.

Flatten

Flatten removes one level of nesting. Given a set of sets, the union of all element sets is returned.
Example:
    flatten ({ {<1,2>, <1, 2, 4>} {<2>} })
returns {<1,2>, <1, 2, 4>, <2>}.

Pick

Pick extracts an element of a singleton set, i.e. a set is converted into a value of its element type. *Pick*({2}) returns 2.

5.3.2 List Operators

List Construction

List Construction is denoted by "<" and ">", enclosing an arbitrary number of values separated by commas. Thus, <5, 3, 1> is a list of the first three odd natural numbers.

Accessing Lists

A list can be decomposed into two parts: its first element and in its remainder (the list except its first element). *L.first* returns the first element and *l.rest* returns the remainder of a list *l*, respectively. Thus, <5, 3, 1>*.first* returns 5 while <5, 3, 1>*.rest* returns <3,1>. As an abbreviation, elements of lists can be accessed by specifying the index of the desired element. The index is specified by a natural number enclosed in square brackets. E.g., <5, 3, 1>[2] returns 3 and is an abbreviation for <5,3,1>*.rest.first*.

The *sublist* operator provides for the extraction of a part of a list. Beside the input list, the starting element and the number of consecutive elements (i.e. the length of the sublist) must be specified. *Sublist*(2,3, <5, 3, 1, 4, 2>) extracts a sublist of three elements, starting at the second list element. In this case, the result will be <3, 1, 4>.
Sublist(2, *length*(l) - 1, *l*) is equivalent to *l.rest*.

Concatenation

The concatenation operator ("||") appends two lists and returns a new list as result. Thus, <5, 3, 1> || <4, 2> results in the list <5, 3, 1, 4, 2>.

Elements

Lists can be converted into sets. Applying the *elems* operator to a list means to create a set containing the elements of the list and to remove duplicates. Obviously, order is not maintained anymore. *Elems*(<5, 3, 1, 3>) results in {1, 3, 5}.

Length

The *length* operator computes and returns the number of elements of a given list. Thus, *length*(<5, 3, 1, 3>) returns 4.

Flatten

Like *flatten* for sets, list-*flatten* removes one level of nesting. Given a set or list of lists, the concatenation of the member lists is returned.

Example:
flatten (< <1,2>, <1, 2, 4>, <2> >) yields <1, 2, 1, 2, 4, 2>.

Index Lists

An index list is a list of natural numbers denoting the indices of a list. *indl*(<5, 3, 1>) returns <1, 2, 3>. Index lists are required if, e.g., every element of a list needs to be compared with its successor.

Example: let *l* be the list <9, 1, 7, 3, 5>. From *l*, we want to construct a list which contains only those elements of *l* which are greater than their successor:

select l[i]
from i *in indl*(l)
where i < *length*(l) *and* l[i] > l[i+1].

Result: <9, 7>.

5.3.3 Tuple Operators

Tuple Construction

In order to construct tuples, attribute names and corresponding values have to be specified. E.g., [department: "IFI", address: "Zürich"] constructs a tuple of two components. Attribute names have to be unique within one tuple. The values of attributes need not necessarily be constant, and a *select* statement may also be specified. E.g. [director: "Ch. Brown", subordinates: *(select ... from ... where)*] is a valid tuple construction.

Accessing Attributes

Components of tuples are extracted using the dot notation. If *t* is a tuple with attribute *a*, *t.a* extracts the value of the component *a* in *t*. E.g., [department: "IFI", address: "Zürich"].department = "IFI". Alternatively, position numbers can be specified instead of attribute names. Then, [department: "IFI", address: "Zürich"].1 again returns "IFI".

Tuple Concatenation

A new tuple is obtained by concatenating two given tuples. This is a shorthand for first extracting each attribute of the two tuples, respectively, and then constructing a new one with the given attributes. E.g., the result of [department: "IFI", address: "Zürich"] || [director: "Ch. Brown", subordinates: {...}] is [department: "IFI", address: "Zürich", director: "Ch. Brown", subordinates: {...}].

5.3.4 Array Operators

Array Construction

NO^2 arrays are always of fixed length. In order to construct a one-dimensional array, a number of elements must be given. The number has to conform to the length specified in the corresponding value set definition. For example, [5, 3, 1] creates an array with length 3.

Arrays with a dimension greater than one can be constructed in a nested manner.

Accessing Arrays

Arrays can be accessed like lists, i.e. elements or subarrays can be extracted. [5, 3, 1][2] returns 3, while *subarray*(2, 2, [5, 3, 1]) returns [3, 1].

An index list operator is not supported, since the length of arrays is always determined beforehand.

5.4 Boolean Operators

Boolean operators provide for comparisons of values and objects and for the construction of formulas out of simpler ones. First we define the comparison operators (or *predicates*, in logical terms), then we describe how formulas can be combined to form more complex ones.

5.4.1 Predicates

Predicates usually take two values as arguments and return a boolean value (i.e. *true* or *false*).

Comparison Operators

As described in the NO^2 data definition language, there are different kinds of equality. Two objects can be tested for identity, shallow equality, or deep equality, while to values (depending on the kind of value set) atomic, shallow, or deep equality is applicable. While atomic equality is assumed to be the predefined equality between numbers, strings, etc., shallow and deep equality as well as identity are defined. Thus, if o_1 and o_2 are two objects,

- o_1 *identical* o_2 tests for identity,
- o_1 *deep-equal* o_2 tests for deep equality, and
- o_1 *shallow-equal* o_2 tests for shallow equality.

The latter two are also applicable to (complex) values. If just "=" is written, identity is assumed in the case of comparing objects and shallow equality in the case of complex values.

Quod furthermore provides for the usual comparison operators like <, >, etc. They can be applied to value sets having order defined on them (i.e. integers, reals, and strings).

Of course, operators will rather seldom compare two atomic values. Therefore *Quod* allows one or both operands of comparison operators to be *select* statements again.

Membership Operators

Membership tests can be applied to lists or sets. Thus, 1 *in* <5, 3, 1> and 1 *in* {0, 1, 2, 3} both return *true*. As in the case of comparison operators, one or two operands of the membership test may be *select* statements again.

Subset Operators

Subset and sublist comparison can be applied to sets or lists, respectively. Those sets or lists may in turn be results of (sub) queries. {0, 2} *subset-of* {0, 1, 2, 3} returns *true* as well as <5, 3> *sublist-of* <5, 3, 1> does. More precisely, *l sublist-of l'* holds, when there are numbers $0<n,m \leq length(l')$ such that *sublist*(*n, m, l'*) = *l.*

5.4.2 Formulas

As usual, predicates and formulas can be combined to form more complex formulas. *Quod* provides for the logical connectors *and*, *or*, and *not*.

Furthermore, formulas can be quantified. Universal as well as existential quantification is supported. Quantified variables can be bound to lists or sets, i.e., if *elem* is a variable, *set* denotes a set and *f* is a formula with free variable *elem*, then

forall elem *in* set*:* f

is a correct formula. The semantics of this formula is that *set* is selected only if all of its members satisfy the restriction specified by *f*. As noted before, quantified formulas may be applied to lists as well. If *exists* is used instead of *forall*, at least one element has to satisfy the given condition.

Example: All *research_groups* having no member with a PhD title.
select r
from r *in* research_group
where forall m *in* r.members:
not(m.title = PhD)

HDBL furthermore allows for specification of cardinalities in quantified formulas, i.e. the upper and lower bounds of the number of elements satisfying the restriction is expressible. Since we do not want to make *Quod* too complex, we do not provide such features.

5.5 Built-in Operators

Up to now, we have described the core features of the *Quod* query language. Further operators are required, e.g., for sorting or grouping lists.

Arithmetic operators

Arithmetic operators take lists or sets of values as arguments and perform computations like adding integers, etc. The supported operators include *sum*, *min, max,* and *avg*.

Sorting

Lists can be sorted, or, more precisely, their ordering can be changed in that a new ordering criterion is specified. Sorting can be applied to sets as well, in this case the type of the result is a list with the specified ordering criterion. The syntax of ordering is

order <variable>
in <set_or_list>
by <values> [*asc* | *desc*].

The first parameter defines a variable ranging over the list specified by the second parameter. The third parameter describes the values used for sorting. These values must be of a type supporting ordering (e.g. integers or strings, but not references). Usually the values used in the *by* clause will refer to the defined variable. Example:

order x
in <5, 3, 1>
by x *asc*
returns <1, 3, 5>.

Let *s* be the list of sets < {0, 4}, {}, {1, 2, 4, 8, 16}>. Then,

order x
in s
by *card*(x) *asc*
returns <{}, {0, 4}, {1, 2, 4, 8, 16}>.

Grouping

Grouping means to partition sets or lists according to a given criterion. The syntax of grouping is similar to that of sorting:

group <variable>
in <set-or-list>
by <value>.

The first and the second parameter again specify a variable and a set or list respectively. The third parameter specifies the grouping criterion. All elements of the specified list or set having the same value under the group criterion are partitioned into the same subset or -list.

Example:
group x
in {<1,2>, <2>, <1, 2, 4>}
by x[1].

Result: { {<1,2>, <1, 2, 4>} {<2>} }.

5.6 Queries and Inheritance

Since NO^2 provides for inheritance, the question arises how the results and correctness of queries are affected by that feature. For example, assume a user queries for all *researchers*. Since each *assistant is_a researcher*, the extension of *assistants* should be contained in the result as well as *professors*. The general rule is that a type is examined or accessed in a query, if one of its (direct or indirect) supertypes is queried.
Example: The names of all *researchers*.

select name
from researcher.

Result: {“Charly Brown”, “Lucy”, “Snoopy”, . . ., “Pig Ben”}.

5.7 More Examples

Names of all departments, their directors and the number of their research groups:

select [dept: d.name,
boss: d.director.name,
group_cnt: *card*(d.groups)]
from d *in* department.

Result: { ...., [“Computer Science”, “Charly Brown”, 2], ...}.

Assume that *members* are ordered in *research_group* according to the time effort they spend in the group. Then, find all groups where the head works more than the rest of the members:

select g
from g *in* research_group
where g.head = g.member.*first*.

Result: A singleton set containing the “Implementation” group.

All researchers, who are both *member* and *head* of *some* (not necessarily the same) group:

select r
from r *in* researcher
where *exists* g *in* research_group: g.head = r
and
exists g *in* research_group: r *in* g

Result (*researchers* are represented by their names): {“Snoopy”, “Lucy”, “Pig Ben”}.

Assistants who join more than one group:

select assi
from assi *in* assistant
where *card* (*select* g
from g *in* research_group
where assi *in* g.member)
> 1.

This query yields “Lucy” as results.

5.8 Query Definition

NO^2 schemas may become rather complex, and thus queries may get hard to read. *Quod* therefore provides for a mechanism to define parameterized, named queries and to incorporate them in other queries later on. This approach is quite useful especially if queries contain common subqueries. The syntax is

define query <name> [(<parameters>)] *as* <query>.

The query identified by <name> may then occur anywhere a query is allowed. In this case, <name> is textually substituted by <query>.

6 Recursive Queries

Complex object structures can be used to represent different levels of abstraction of real-world entities. For instance, if we want to know all details about the "Computer Science Department", we have to consider the corresponding object with all its subobjects. Furthermore, the degree of abstraction may be varying according to the depth of nesting *research_groups* are considered.

Modelling concepts like *is_part_of* hierarchies are rather useless unless adequate query and manipulation facilities exist. Unfortunately, up to now queries do not support the consideration of different levels of abstraction in the case of recursive types (e.g., *research_group*, which may contain subobjects of the same type). While we are able to retrieve e.g. the entire "Computer Science Department" object (and thus all its subobjects as well), we are not able to retrieve all direct and indirect *sub_groups* of that object in <u>one</u> set. Specific levels (e.g. of *is_part_of* hierarchies) can be considered using nested queries. Nevertheless, when object types are defined to be recursive, the depth of nesting cannot be determined (statically). In consequence, *Quod* provides for *recursive queries*.

Recursion in object-oriented query languages has been implemented and proven successful [Schiefer 89, Schoening 89]. As well as these approaches, *Quod* does not provide for arbitrary recursion, but for construction of (generalized) transitive closures (of references).

In order to formulate a recursive query, a (cyclic) path in the type graph has to be specified. Restrictions can be added to the different nodes in such paths. The result of a recursive query can either be

- the transitive closure, i.e. all objects reachable via the specified path from the specified start objects, or
- the generalized transitive closure, i.e. the transitive closure together with the "history" [Schiefer 89] of the reachable objects.

In our example, the transitive closure comprises all *groups* and *sub_groups* of the "Computer Science Department":

{ooDBS, oo Data Models, Implementation, Database Theory}.

The generalized transitive closure would consist of a set of lists:

{ <ooDBS>,
<Database Theory>,
<ooDBS, oo Data Models>,
<ooDBS, Implementation> }.

In the following, we describe recursive queries by means of some examples.
Example: all *research_groups* being a (direct or indirect) group of any department:

select g
from r *in* department.groups,
g *in* {r.sub_groups} *.

The set bracket in the *from* clause specifies that the transitive closure is desired. As a condition it is required that the first and the last element of the path enclosed by the brackets are of the same type (*research_group* in the example). The meaning of the "*" is explained below. The variable *r* will represent the direct groups of all departments. The computation of the transitive closure will start with those direct groups, compute the *sub_groups* of them, and so on.

Example: only the groups of the "Computer Science Department" are desired:

select g
from d *in* department,
r *in* d.groups,
g *in* {r.sub_groups} *
where d.name = "Computer Science".

Now, let us assume that the "Implementation" group has *subgroups* again. This time, however, we ask for all groups of the "Computer Science Department" without going into all available details, i.e. we ask for *sub_groups* which have a depth of nesting less or equal than 2:

select g
from d *in* department,
r *in* d.groups,
g *in* {r.sub_groups} 2
where d.name = "Computer Science".

The number in the from clause specifies up to which level the transitive closure has to be computed. If a "*" is given, the entire transitive closure will be computed.

As a next example, we are interested in the generalized closure: for each element of the transitive closure, we would like to obtain the corresponding path and the nesting depth in the *group*s hierarchy.

select [groups: <g>,
depth: *length*(<g>)]
from d *in* department,
r *in* d.groups,
g *in* {r.sub_groups} *
where d.name = “Computer Science”.

Result:
{ [<ooDBS>, 1],
[<Database Theory>, 1],
[<ooDBS, oo Data Models>, 2],
[<ooDBS, Implementation>, 2] }.

The “< >” brackets enclosing *g* specify that the generalized transitive closure is required.

Finally, the names of all *researchers* working in any *research_group* of the “Computer Science Department” is desired. Additionally, the usage of defined queries is shown:

define query recursive_query
as *select* g
*from*d *in* department,
r *in* d.groups,
g *in* {r.sub_groups} *
where d.name = “Computer Science”.

select name
from researcher
where researcher *in* recursive-query

7 Update Facilities

Quod not only provides for query constructs, but also for update facilities. These include insertion, modification, deletion, and migration of objects.

7.1 Insert

Insert creates new objects, checks the type correctness of their values, and adds them to the extension of the specified type. The syntax is

insert <value>
into <extension>.

The parameter <value> must specify a value allowed for the value set of <extension>. Especially, <value> in turn may contain queries again. Extensions are denoted by the name of the respective type (i.e. there is exactly one extension per type).
Example:

```
insert [ name: “Peppermint Betty”,
         address: “IFI”,
         title: PhD]
into  assistant.
```

Further example: the creation of a new *research_group* having all *assistants* with a *Dipl* title as *members.*

```
insert [ name:       “dipl_group”,
         head:        pick (select  d.director
                            from    d in department
                            where   d.name = “Computer Science”)
         members:    order x
                     in     (select a
                            from    r in recursive_query,
                                    a in r.members
                            where   a.title = Dipl)
                     by     x.name asc,
         sub_groups: { } ]
into  research_group
```

7.2 Update

Update modifies the value of one or more objects. Again, type correctness of modified values and objects is checked. The syntax of *update* is:

```
update <variable> in <extension>
set    <path> = <value>
where  <restriction>.
```

Example: Pig Ben gets a new title:

```
update   r in researcher
set      r.title = Dr
where    r.name = “Pig Ben”.
```

7.3 Delete

The delete operator (*delete*) removes objects from an extension. The set of objects to be removed is specified in the *where* clause.

```
delete  <variable> in <extension>
where   <condition>.
```

Example: The groups containing at least one member with a *Dipl*-title:

```
delete  r in research_group
where   exists m in resarch_group.members:
          m.title = Dipl.
```

The semantics of the delete operator partially depends on the schema definition. If an object to be deleted possesses dependent subobjects, those are deleted as well. If, in our example, *members* are dependent subobjects of *research_group*, *members* will be deleted as well.

7.4 Migrate

Migration means a change of the type of an object without explicitly creating a new object, i.e. the identity of the concerned object remains the same while its value changes and the object is added to a new and removed from its old extension. If an object migrates by means of user-specified *delete-insert*-sequences, a new identity would replace the old one and thus, the entire context of the object gets lost. This is cumbersome if a migrating object is referenced by other objects before and after its migration. In this case the user would have to set multiple references again.
Syntax:

migrate <variable> *in* <extension>
to <extension>
[*value* <value>]
[*where* <condition>].

The first (optional) parameter defines a variable representing the object(s) to be migrated. The second parameter specifies the old extension while the third one denotes the new one. <value> is the new value of the migrating object (it must then be valid for the new extension). The where clause is optional and restricts the set of migrating objects.

Example:
migrate p *in* person
to professor
value [name: person.name,
address: person.address,
title: *Prof*,
repertoire: {“food”}]
where person.name = “Snoopy”.

A further question to be clarified for the migration of objects is the relationship between the old and the new type of a migrating object, i.e., is migration to arbitrary types possible? Obviously, if the old and the new type are located “far from each other” in the type lattice, many references to a migrating object may get unvalid. Furthermore, in this case there would be little benefit for the user, since many references and a great part of the value of the migrating object had to be altered. Even if the new type of a migrating object is a supertype of the old one, some references to the object might get uncorrect. On the other hand, if the new type is a subtype of the old one (i.e., if the migrating object is “specialized”), due to substitutability, all references to the object stay correct. Thus, we restrict migration to those cases where the new type of an object is a subtype of its old one.

Note that the system has to guarantee implicit consistency constraints not to be violated as a result of migration. First, if an attribute of the value set of the old type was refined (*value restriction*) in the definition of the new type, parts of the value of the object may get inconsistent. Second, if *structure extension* was applied in the definition of the new type of the migrating object, at least those attributes must be given values which were defined as *required* or *unique*.

8 Data Definition Facilities

The NO^2 query language also provides for data definition constructs. The syntax of object type and value set definitions is the same as described above, regardless whether type definitions are entered interactively (e.g., via the *Quod* interface) or obtained from a (batch) DDL-program. Thus, the syntax of data definitions is not repeated here; the reader is referred to the appendix.

There is one additional point concerning the definition of types in an interactive way. In contrast to SQL, type definitions may depend on each other, i.e. types may only be well-defined if other types are defined as well, especially if subobject or general references are defined. Even cycles in the "uses"-graph of type definitions may occur. Consequently, in order to check the consistency of type definitions, sets of definitions have to be regarded instead of single definitions, and the language must provide for constructs telling the system when type definitions start and when they are completed.

Grouping related definitions is possible by the use of *schema transactions* (called *DDL-transactions* in [Mitschang 88]). The command *begin-schema-definition* opens a schema transaction. This operator requires one argument, which has the meaning of a schema name. *end-schema-definition* finishes a schema transaction, while *abort-schema-definition* stops such a transaction and rolls all definitions back. Schema transactions support atomicity, consistency, isolation, and durability.

Part 3: An Algebra for NO^2

In the history of relational database systems, the development of formal foundations has preceded the design and implementation of prototypes and commercial systems. A generally accepted theory existed to which the various systems conformed more or less, and therefore it was justified to speak of *the* relational model. In contrast, the development of object-oriented database systems started with system designs and implementations right away. Only now, as numerous prototypes and commercial systems exist, it is tried to build a theory. However, *the* object-oriented data model does not exist up to now, and since most theories address specific (already existing) systems, there is no common formal object-oriented data model.

Algebras are one possibility to describe a theoretical foundation and may be used in different ways:

- as formal semantics of data and of a query language,
- as a framework for optimization purposes,
- as a query language itself.

If an algebra is used in the first way, the operational part (i.e. the operators) provides for a formal language to express the results of queries. The structural part describes the data to be stored in a database which have to conform to the respective data model. Both parts are defined in mathematical terms, thus describing semantics clearer than examples or natural language could do. As a consequence, looking at different formal models, it may turn out that different terminology is used to describe the same things, and that sometimes people use the same notions but attribute different meanings to them. For example, the need to support "complex objects" is widely agreed upon [Atkinson 89], but there are several meanings of the term. Some (e.g., NO^2) use the term complex object in combination with *is-part-of* relationships, others (e.g. O_2 [Lecluse 89a]) in combination with a general referencing mechanism. A third group (e.g., ORION [Kim 89b]) assumes the same semantics like NO^2, but uses other notions ("composite objects" in the case of ORION). In this situation, formalization may help to figure out the concrete semantics of a specific terminology and the differences between various approaches.

Algebras have been used for optimization mainly in the case of conventional relational systems (*algebraic optimization*). In this case, one is interested in algebraic laws (e.g., associativity, commutativity) valid for specific operators, allowing algebraic expressions to be transformed in semantically equivalent, but more efficiently executable ones.

Finally, an algebra may be used as a query language itself. Nevertheless, since many people would regard algebras as being not very comprehensive, algebras are not supported as a query language in commercial systems.

This part addresses the first point: a formalization of the NO^2 data model and a future query language by means of an algebra. Although one might draw conclusions concerning optimization, this issue is not addressed in this paper (see [Demuth 91]).

In particular, the main concepts of NO^2 to be reflected by the algebra include:

- objects, values, and their distinction,
- complex object structures,
- arbitrarily nested and structured values,
- type hierarchies with inheritance.

The clear distinction between values and objects is an important feature. In [Shaw 90], results of queries (or algebraic expressions) are always objects, and in [Kim 89a] “anything” is an object. As a consequence, objects have to be generated as results of queries, and problems arise since the same query applied several times does not necessarily return identical results (due to object identity), and the “location” of results in the class lattice has to be determined. Thus, like [Cluet 89], we will distinguish objects from values and allow queries to extract objects and values or to restructure values, but we will not provide for newly created objects as results of queries.

Next, we will formalize the notion of complex object structures and distinguish between subobjects and general references among objects, while most other algebras only deal with one kind of object association.

Another crucial point is the complete freedom in nesting and structuring complex values. In NO^2, complex values can be arbitrarily nested, and thus a regular structure of the data to be dealt with cannot be assumed.[8] While most algebras provide for sets (and tuples) only, both, sets and lists will be provided, and lists will be “first class” values in the same sense as sets are. Furthermore, operators are provided that access partial values without assuming any regular structure.

Type hierarchies influence the results of queries since subtypes have to be involved in queries against their supertypes. For instance, if a user asks for all instances of a type *person*,

---

8. NF^2 relations, for instance, can be termed *regular*, since relations are sets of tuples, where component values in turn can either be atomic or relations again, but nothing else.

he/she would expect to obtain all instances of type *student* as well if *student* is defined as a subtype of *person*.

Our formalization will result in a *many-sorted algebra* (see [Goguen 78] for a formal definition). After an overview on algebras for other data models or query languages, we will describe the domain (section 10) and the operators of the NO^2 algebra (section 11). Section 12 then summarizes the definition. In section 13, some shorthands and useful extensions are described. While in section 14 different kinds of consistency constraints are formalized, section 15 finally shows some sample queries.

Throughout this paper we will give example queries. The corresponding schema is shown in Figure 3. A sample database is depicted in Figure 4. The object of type *department* will be denoted *d,* the object of type professor *p.* The research groups will be abbreviated by g_1 . . . g_4 ("ooDBS", "Database Theory", "OO Data Models", and "Implementation"), and the assistants will be named a_1 up to a_5.

9 Comparison to Other Algebras

Algebraic foundations have been proposed for various data models. Obviously, the relational algebra [Ullmann 88, Maier 86] has influenced many others developed later, and this is also true for the NO^2 algebra.

One extension of the relational algebra is the NF^2 algebra [Schek 86, Jaeschke 85], where attributes are allowed to be themselves relation-valued. [Gueting 89] describes an algebra for nested *lists* (or sequences) of tuples. In comparison to [Schek 86], sets are replaced by lists. The algebra of [Gueting 89] is claimed to be closer to a query language than other algebras, since operators offered by most query languages like *sort* or *count* [e.g., Date 83] have been formalized. This is due to the provision of lists which allows operators like *sort* to be defined, which would not make sense for sets as results. Next, the algebra of [Gueting 89] is *many-sorted* and contains numbers as one specific sort. In consequence, operators like *count* that map lists to integer values can be expressed in this algebra, while they cannot in the relational or NF^2 algebra (since their domains consist of sets only).

One common characteristic of all algebras mentioned above (and a difference to NO^2) is the *regularity* of data structures: values to be manipulated by the operators are always lists or sets of *tuples*, whose components in turn may consist of sets or lists, respectively. In NO^2, on the other hand, values may be arbitrarily nested and constructors for complex values (tuple, list, set, or array constructors) are completely orthogonal.

The data structures defined in the algebra of [Abiteboul 88] are (like those of NO^2) completely orthogonal. In this algebra tuple and set constructors are provided and may be arbitrarily nested. Thus, we borrowed *filters* from that algebra as a generic mechanism to deal with sets (whose elements in turn may be sets, tuples, or atomic values).

A further difference of all the algebras described above is that NO^2 is (structurally) object-oriented [Dittrich 90b]. In contrast, the algebras mentioned above do not distinguish objects and values nor include complex object structures and type hierarchies[9].

There have been several approaches for object-oriented data models. [Mitschang 89] describes an algebra for the structurally object-oriented data model MAD [Mitschang 88]. In this approach, objects are either *atoms* (comparable to tuples) or *molecules*. Molecules are represented by graphs whose nodes are atoms and whose edges represent *links* between atoms. While in NO^2 complex object types are defined statically (in the schema), MAD allows for the (static) definition of atom types and links, and provides for the dynamic definition of molecule types based on atoms and links between them. Since molecules are represented by bidirectional links, the structure of complex objects (and the level of nesting of subobjects) can only be recognized by comparing different graphs.

[Shaw 90] proposes an algebra for a behaviorally object-oriented system. This algebra can be termed *object-generating* since results of queries are always objects. In consequence, identical queries do not have identical results (although the respective results will have equal values, the resulting objects will have different identities in each query, respectively), and new operators for duplicate elimination have to be defined. The algebra of [Scholl 89] does not always create new objects, i.e. the creation of objects as results of queries depends on the choice of the operators (e.g. object *preserving* and object *creating* joins are provided).

A further problem of object-generating algebras connected with type hierarchies is the location of the results in the class lattice. A solution provided by [Kim 89a] is to locate the class of the results as a direct subclass of the top element of the lattice. [Scholl 89] gives more specific rules where to locate result classes in the class lattice (depending on the operator and the input classes).

Since “everything” is an object in [Kim 89a], queries have to result in objects again. [Cluet 89], on the other hand, describes an algebra for the O_2 data model [Lecluse 89a], which distinguishes (like NO^2) objects and values. In consequence, there is no need to generate objects as results. The approach of [Cluet 89] allows for the extraction of *existing* objects or (parts of) their values. In neither case, new objects will have to be (automatically) created.

10 The Algebra Domain

Loosely speaking, a (many-sorted) algebra can be understood as a collection of sets together with a set of operators. The first is called the *domain* (or structural part, carrier), while the operators are also referred to as the operational part. A requirement for the operators is the closure condition, i.e. parameters and results of operators are required to be el-

9. Although the nested sets and tuples of the algebra of [Abiteboul 88] are called “complex objects”, those are no objects in the sense of NO^2 (due to the lack of object identity [Khoshafian 86]).

ements of a set of the domain. This section and the next one introduce the individual parts of the NO^2 algebra in turn.

10.1 Objects and Object Types

Objects are pairs (object_identifier, value). Object identifiers uniquely identify an object. They remain unchanged over the whole lifetime of an object and are not reused even if the object does not exist any more [Khoshafian 86].

> **Definition 10.1** (Object)
> An *object* is a pair (object_identifier, value). For any two different objects their identifiers are different, too. For a given object *o*, the function *oid* returns the identifier of *o*, while the function *oval* returns the value of *o*.

Objects with values having the same structure are collected into object types. Objects of a given type are also called *instances* of that type. A description of allowable values of possible instances is associated with an object type. Thus, a value set defines the *intension* of an object type.

> **Definition 10.2** (Object Type)
> An object type is a pair (ot_name, value_set), where *ot_name* is the object type's name and *value_set* is a set of values allowed for instances of the type. *Ot_name* has to be unique among all object type names.

> **Definition 10.3** (Extensions)
> The *extension* of an object type is the set of its instances exactly at a given point in time. The result of the function *ot_ext* applied to an object type results in the corresponding extension. For a given object type name, the function *otype* returns the corresponding object type (see Definition 10.2).

Note: different object types with equal intensions, but disjoint extensions may exist. Thus, unique names are required to distinguish those types.

In the following, we write *VS* for value sets, and *v* for values (elements of a given value set). Given the name of an object type (say, *ot*), the corresponding extension is obtained by applying the function *ot_ext* to *otype(name)*. In the sequel, we abbreviate the expression *ot_ext*(*otype(ot_name)* by just writing *ot_name*.

Note that we just deal with the *structural* part of objects (and of values, in the next subsection) and treat the operational aspects later, while sometimes structure *and* operators together are regarded to define a type (e.g., ADT's).

10.2 Values and Value Sets

Value sets may be either basic or complex. They are defined by either enumerating their members, or by means of *constructors* that specify how members may be composed out of other values. Basic value sets include: *integer, real, boolean, string,* etc. Furthermore, there is a basic value set *object_identifier*, or *OID* for short. Complex value sets are built by means of set, list, tuple, and array constructors.

Definition 10.1 (Lists)
If *VS* is a value set, then *list(VS)* is a value set. *Nil* is the empty list and is a member of *list(VS)*. Beside *Nil*, *list(VS)* consists of values $<v, vl>$, where v is any element of *VS* and *vl* is already an element of *list(VS).* The value v is then called a *member* of $<v,vl>$. *List(VS)* is said to be of list type.

A list of the first three natural numbers is represented as $<1, <2, <3,\ NIL > > >$. Sometimes we simply write $<1, 2, 3>$ for short.

Definition 10.2 (Sets)
If *VS* is a value set, then *set(VS)* is also a value set. The elements of *set(VS)* are values $\{v_1, \ldots, v_n\}$, where $n > 0$ and v_i is an element of *VS* for $1 \leq i \leq$ n. Furthermore, $\{\}$ (the empty set) is an element of *set(VS)* for all *VS*. *Set(VS)* is said to be of set type.

Definition 10.3 (Tuples)
If $VS_1, \ldots, VS_n$ are value sets, $a_1, \ldots, a_n$ are (attribute) names and $a_i \neq a_j$ for $i \neq j$, then $tuple(a_1\!: VS_1, \ldots, a_n\!: VS_n)$ is a value set. This value set is said to be of tuple type. The elements of the tuple value set are values $[a_1\!:v_1, \ldots, a_n\!:v_n]$, where v_i is an element of VS_i. The a_i are called *attributes*, and the v_i are called *attribute values*.

Note that *tuple* constructs single tuples, but not sets of tuples (*relations*). If a relation is to be constructed, *set* construction has to be explicitly applied to the *tuple* constructor: a set of tuples (of atomic values) is a relation in the usual sense [Date 83].

Definition 10.4 (Arrays)
If *VS* is a value set and n is an integer, then *array* $[n]$*(VS)* is a value set. A*rray* $[n]$*(VS)* consists of values $[v_1, \ldots, v_n]$, where v_i is an element of *VS*. The v_i are called *components* of $[v_1, \ldots, v_n]$.

Note: for simplicity, we provide for one-dimensional arrays of fixed length (*static arrays*) only. Multi-dimensional arrays can be modelled by nesting arrays (i.e. applying the *array* constructor to arrays). For convenience, shorthand notations may be defined for this case.

10.3 Object Structures

In order to allow the modelling of complex object structures, we regard object types as additional value sets. Furthermore, sets of references to (existing) objects are treated as basic value sets, too. They are intended to represent general associations between objects. In order to model these general associations, we use an additional constructor *REF*.

Definition 10.1 (General References)
If *ot* is an object type and *OID* is the set of object identifiers, then
$VS = REF(OID)$ is a valid value set. If *o* is an instance of *ot*, then *ref (oid(o))* is a member of *VS*. *Ref*(*oid(o)*) is called a *general reference.*

Apart from general references, subobjects have to be modelled in the algebra domain. Subobjects are built into the value of the parent object, i.e., when an object acts as a subobject it is treated like a value.

Definition 10.2
If *ot* is an object type, then $ot \cup \{nil\}$ is a valid value set.

An object is a subobject of another object, if it is "contained" in the value of the latter, i.e., if the value of the parent object is a list, array, set, or tuple (recursively) constructed from the subobject and possibly some other values. The next two definitions formalize the notion of structured objects and subobjects.

Definition 10.3
An object *o* is *contained* in a value *v*, if
1. $o = v$, or
2. *v* is a list, array, set, or tuple value, *o* is contained in a value *v*', and *v*' is a component, element, or attribute value of *v*.

Definition 10.4 (Subobjects)
An object o_2 is a *subobject* of object o_1, if
1. o_1 has value *v* and o_2 is contained in *v*, or
2. o_2 is subobject of an object o_3 and o_3 is subobject of o_1.

o_1 is called a *structured object,* if it has at least one subobject, otherwise it is called a *simple object.* O_2 is also called a *component* (object) of o_1.

Objects can be direct (case 1 of the definition) or indirect (case 2 of the definition) subobjects of other objects. In the first case, we say that there is a *subobject reference* from the object to the subobject. In the latter case, they are a member of the transitive closure of the subobject relationship. While the transitive closure of general references may contain cycles, that of subobject references may not, since this would express the fact that an object is "contained" in itself. Therefore the *nil* object was added to an extension in Definition 10.2. In the case of recursive types, where an object may contain subobjects of the same type, cycles can be prevented in that the *nil* object is referenced.

Finally, the transitive closure of subobject references is used to define the notion of *shared subobjects*.

Definition 10.5 (Shared Subobjects)
An object o is called a *shared object* (of objects o_1 and o_2), if
1. *o* is a subobject of o_1 as well as of o_2, and

2. o_1 is not a subobject of o_2 and vice versa.
The objects o_1 and o_2 are then said to *overlap.*

We conclude this section with a definition of valid value sets, which are used in the definition of the NO^2 algebra domain.

Definition 10.6 (Value Sets - Summary)
The following value sets are valid:

1. all basic value sets,
2. complex value sets:
   a. lists with value sets as component value sets,
   b. sets with value sets as component value sets,
   c. arrays with value sets as component set,
   d. tuples with distinct attribute names and value sets as attributes.
3. object type extensions, and
4. references of object type extensions.

These are the only valid value sets.

The collection of all value sets V_i is called the *domain* of the algebra.

A set of existing extensions can be regarded as a *database*. Distinct from a database is a *schema*, which is a set of corresponding object types and which describes the allowable instances of types. Furthermore, a database is time dependent (since extensions may change over time).

10.4 Treatment of Specialization and Type Hierarchies

Type hierarchies express the *is-a* relationship between object types. If we want to extract all *persons* from our database, we expect to obtain the instances of *researcher* in addition to the *person* instances, since every *researcher* is thought to be also a *person*. Multiple approaches might be followed with respect to specialization relationships. A query might be expanded to a union of queries: one against the original type and one against each (direct or indirect) subtype, respectively. Nevertheless, in this approach the *is-a* relationship stays outside the algebra. A further solution would be to state the specialization relationship between object types explicitly within the algebra. However, denoting relationships between object types would lead to a more calculus-oriented approach. Therefore, we adopted the following solution [Beeri 89]: beside its “own” instances an object type extension contains the instances of all its subtypes as well. Specialization relationships then still stay “outside” the algebra, but for a given domain a transformation can be specified which results in a domain without specialization, but with the desired semantics. Following this approach, instances of subtypes are involved in queries against their supertypes automatically. In order to obtain extensions where values are all of the “same” type, structure extension and attribute renaming has to be undone. As a result, an object can be contained in different extensions and may have different values, depending under what specific type it is regarded.

In our example, object a_1 (the *assistant* named “Lucy”) *is_a researcher* and *is_a person*. Object a_1 has *tuple(name : “Lucy”, address: ..., title: Dipl)* as its value. As outlined above, conceptually there are two other objects, which are members of the extensions of *researcher* and *person*, respectively. The instance of *researcher* (say, r_1) has the same value as a_1, the instance of *person* has the value *tuple(name : “Lucy”, address: ...)*. In the following, an *assistant* a_i is denoted by r_i when regarded in the context of the *researcher* type.

11 Operators

This section describes the operational part of the NO^2 algebra. Operators known from many algebras (e.g. [Ullmann 88, Schek 86]) will be extended and some new operators will be defined. While in the case of the relational and the NF^2 model, structures are rather regular (i.e., sets of tuples or nested sets of tuples), there may be arbitrarily composed structures in NO^2. Furthermore, most algebras only deal with sets as first class values, while lists are not provided. In our case, however, access of (parts of) values and objects structured in any possible way is required.

In principle, there are two alternatives to introduce and group the operators: according to the kind of value set they are defined for, or according to their functionality. We decided for the second alternative and grouped the operators into the following sets:

- constructors,
- basic set and list operators (like set union),
- projection operators (providing for access to parts of complex structures),
- images (iterating over a set or list and simultaneously applying a given function to each element),
- selection,
- restructuring (and reordering) operators.

The notion we use for the definition of operators is as follows:
op: $VS_1 \times \ldots \times VS_n \rightarrow VS_{n+1}$ is called a *signature* and means that operator op has arity n and that it maps $v_i \in VS_i$ into an element of VS_{n+1}. $VS_1, \ldots VS_n$, VS_{n+1} may be any value set (basic or complex) permitted by the algebra domain.

11.1 Constructors

The first group of operators are constructors to build composite values. Since they have already been introduced in the section on the algebra domain, we just repeat their signature here.

Definition 11.1 (Constructors)
$\{\ \} : VS \rightarrow set(VS)$.

$< > : VS \times list(VS) \rightarrow list(VS).$

$[\,]_{a1, \ldots, an} : VS_1 \times \ldots \times VS_n \rightarrow tuple(a_1: VS_1, \ldots, a_n: VS_n).$

$[]: VS^n \rightarrow array[n]\ (VS)$

Note that the set constructor is a unary operator (in order to assign a fixed arity to it), i.e. it creates singleton sets. Sets containing more than one element can be created by the use of the set constructor in combination with set union. Thus, set construction applied to 1 returns {1}, and a set of three numbers is written $\{1\} \cup \{2\} \cup \{3\}$. Nevertheless, we will write {1,2,3} for short.

11.2 List and Set Operators

As in most algebras, set theoretic operators like union and difference are provided. The concatenation of lists corresponds to set union. In the case of sets, we assume that union and difference are already defined on the domain. Difference is not defined for lists, if this operation has to be performed, the lists have to be converted to sets explicitly.

Definition 11.1 (Set Theoretic Operators)
$\cup : set(VS) \times set(VS) \rightarrow set(VS).$
$\setminus : set(VS) \times set(VS) \rightarrow set(VS).$

The next operator (*card*) returns the number of elements in a set:

Definition 11.2 (Cardinality)
$card: set(VS) \rightarrow integer.$
$card\ (\{\}) = 0,$
$card\ (\{v\} \cup vs) = card\ (vs \setminus \{v\}) + 1.$

Uniting lists actually means to concatenate or append lists. Thus, in contrast to set union, list union is not commutative.

Definition 11.3 (List Operators)
$+_L : list(VS) \times list(VS) \rightarrow list(VS).$
$v +_L NIL = NIL +_L v = v.$
$<v, vl> +_L wl = < v, vl +_L wl >.$

Since some operators apply to sets but not to lists, it may be necessary to convert a list into a set (over the same value set). This is provided by the operator *members*, which transforms a list into a set consisting of all the elements of the list. As usual duplicates are eliminated; note, however, that “elimination of duplicates” inherently relies on a notion of equality (see below for various forms of equality and identity).

Definition 11.4 (List Members)
members: $list(VS) \rightarrow set(VS)$.
members $(NIL) = \{ \}$,
members $(v, l) = \{v\} \cup members(l)$.

Example:
all researchers that are members of the research group g_3, but not of g_4:
members($<r_1, r_2>$) \ members($<r_1, r_3>$) = $\{r_2\}$.

The next operator (*length)* is the correspondence to set cardinality, it returns the length of a list, i.e., the number of elements forming the list.

Definition 11.5 (Length)
length: $list(VS) \rightarrow integer$.
length $(NIL) = 0$,
length $(<v, vl>) = length(vl) + 1$.

11.3 Projection Operators

In the relational algebra, projection is rather simple: (sets of) tuples are restricted to (sets of) tuples containing less attributes than the original ones. Even in the NF^2 case, projection is rather simple since the structure of data is regular. In NO^2, the situation is more complicated due to several reasons:

- there are different kinds of structured values: tuples, which are structured by use of names, and lists and arrays, which are structured by the ordering among their elements. Additionally, objects are structured since they are pairs of identifiers and values.
- The structure of complex values is not regular and values can be arbitrarily nested.

These two requirements will be addressed in this and the next subsection. The first issue is that projections have to be defined for various kinds of value sets, i.e. tuples have to be projected to attributes, lists to their first elements and the remainder, and so on. Furthermore, it must be possible to nest projections arbitrarily.

Definition 11.1 (Projection)
$\pi_T [a_i] : tuple(a_1: VS_1, \ldots, a_n: VS_n) \rightarrow VS_i$
$\pi_T [a_i]([a_1: v_1, \ldots, a_n: v_n]) = v_i$, if $1 \leq i \leq n$.

$\pi_F : list(VS) \rightarrow VS$.
$\pi_F (<v, vl>) = v$.

$\pi_R : list(VS) \rightarrow list(VS)$.
$\pi_R (<v, vl>) = vl$.

$\pi_A : integer \times array[n] (VS) \rightarrow VS$.
$\pi_A (i, [v_1, \ldots, v_n]) = v_i$, if $1 \leq i \leq n$.

π_F and π_R are comparable to operators in functional programming languages (e.g. *car* and *cdr* in Lisp [Steele84]). Given the list of members (say, *l*) of the “Implementation” group, π_F (l) = r_1 and π_R (l) = $<r_3>$.

Projection operators are required for objects, too. One operator returns the identifier of an object, another one the corresponding value. A third operator takes a reference as parameter and returns the referenced object.

Definition 11.2 (Projection of Objects)
$\pi_O : OT\_EXT \rightarrow OID$
$\pi_O\ (o) = oid(o)$.

π_V : *OT_EXT* → VS, where *OT_EXT* is an object type extension and *VS* is the corresponding value set.
$\pi_V\ (o) = oval(o)$.

$\pi_D : REF(OT\_EXT) \rightarrow OT\_EXT.$
$\pi_D\ (ref(\pi_O\ (o))) = o.$

π_D can be regarded as a “dereferencing” operator.
Examples:
the title of the assistant named “lucy” (a_1):
π_T [*title*] ($\pi_V(a_1)$) = Dipl,
name of the computer science department’s director:
π_T [*name*] (π_D (*ref*(π_O (π_T [*director*] (π_V (*d*))))) = “Charly Brown”.

Note that we could have defined objects as tuples with attributes *oid* and *value*. In consequence, we would not require special projection operators for objects. However, following this approach it would be possible to “construct” objects by using the tuple constructor instead of explicitly creating them. In consequence, when looking at such tuples, it would not be possible to decide whether they are “original” objects or tuples defined by the user as the result of a query. In order to avoid this problem, we hide the layout of objects from the user and provide special operators to access identifiers and values of objects.

11.4 Images

Up to now, we are able to project structured values to one of their component. However, we also deal with sets or lists of structured values. For example, given an object type extension, we want the values of all objects contained in that extension. Given a set or list of anything, we want to perform a function on every element and collect the results in a new set or list, respectively. Operators taking other operators as arguments, applying the specified operator to the elements of a further parameter, and collecting the results in sets or lists again, are called *filters* [Beeri 89, Abiteboul 88] or *images* [Shaw 90]. For functions, we use the notation of [Shaw 90], which is also known from lambda calculus. Using this notation, we have solved a further problem: how to “name” elements of a set or a list. If a specific projection has to be performed on all elements of a given set, there has to be a "variable" ranging over all elements of the set or list in order to express the desired projection. In our notation, $\lambda s.\pi_V(s)$ expresses a projection to the value of an object. *S* denotes the argument of the function and is substituted by actual parameters when applied to a concrete value.

Definition 11.1 (Image)
$\iota_S\ [\lambda s.\ f(s)] : set(VS) \rightarrow set(VS')$, where $\lambda s.\ f(s)$ induces a function from *VS* to *VS'* and *f* is a sequence of (algebraic) operators.
$\iota_S\ [\lambda s.\ f(s)\ ]\ (v) = \{\ x\ /\ \exists\ y \in v \wedge x = f(y)\ \}$.

$\iota_L\ [\lambda s.\ f(s)\ ] : list(VS) \rightarrow list(VS')$, where $\lambda s.\ f(s)$ induces a function from *VS* to *VS'* and *f* is a sequence of (algebraic) operators.
$\iota_L\ [\lambda s.\ f(s)\ ]\ (NIL) = NIL$.
$\iota_L\ [\lambda s.\ f(s)\ ]\ (\ <v, vl>\ ) = <f\ (v), \iota_L\ [\ \lambda s.\ f(s)]\ (vl)\ >$.

Examples:
the projection to the corresponding value for all instances of a given extension *ex*:
$\iota_S\ [\ \lambda s.\pi_V(s)\ ]\ (ex)$,
names of all researchers:
$\iota_S\ [\ \lambda s.\ \pi_T\ [name]\ (\pi_V\ (s)\ )]\ (researcher)$ = {"Charly Brown", "Lucy", . . ., "Pig Ben"}.

11.5 Selection

Selection can be regarded as a special case of *filtering*, since it maps sets into sets (or lists, respectively) [Abiteboul 88]. Selection operators are defined for sets and lists. We define selection by means of predicates first and show how more general formulas can be expressed afterwards.

Predicates have the form v_1 *cop* v_2, where *cop* is a binary comparison operator like e.g. equality. Equality of basic values is assumed to be predefined as usual, but there are various possibilities for equality of objects and, in consequence, for references and complex values containing any kind of references. For example, what does it mean for objects to be "equal"? Equality of object identifiers or equality of values? We distinguish between identity and (shallow and deep) equality of objects, and shallow and deep equality of values. In the case of basic values or complex values without references, shallow and deep equality are equivalent.

Identity of objects is rather simple: it just means that surrogates of objects are equal ("equal" in the sense of basic values).

Definition 11.1 (Identity of Objects)
Two objects o_1 and o_2 are identical ($o_1 =_{id} o_2$), if $oid(o_1) = oid(o_2)$.

For values, there are multiple possibilities to define equality. The two chosen are shallow and deep equality. They differ in the way how they incorporate references. Shallow equality means to compare references showing up as components in complex values. That is, if there are two sets of subobject references, the sets are shallow equal if each object referenced in one set is also referenced in the other. Deep equality is more general: different objects may be referenced, but referenced objects in turn must be deep equal. Objects are deep or shallow equal if their values are.

Definition 11.2 (Shallow Equality)

1. Two objects o_1 and o_2 are shallow equal ($=_s$), if $oval(o_1) =_s oval(o_2)$.
2. Two values v_1 and v_2 are shallow equal, if one of the following properties holds:
   a. both values are atomic and $v_1 = v_2$,
   b. both values are references $ref(oid(o_1))$ and $ref(oid(o_2))$, and $o_1 =_{id} o_2$,
   c. both values are complex, they have identical structures, and all corresponding components are shallow equal, i.e.:
      i. if $v_1 = [a_1: u_1, \ldots, a_n: u_n]$, then $v_2 = [a_1: w_1, \ldots, a_n: w_n]$ and $u_i =_s w_i$, $1 \le i \le n$,
      ii. if $v_1 = \{ u_1, \ldots \}$ and $v_2 = \{ w_1, \ldots \}$, then for each $u \in v_1$ there exists $w \in v_2$ such that $u =_s w$, and for each $w \in v_2$ exists $u \in v_1$ with $u =_s w$,
      iii. if $v_1 = \langle v, vl \rangle$, then $v_2 = \langle w, wl \rangle$, $v =_s w$ and $vl =_s wl$,
      iv. if $v_1 = [u_1, \ldots, u_n]$, then $v_2 = [w_1, \ldots, w_n]$ and $u_i =_s w_i$, $1 \le i \le n$.

Definition 11.3 (Deep Equality)

1. Two objects o_1 and o_2 are deep equal ($=_d$), if $oval(o_1) =_d oval(o_2)$.
2. Two values v_1 and v_2 are deep equal, if one of the following conditions holds:
   a. both values are atomic and $v_1 = v_2$,
   b. both values are references $ref(oid(o_1))$ and $ref(oid(o_2))$, and $o_1 =_d o_2$,
   c. both values are complex values and are deep equal by their components (for a more formal definition replace $=_s$ by $=_d$ in case 2.c in the definition above).

Other comparison predicates include comparison of numbers, subset and element relationships for sets and lists.

Definition 11.4 (Selection Predicates)

If $_{v1}$ and $_{v2}$ are values $\in VS$, then $v_1\ cop\ v_2$ is a selection predicate, where $cop \in \{=, \neq, =_{id}, =_s, =_d, <, \le, \in, \subset, member, sublist\}$. These are the only selection predicates.

Now we are able to define selections for sets:

Definition 11.5 (Set Selection)

$\sigma_S[\lambda s.\ p] : set(VS) \rightarrow set(VS)$, where p is a selection predicate with possible free variable s.

$\sigma_S[\lambda s.\ p](v) = \{ x \mid x \in v \wedge p(x) \}$.

According to the definition above, selection expressions can only contain predicates, but no formulas (conjunctions of predicates, etc.). However, it is easy to extend the definitions of selections to include formulas as well:

Definition 11.6 (Formulas in Set Selections)
$\sigma_S\,[\lambda s.\, f]\,(v) = \sigma_S\,[\lambda s.\, g]\,(v) \cap \sigma_S\,[\lambda t.\, h]\,(v)$, if $f = g \wedge h$,
$\sigma_S\,[\lambda s. g]\,(v) \cup \sigma_S\,[\lambda s.\, h]\,(v)$, if $f = g \vee h$,
$v \setminus \sigma_S\,[\lambda s.\, g]\,(v)$, if $f = \neg\, g$.

Before we can define selection for lists, predicates that are specific to lists are introduced. First, the element (or member) relationship can be defined. Second, a *sublist* predicate is introduced. Both are in analogy to the set case.

Definition 11.7 (Member)
$member\,(v, NIL)$ = false,
$member\,(v, <w, wl>) = true$, if $v = w$,
$member(\,v, wl)$, else.

The equality used in this definition is understood as identity, if v is an object, and as shallow equality, if v is a complex value.

Definition 11.8 (Sublist)
$sublist\,(vl, NIL) = false$,
$sublist\,(NIL, vl) = true$,
$sublist\,(<v, vl>, wl) =$ $true$, if $\exists\, ul, zl$, such that $wl = <ul, <v, vl>, zl>$,
$false$, otherwise.

Finally, selection of lists is specified. Again, we get a definition similar to sets.

Definition 11.9 (Selection for Lists)
$\sigma_L\,[\lambda s.\, p] : list(VS) \rightarrow list(VS)$, where p is a selection predicate.
$\sigma_L\,[\lambda s.\, p\,]\,(NIL\,) = NIL$,
$\sigma_L\,[\lambda s.\, p]\,(<v, vl>) = <v, \sigma_L\,[\lambda s.\, p]\,(vl)>$, if $p(v)$,
$\sigma_L\,[\lambda s.p]\,(vl)$, otherwise.

Selection of lists can be extended to capture formulas instead of predicates only.

Definition 11.10 (Formulas in List Selection)
$\sigma_L\,[\lambda s.\, f\,]\,(<vf, vl>\,) =$ $<\mathrm{vf}, \sigma_L\,[\lambda s.\, f]\,(\mathrm{vl})>$, if $f = g \wedge h$ and $h(vf) = g(vf) = true$,
$<\mathrm{vf}, \sigma_L\,[\lambda s.\, g]\,(\mathrm{vl})>$, if $f = \neg\, g$ and $\mathrm{g(vf)} = false$,
$\sigma_L\,[\lambda s.\, \neg\,(\,\neg g \wedge \neg h\,)\,]\,(v)$, if $\mathrm{f} = g \vee h$.

Example:
all researchers having a PhD as title:
$\sigma_S\,[\,\lambda s.\, \pi_T\,[title]\,(\pi_V\,(s)\,) = \mathrm{PhD}]\,(researcher) = \{r_2, r_4\,\}$.

11.6 Restructuring Operators

An commonly required operation is the elimination of one level of nesting: for example, a query results in a set of sets while a set is required. The operator transforming a set of sets into a set is usually called *flatten* [Shaw 90] or *collapse* [Cluet 89]. Similar to other operators, *flatten* is provided for lists as well as for sets. The formal definition of *flatten* (ϕ) is:

Definition 11.1 (*Flatten* for Sets)
$\phi_S : 2^{set(VS)} \rightarrow set(VS)$
$\phi_S (v) := \{ x / \exists\, w : w \in v \wedge x \in w \}$.

Definition 11.2 (*Flatten* for Lists)
$\phi_L : list\,(list\,(VS\,)\,) \rightarrow list(\,VS\,)$.
$\phi_L\,(NIL) = NIL$.
$\phi_L\,(\,<vl,\,vll>\,) = vl +_L \phi_L\,(vll)$.

The first two definitions show *flatten* for sets of sets and for lists of lists, respectively.
Example:
subgroups of the research groups:
$\iota_S\,[\lambda s.\ \pi_T\,[sub\_groups]\,(\pi_V\,(s)\,)](research\_group) = \{\ \{g_3,\,g_4\},\,\varnothing\}$.
Thus, the result is a set of sets. If we assume E to be the above expression and ask for ϕ_S (E), the result will be $\{g_3,\,g_4\}$.

Up to now we are able to transform lists into sets, but there is no operator transforming sets into lists (which is also a kind of restructuring). The first problem is that an ordering has to be given if a list has to be constructed. Thus, converting sets into lists (or lists into other lists) is provided by the *sort* operators. Sort operators are applicable for sets or lists, in any case the result is a list again. The second question is the ordering relation used by the *sort* operators: naturally, one does not only want to sort integers or strings, but also objects or set (which in turn have no ordering defined on them). Therefore, one parameter of *sort* specifies a function which returns the value relevant for sorting.

Definition 11.3 (Sorting for Lists)
Let $VS = list(\,VS')$, f be a function from VS' to VS'' and $\leq_{VS''}$ be a total ordering on VS''. Then, $asc\text{-}sort_L[\lambda s.\,f(s)]$ is an operator from VS to VS.
$asc\text{-}sort_L[\lambda s.\,f(s)]\,(l) = l'$, where l' is a permutation of l and $f(l[i]) \leq_{VS''} f(l[j])$ for $1 \leq i \leq j \leq length(l)$.

Similarly, let $VS = list(\,VS')$, f be a function from VS' to VS'' and $\geq_{VS''}$ be a total ordering on VS''. Then, $desc\text{-}sort_L[\lambda s.\,f(s)]$ is an operator from VS to VS.
$desc\text{-}sort_L[\lambda s.\,f(s)]\,(l) = l'$, where l' is a permutation of l and $f(l[i]) \geq_{VS'} f(l[j])$ for $1 \leq i \leq j \leq length(l)$.

The lambda-expression enclosed in square brackets defines a function returning values which in turn are responsible for the resulting ordering. Sorting can be extended for sets in a straightforward way. It is then applied to sets, while the result is always a list. Thus, sorting is a mechanism to convert sets into lists.

Definition 11.4 (Sorting for Sets)
Let $VS = set(\ VS')$, f a function from VS' to VS'' and $\leq_{VS''}$ be a total ordering on VS''. Then, $asc\text{-}sort_S[\lambda s.\ f(s)]$ is an operator from VS to VS.
$asc\text{-}sort_S[\lambda s.\ f(s)]\ (l) = l'$, where l = members(l') and $f(l[i]) \leq_{VS''} f(l[j])$ for $1 \leq i \leq j \leq card(l)$.

Let $VS = set(\ VS')$, f a function from VS' to VS'' and $\geq_{VS''}$ be a total ordering on VS''. Then, $desc\text{-}sort_S[\lambda s.\ f(s)]$ is an operator from VS to VS.
$desc\text{-}sort_S[\lambda s.\ f(s)]\ (l) = l'$, where l = members(l') and $f(l[i]) \geq_{VS''} f(l[j])$ for $1 \leq i \leq j \leq card(l)$.

Example: sorting the set of all *research_groups* according to their name:
$asc\text{-}sort_S\ [\lambda s.\ \pi_T\ [name]\ (\pi_V(s)\ )\ ]\ (research\_group)$.
Result: <g2, g4, g3, g1>, according to their names ("Database Theory", "Implementation", "OO Data Models", "ooDBS").

12 The Algebra

With the definitions of value sets and the operators as given above, we now can define the algebra in its entirety. Intuitively, the algebra consists of a collection of value sets and a collection of operators.

Definition 12.1 (Algebra)
The NO^2-Algebra is a pair (D, O), where
1. D is the domain of the algebra (a collection of value sets) as defined in section 10,
2. O is the set of operators

$\{\{\},<>,[],[]\}$	(the constructors),
$\cup\ \{\ \cup, \backslash, card, +_L, members, count\}$	(the basic set and list operators),
$\cup\ \{\pi_T, \pi_A, \pi_F, \pi_R, \pi_D, \pi_O, \pi_V\ \}$	(the projection operators),
$\cup\ \{\iota_S, \iota_L\ \}$	(the image operators),
$\cup\ \{\sigma_S, \sigma_L\ \}$	(the selection operators),
$\cup\ \{\phi_S, \phi_L$, asc_sort$_S$, desc_sort$_S$, asc_sort$_L$, desc_sort$_L\}$	(the restructuring operators).

The operators are also summarized in Table 2.
As already mentioned, the NO^2 algebra is a many-sorted one, which supports multiple domains and thus the ability to model operators that cannot be formalized in "usual" algebras. Essentially, a many-sorted algebra is essentially a family of sets A_i (called the *carriers*) together with a collection of operators (functions) among them. More formally, the algebra can be defined as follows [Goguen 78]. First, an S-sorted signature Σ which is a family of $\Sigma_{w,s}$ sets, with $w \in S^*$ and $s \in S$ is given, where S can be viewed as a set of sorts (i.e. names). If an operator σ is an element of $\Sigma_{w,s}$, then the sorts denoted by w identify the

input sort, while s identifies the result sort. A Σ-algebra A then consists of a set A_s (the carrier of A of sort s) for each $s \in S$ and a function σ_A: $A_{s1} \times \ldots \times A_{sn} \rightarrow A_s$ for each $\sigma \in \Sigma_{w,s}$ with $w = s_1 \ldots s_n$. Thus, S contains the names of the sets A_i, and $\Sigma_{w,s}$ defines names and signatures of operators.

For the NO^2 algebra, S is the set of all value set names, and $\Sigma_{w,s}$ is the set of operator symbols. Defining an operator op: $VS_1 \times \ldots \times VS_n \rightarrow VS$ in the algebra means that op is an element of Σ_{ws}, where $w = VS_1 \ldots VS_n$ and s = VS. Writing a definition like $op(v) = \ldots$ in the algebra means to define the corresponding function σ. Thus, we did not distinguish between the operator names and the corresponding functions σ, and we identified the value sets with their names.

	object	tuple	set	list	array
constructor		[ ]	{ }	< >	[ ]
basic operators			$\cup$ \ card	$+_L$ members length	
projection	π_O, π_V, π_D	π_T		π_F, π_R	π_A
image			ι_S	ι_S	
selection			σ_S	σ_L	
restructuring			ϕ_S, asc_sort$_S$ desc_sort$_S$	ϕ_L asc_sort$_L$ desc_sort$_L$	

Table 2: Summary of NO^2 algebraic operators

13 Some Goodies

This section describes some useful abbreviations of and extensions to existing operators. The first abbreviation will lead to the definition of *paths*.

13.1 Paths

Since projections of structured objects and values to partial value(s) nested some levels deep may be cumbersome to write and read, it is desirable to abbreviate (nested) applications of projection operators. *Paths* are a means to specify (sequences of) projections. In case of tuples we use the well-known dot-notation: $t.b$ denotes projection of a tuple t to the component b. Thus, $t.b = \pi_T\,[b]\,(t)$. Elements of lists can be specified by the list and an index: $l\,[i]$ denotes projection of the list l to the i^{th} member. Thus, $l[i] = \pi_F(\pi_R^{\;i-1}\,(l))$. The same notation is used for arrays. For objects, dot notation is also used: if o is an object, *o.oid* denotes its identifier and *o.value* denotes its value. If we assume that “oid” and “value” are forbidden to be chosen as attribute names, then writing *o.aname* is permitted as a shorthand for *o.value.aname*.

We furthermore allow image operators to occur in paths: “{ }” then denotes the set image operator while “< >” denotes the list image operator. Inside the brackets the abbreviations described above are allowed

13.2 Cross Products and Joins

Cross Products are defined for lists and sets of tuples. In the latter case, the set theoretic semantics carry over. Since cross products are required to realize joins (which are less important than in the relational case, but are nevertheless useful), we decided to support joins for lists (and thus require cross product for lists). Before we define cross products, we introduce an abbreviation for tuple concatenation. Tuples can be concatenated by a combination of tuple projections and tuple constructions. For better readability, we define an operator $+_T$ concatenating two tuples and use it afterwards in the definition of cross products.

Definition 13.1 (Tuple Concatenation)
Let $U = tuple\ (a_1\!: VS_1, \ldots, a_n\!: VS_n)$ and $V = tuple(a_{n+1}\!: VS_{n+1}, \ldots, a_{n+m}\!: VS_{n+m})$. Then,
$+_T : U \times V \rightarrow tuple(a_1\!: VS_1, \ldots, a_n\!: VS_n, a_{n+1}\!: VS_{n+1}, \ldots, a_{n+m}\!: VS_{n+m})$.
$[\ a_1\!: v_1, \ldots, a_n\!: v_n]\ +_T\ [\ a_{n+1}\!: v_{n+1}, \ldots, a_{n+m}\!: v_{n+m}] = [\ a_1\!: v_1, \ldots, a_{n+m}\!: v_{n+m}]$.

Definition 13.2 (Cross Product for Sets)
Let $U = tuple(a_1\!: VS_1, \ldots, a_n\!: VS_n)$, $V = tuple(a_{n+1}\!: VS_{n+1}, \ldots, a_{n+m}\!: VS_{n+m})$, and $W = tuple(a_1\!: VS_1, \ldots, a_n\!: VS_n, a_{n+1}\!: VS_{n+1}, \ldots, a_{n+m}\!: VS_{n+m})$. Then $\times_S$ is a function from $set(U) \times set(V) \rightarrow set(W)$.
$u \times_S v = \{\ s +_T t\ /\ s \in u \wedge t \in v\ \}$.

Definition 13.3 (Cross Product for Lists)
$\times_L : list(VS_1) \times list(VS_2) \rightarrow list(VS_1 \times VS_2)$, where VS_1 and VS_2 are tuple types with disjoint attribute names.
$vl \times_L NIL = NIL \times_L vl = NIL$.
$<v, vl> \times_L <w, wl> = <v +_T w, <v, NIL> \times_L wl> +_L vl \times_L <w, wl>$.

Of course, joins can be expressed by the algebra’s operators. Like in the relational case, a join can be expressed by a cartesian product followed by a selection, which compares the join attributes. Since both cartesian product and selection were defined for lists as well, lists of tuples can be joined in the NO^2 algebra.

14 Consistency Constraints

Although unusual in algebraic foundations for data models, we are able to express some consistency constraints which can be specified in the NO^2 data definition language (DDL).

Referential Integrity

The first (inherrent) consistency constraint refers to referential integrity: any referenced object has to be an instance of the respective object type extension. In other words, referenced objects have to exist in the database, and dangling references are prohibited.

> **Definition 14.1** (Referential Integrity)
> Let *OT* and OT' be object types with extensions *OT_EXT* and *OT_EXT*', respectively. Furthermore, let *OT*' be a referenced by *OT*. Then the following condition has to hold:
> ∀ ob OT_EXT: ob references ob' and ob' has type *OT'* ⇒ ob' ∈ OT_EXT'.

Uniqueness Constraints

Assume that parts of the value set of an object type are defined as unique. Those parts then have to be atomic and may be components of a tuple value set, but not of a set, list, or array. Thus, unique components are "reachable" by a sequence of value and tuple projection. Furthermore, combinations of components may be declared as unique. Thus, the part of the value of an object which has to be unique can in turn be represented as a tuple. Then, for any two (not identical) objects of an extension, the corresponding tuples have to be different.

> **Definition 14.2** (Uniqueness Constraints)
> Let *OT* be an object type with extension *OT_EXT*. Furthermore, let *L* be a list <e1,...en> where e_i is a sequence of projection operators, which returns one of the unique components when applied to an object of *OT_EXT*. Then the following condition has to hold:
> $\forall\, o_1, o_2 \in \text{OT\_EXT}: [a_1: e_1(o_1),..., a_n: e_n(o_1)] = [a_1: e_1(o_2),..., a_n: e_n(o_2)] \Rightarrow o_1 =_{id} o_{2}$..

As an example, assume that the *address* attribute in *department* were defined as a tuple with components *street, town, country,* and *phone*, and that *town* is of type *string*. Furthermore, there may not be two *departments* with the same *name* in the same *town*. Thus, the combination of *name* and *town* is declared as unique. Consequently, if there are two objects o_1 and o_2 of type *department* with

$$[\ a_1: \pi_T[\text{ name }]\ (\pi_V\ (o_1)\ ),\ \ a_2: \pi_T[\text{ town }]\ (\ \pi_T[\text{ address }]\ (\pi_V\ (o_1)\ )\ )\ ] =$$
$$[\ a_1: \pi_T[\text{ name }]\ (\pi_V\ (o_2)\ ),\ \ a_2: \pi_T[\text{ town }]\ (\ \pi_T[\text{ address }]\ (\pi_V\ (o_2)\ )\ )\ ],$$

then o_1 and o_2 have to be identical.

Sharability and Exclusiveness

Sharability and exclusiveness of subobjects can be viewed as implicit consistency constraints. If a subobject reference was declared as exclusive, components may not be shared by different complex objects.

> **Definition 14.3** (Exclusiveness)
> Let OT_1, OT_2 and OT_3 be three (not neccesarily distinct) object types and *VS* be the corresponding value set of OT_1 . Then, if OT_2 is contained in the definition of *VS* and declared as exclusive, the following condition has to hold:
> $\forall\ o_1 \in OT\_EXT_1, o_2 \in OT\_EXT_2, o_3 \in OT\_EXT_3$:
> o_2 is not a shared subobject of o_1 and o_3.

For the formal definition of *shared subobject*, see Definition 10.5.

15 Examples

In this section we will give some sample queries. In the examples we will use paths.
The first query is "all heads of a *research_group* who are also a member of the respective group".

$\iota_S\ [\lambda t.\ \pi_D\ (t.head)\ ]\ (\sigma_S\ [\lambda s.\ \pi_D\ (s.head)\ member\ s.members\ ]\ (\text{research\_group})\ )$.

The result will be $\{r_2, r_1, r_5\}$.

In the next query, we want to know the names of all research groups of the computer science department together with the names of the heads. First we select the *groups* of the computer science department (E_1), and then construct the result tuples.

E_1: $\phi_S\ (\iota_S\ [\lambda \text{v. v.groups}\ ]\ (\sigma_S\ [\ \lambda t.\ t.name = \text{"Computer Science"}]\ (\text{department})\ )$

$\iota_S\ [\ \lambda s.\ [\text{group: s.name, head: } \pi_D(\ \text{s.head}\ ).\text{name}]\ (E_1\ )$.

The result is { ["group: "ooDBS", head: "Charly Brown"], ["Database Theory", head: "Pig Ben"] }. Note that we do not get *indirect* groups of the computer science department. In order to get *all* groups, we had to be able to formulate transitive queries (which has not currently been defined for the NO^2 algebra).

Next, we want to know the names of all researchers working together.

$$E_2: \iota_S\ [\ \lambda s.\ \iota_L\ [\ \lambda t.\ [\ member: t.name]\ ]\ (s.members) \\ \times_L \\ \iota_L[\ \lambda u.\ [works\_with: u.name]\ (s.members)\ ]\ (research\_group)$$

The result of E_2 is a set of lists of tuples. If we apply flatten to it, the result is a list.

E_3: ϕ_{SL} (E_2)

Next, we can eliminate pairs of equal names.

σ_L [$\lambda v.$ *v.member* $\neq$ *v.works_with*] (E_3)

The final result is { [*member*: "Lucy", *works_with*: "Snoopy"], [*member*: "Snoopy", *works_with*: "Lucy", . . .}. Finally, we want to know the department directed by "Charly Brown":

E_4: σ_S [λs. *s.name* = "Charly Brown"] (*professor*)

The result of E_4 is a singleton set containing *p*.

ι_S [$\lambda t.$ *t.name*] (σ_S [$\lambda u.$ π_D(*u.director*) $\in$ E_4] (department).

The result will be {"Computer Science"}.

16 Conclusion

In this report, we have described the structurally object-oriented data model NO^2. The data definition language supports rich data modelling capabilities, including complex objects, complex values and type hierarchies. General references and subobjects allow for the modelling of various semantics of object structures, whereby specific consistency constraints on (sets of) complex objects like exclusiveness and dependency can be expressed. The set of constructors for complex values is completely orthogonal, which also contributes to the modelling power of NO^2.

A query language has been introduced that supports the retrieval of objects and values. Additionally, operators for insertion, update, deletion, and migration of objects have been presented. This data manipulation language is declarative, thus supports the descriptive, complex-value oriented access to objects quite well. In principle, the query language is similar SQL, such that the necessary effort to get started is expected to be small.

Furthermore, an algebra has been defined. To that end, we opted for a many-sorted algebra, which is not only able to cope with the rich data structures of NO^2, but also permits to represent more operators as conventional algebras do (e.g., sorting, cardinality). Currently, this algebra is used for three distinct purposes: it serves as a formalization of data structures and queries, operator trees conforming to it are input to the CoOMS server [Dumm 91], and algebraic laws are the basis for algebraic optimization.

As mentioned above, CoOMS is currently under implementation at SNI. Simultaneously, work on persistent CooL (based on CoOMS and NO^2) has been started. Nevertheless, for real-world applications features beyond the core data model are required. We have dealt with these features elsewhere, namely with transaction management [Geppert 92a], a view mechanism [Geppert 92b], and schema evolution [Scherrer 92].

17 References

Abiteboul 88
: S. Abiteboul, C. Beeri: *On the Power of Languages for the Manipulation of Complex Objects.* Technical Report 846, INRIA, France, 1988.

Atkinson 89
: M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik: *The Object-Oriented Database Manifesto.* Proc. First Intl. Conf. on Deductive and Object-Oriented Database Systems (DOOD), Kyoto 1989.

Bancilhon 89
: F. Bancilhon: *Query Languages for Object-Oriented Database Systems*: *Analysis and a Proposal.* In T. Härder (ed.): Proc. Datenbanksysteme in Büro, Technik und Wissenschaft. IFB 204, Springer 1989.

Bancilhon 90
: F. Bancilhon, S. Cluet, and C. Delobel: *The O_2 Query Language Syntax and Semantics.* Technical Report 45-90, Altaïr 1990.

Banerjee 87
: J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, D. Woelk and N. Ballou: *Data Model Issues for Object-Oriented Applications.* ACM Transactions on Office Information Systems 5:1, 1987.

Beech 88
: D. Beech: *A Foundation for Evolution from Relational to Object Databases.* In J.W. Schmidt, S. Ceri, and M. Missikoff (eds.): Advances in Database Technology - EDBT '88. Lecture Notes in Computer Science 303, Springer 1988.

Beeri 89
: C. Beeri: *Formal Models for Object-Oriented Databases.* Proc. First Intl. Conf. on Deductive and Object-Oriented Database Systems (DOOD), Kyoto 1989.

Bernstein 87
: P.A. Bernstein: *Database System Support for Software Engineering*. Proc. 9. Intl. Conf. on Software Engineering, Computer Science Press, 1987.

Carey 88
: M.J. Carey, D.J. DeWitt, and S.L. Vandenberg: *A Data Model and Query Language for Exodus.* Proc. ACM-SIGMOD Intl. Conf. on Management of Data, Chicago, 1988.

Cluet 89
: S. Cluet, C. Delobel, C. Lécluse, and P. Richard: *Reloop, an Algebra Based Query*

Language for an Object Oriented Database System. Proc. First Intl. Conf. on Deductive and Object-Oriented Database Systems (DOOD), Kyoto 1989.

Codasyl 78
Codasyl Data Description Language Committee Report. Information Systems 3:4, 1978.

Date 83
C.J. Date: *An Introduction to Database Systems.* Volume II, Addison Wesley, 1983.

Demuth 91
B. Demuth, A. Geppert, T. Gorchs: *Algebraic Query Optimization in the CoOMS Structurally Object-Oriented Database System*. J.C. Freytag, D. Maier, G. Vossen (eds.): Proc. Workshop on Query Processing in Object-Oriented, Complex-Object and Nested Relation Databases, Dagstuhl (Germany), June 1991 (to appear at Morgan Kaufmann Publishers, 1993).

Dittrich 85
K.R. Dittrich, A.M. Kotz, J.A. Muelle, P.C. Lockemann: *Database System Support for Design Applications*. Informatik Spektrum 8, 1985 (in German)

Dittrich 90a
K.R. Dittrich, A. Geppert, and V. Goebel: *The NO^2 Data Definition Language.* Project Report ITHACA.Unizh.90.X.4.#1, Institut für Informatik, University of Zürich 1990.

Dittrich 90b
K.R. Dittrich: *Object-Oriented Database Systems: The Next Miles of the Marathon.* Information Systems, 1990.

Dumm 91
T. Dumm, T. Gorchs, M. Watzek: *The CoOMS Persistent Object Server. Requirement Specification, Functionality, and Architecture*. Project Report ITHACA.-SNI.91.X.5#2, Siemens-Nixdorf Information Systems, 1991.

Elmasri 89
R. Elmasri, S.B. Navathe: *Fundamentals of Database Systems*. Benjamin/Cummings Publishing, 1989.

Geppert 90a
A. Geppert, K.R. Dittrich, V. Goebel, S. Scherrer: *Quod: A Query Language for NO2*. Project Report ITHACA.Unizh.90.X.4.#2, Institut für Informatik, University of Zuerich, 1990.

Geppert 90b
A. Geppert, K.R. Dittrich, V. Goebel: *An Algebra for the NO^2 Data Model.* Project

Report ITHACA.Unizh.90.X.4.#3, Institut für Informatik, University of Zürich, 1990.

Geppert 92a
A. Geppert, S. Scherrer, K.R. Dittrich: *Transaction Management in CoOMS*. Project Report ITHACA.Unizh.92.X.5#2, University of Zurich, 1992.

Geppert 92b
A. Geppert, S. Scherrer: *A View Mechanism for NO2*. Project Report ITHACA.Unizh.93.X.5#1, Institut für Informatik, University of Zuerich, 1992.

Geppert 93
A. Geppert: *The Syntax of the Database Language Quod*. Project Report ITHACA.Unizh.93.X.5#2, Institut für Informatik, University of Zuerich, 1993.

Goguen 78
J.A. Goguen, J.W. Thatcher, E.G. Wagner*: An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.* In R.T. Yeh (ed.): Current Trends in Programming Methodology, Vol IV. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

Gueting 89
R.H. Gueting, R. Zicari, D.M. Choy*: An Algebra for Structured Office Documents.* ACM Transactions on Office Information Systems 7:4, 1989.

Jaeschke 85
G. Jaeschke: *Recursive Algebra for Relations with Relation Valued Attributes.* IBM Heidelberg Scientific Center, Technical Report TR 85.03.002, 1985.

Khoshafian 86
S. Khoshafian, G. Copeland: *Object Identity.* Proc. Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1986.

Kim 89a
W. Kim: *A Model for Queries for Object-Oriented Databases.* Proc. Intl. Conf. on Very Large Data Bases (VLDB), Amsterdam 1989.

Kim 89b
W. Kim, E. Bertino, J.F. Garza: *Composite Objects Revisited*. Proc. ACM-SIGMOD Intl. Conf. on Management of Data, Portland 1989.

Lecluse 89a
C. Lecluse, P. Richard: *The O_2 Data Model.* Technical Report, Altair 39-89, Le Chesnay Cedex, 1989.

Lecluse 89b

C. Lecluse, P. Richard*: Modelling Complex Structures in Object-Oriented Databases.* Proc. ACM Symposium on Principles of Database Systems (PODS), 1989.

Lockemann 85

P.C. Lockemann, K.R. Dittrich, M. Adams, M. Bever, B. Ferkinghoff, W. Gotthard, A.M. Kotz, R.-P. Liedtke, B. Lueke, J.A. Muelle: *Database Requirements of Engineering Applications. An Analysis*. University of Karlsruhe, Internal Report 12/85.

Lockemann 87

P.C. Lockemann, K.R. Dittrich. *The Architecture of Database Systems.* In P.C. Lockemann, J.W. Schmidt (eds.): Datenbankhandbuch. Springer 1987 (in German).

Maier 86

D. Maier: *The Theory of Relational Databases.* Computer Science Press, Rockville 1986.

Mitschang 88

B. Mitschang: *A Molecule-Atom Data Model for Non-Standard Applications.* Informatik-Fachberichte 185, Springer 1988 (in German)

Mitschang 89

B. Mitschang:. *Extending the Relational Algebra to Capture Complex Objects*. Proc. Intl. Conf. on Very Large Data Bases (VLDB), Amsterdam 1989.

Pistor 85

P. Pistor, R. Traunmüller: *A Database Language for Sets, Lists, and Tables.* Technical Report 85.10.004, IBM Heidelberg Scientific Center, 1985.

Pistor 86

P. Pistor, F. Andersen: *Designing a Generalized NF^2 Model with an SQL-Type Language Interface.* Proc. Intl. Conf. on Very Large Data Bases (VLDB), 1986.

Schek 86

H.-J. Schek, M. H. Scholl: *The Relational Model With Relation Valued Attributes.* Information Systems 11:2 1986.

Scherrer 92

S. Scherrer, A. Geppert, K.R. Dittrich: *Schema Evolution in NO^2*. Project Report ITHACA.Unizh.93.X.5.#3, Institut für Informatik, Universität Zürich, December 1992.

Schiefer 89

B. Schiefer, S. Rehm*: A Query Language for a Structurally Object-Oriented Data Model.* In T. Härder (ed.): Proc. Datenbanksysteme in Büro, Technik und Wissenschaft. IFB 204, Springer 1989 (in German).

Schoening 89

H. Schoening: *Recursion in the MAD Model: Recursive Molecules as Data Model Objects.* In T. Härder (ed.): Proc. Datenbanksysteme in Büro, Technik und Wissenschaft. IFB 204, Springer 1989 (in German).

Scholl 89

M.H. Scholl, H.-J. Schek*: A Relational Object Model*. Submitted for publication, Dec. 1989.

Schröer 91

F.W. Schröer: *The CooL 0.3 Language Description*. Project Report ITHACA.SNI.90.L1.#3, Siemens-Nixdorf Informationssysteme, Berlin 1991.

Shaw 90

G.M. Shaw, S.B. Zdonik: *A Query Algebra for Object-Oriented Databases.* Proc. Intl. Conf. on Data Engineering, Los Angeles, 1990

Steele84

G.L. Steele: *Common Lisp - The Language*. Digital Press, 1984.

Ullmann 88

J.D. Ullman: *Principles of Database and Knowledge Systems. Volume 1.* Computer Science Press, Rockville 1988.

Appendix A: The Syntax of *Quod*

In this appendix, we describe the syntax of *Quod*. for that matter, we use the following conventions:

- "|" denotes alternatives,
- expressions enclosed in square brackets ("[...]") are optional,
- expressions enclosed by set brackets ("{ ... }") can be applied arbitrary often,
- for the sake of readability, keywords are written in upper case letters and are enclosed in quotes.

A.1 The Root

quod_statement ::=	schema_statement | dml_statement | ta_statement | access_statement

A.2 Schema Definition and Database Creation

schema_def ::=	"DEFINE LOGICAL SCHEMA" schema_name ddl_statement { ddl_statement } "END" schema_name | "DEFINE SUBSCHEMA" schema_name OF schema_name ot_name { "," ot_name } "END" schema_name "."
logical_db_def::=	"DEFINE LOGICAL DATABASE db_name "USING" schema_name END db_name "."

A.3 Data Definition

ddl_statement ::=	ot_def | union_def | vs_declaration

A.3.1 Object Type Definitions

ot_def ::=	"DEFINE OBJECT TYPE" ot_name ot_clause {unique_clause} "."
ot_clause ::=	vs_clause | ot_specialization
vs_clause ::=	"=" value_set
ot_specialization ::=	"SUPERTYPE IS" ot_name {"," ot_name } [";" conflict_clause] [extension_clause] [restriction_clause]
conflict_clause ::=	attribute_name "FROM" ot_name {"," attribute_name "FROM" ot_name}

extension_clause ::=	“EXTENDS” [attribute_spec] “BY” attribute_def {“,” attribute_def } {EXTENDS attribute_spec “BY” {“,” attribute_def } }
restriction_clause ::=	“REFINES” [attribute_name] “TO” vs
unique_clause ::=	“UNIQUE” “(“ attribute_spec {“,” attribute_spec} “)
attribute_spec ::=	attribute_name {“.” attribute_name}
union_def ::=	“DEFINE UNION TYPE” ot_name “=” “UNION” “(“ ot_name “,” ot_name {“,” ot_name} “).”

A.3.2 Value Set Definitions

value_set ::=	[REQUIRED] vs
vs ::=	basic_value_set | vs_definition | vs_name
vs_declaration ::=	“DEFINE VALUE SET” vs_name “=” vs_definition “.”
vs_definition ::=	tuple_def | list_def | array_def | set_def | ref_def | subobject_def | enum_def | sub_range_def
tuple_def ::=	“TUPLE” “(“ attribute_def “)”
attribute_def ::=	attribute_name “:” attribute_vs {attribute_name “:” attribute_vs}
attribute_vs ::=	value_set | attribute_name “FROM” ot_name
list_def ::=	“LIST” “(“ value_set “)”
array_def ::=	“ARRAY” “[“ dimensions “]” “OF” value_set
dimensions ::=	integer {“,” integer}
set_def ::=	“SET” “(“ value_set “)”
ref_def ::=	“REF” “(“ ot_name “)”
subobject_def ::=	properties ot_name
properties ::=	sharability_clause dependency_clause
sharability_clause ::=	“SHARABLE” | “EXCLUSIVE”
dependency_clause ::=	“DEPENDENT” | “INDEPENDENT”
enum_def ::=	“ENUM” “(“ name {“,” name} “)”
subrange_def ::=	“SUBRANGE” “[“ integer “.” “.” integer “]”

basic_value_set ::=	"INTEGER" | "REAL" "CHAR" | "STRING" | "BOOL" | "LONG_FIELD"

A.4 Data Manipulation

dml_statement ::=	select_statement | query_definition | insert_statement | update_statement | delete_statement | migrate_statement

A.4.1 Manipulation

insert_statement ::=	"INSERT" value "INTO" extension
update_statement ::=	"UPDATE" variable "IN" extension "SET" path "=" value [where_clause]
delete_statement ::=	"DELETE" variable "IN" extension [where_clause]
migrate_statement ::=	"MIGRATE" variable "IN" extension "TO" extension ["VALUE" value] [where_clause]

A.4.2 Queries

select_statement ::=	select_clause from_clause [where_clause] | sort_statement | group_statement
select_clause ::=	"SELECT" select_expression
select_expression ::=	variable | complex_value | operation
operation ::=	projection | set_operation | list_operation | tuple_operation | array_operation | comptoint_operation | agg_operation
comptoint_operation ::=	"CARD" "(" value ")" | "LENGTH" "(" value ")"

agg_operation ::=	“MAX” “(“ value “)” | “MIN” “(“ value “)” | “AVG” “(“ value “)”
set_operation ::=	“FLATTEN” “(“ value “)” | “PICK” “(“ value “)” | “UNION” “(“ value “,” value “)” | “INTERSECT” “(“ value “,” value “)” “DIFFERENCE” “(“ value “,” value “)”
list_projection ::=	”FIRST” “(“ value “)” | ”REST” “(“ value “)” | value ”[“ number “]”
list_operation ::=	“SUBLIST” “(“ number “,” number “,” value “)” “||” “(“ value “,” value “)” | “ELEMS” “(“ value “)” | “FLATTEN” “(“ value “)” | “INDL” “(“ value “)”
tuple_operation ::=	value “.” name
array_operation ::=	value “(“ number “)”
complex_value ::=	tuple_expression | set_expression | list_expression | array_expression
tuple_expression ::=	“[“ name “:” value {“,” name “:” value} “]”
set_expression ::=	“{“ “}” | “{“ value {“,” value } “}”
list_expression ::=	“NIL” | “<“ value {“,” value } “>”
array_expression ::=	“[“ value {“,” value} “]”
value ::=	operation | atomic_value | complex_value | “(“ select_statement “)” | variable
atomic_value ::=	number | real_number | character | string | “TRUE” | ”FALSE”
from_clause ::=	“FROM” in_clause {“,” in_clause}
in_clause ::=	variable “IN” value | variable “IN” “RECURSIVE” recursive_clause

recursive_clause ::=	“{“ path “}” depth
path ::=	projection
projection ::=	list_projection | tuple_operation
depth ::=	“*” | number
where_clause ::=	“WHERE” formula
formula ::=	“(“ formula “)” | predicate | quantified_formula | “(“ formula “AND” formula “)” | “(“ formula “OR” formula “)” | “NOT” formula
predicate ::=	value comp_operator value
comp_operator ::=	“=” | “IDENTICAL” | “DEEP_EQUAL” | “SHALLOW_EQUAL” | “IN” | “SUBSET_OF” | “MEMBER” | “SUBLIST_OF” | “>” | ”≥” | “<“ | “≤”
quantified_formula ::=	quantifier variable “IN” value “:” formula
quantifier ::=	“EXISTS” | “FORALL”
sort_statement ::=	“ORDER” variable “IN” value “BY” value mode
mode ::=	“ASC” | “DESC”
group_statement ::=	“GROUP” variable “IN” value “BY” value
query_definition ::=	“DEFINE QUERY” name [parameters] “AS” select_statement
parameters ::=	variable {“,” variable}

A.5 Transactions

ta_statement ::=	begin_transaction | commit | abort
begin_transaction ::=	“BEGIN” “TRANSACTION”
commit ::=	“COMMIT” “TRANSACTION”
abort ::=	“ABORT” “TRANSACTION”

A.6 Access Control

access_statement ::=	grant_statement | revoke_statement

grant_statement ::=	schema_grant | db_grant
revoke_statement ::=	schema_revoke | db_revoke
schema_grant ::=	“GRANT” right “ON” “SCHEMA” schema_name FOR user_name {“,” user_name }
db_grant ::=	“GRANT” right “ON” “DATABASE” db_name FOR user_name {“,” user_name }
schema_revoke ::=	“REVOKE” right “ON” “SCHEMA” schema_name FOR user_name {“,” user_name }
db_revoke ::=	“REVOKE” right “ON” “DATABASE” db_name FOR user_name {“,” user_name }
right ::=	READ | WRITE

A.7 Miscellaneous

schema_name ::=	identifier
name ::=	identifier
variable ::=	identifier
extension ::=	identifier
db_name ::=	identifier
vs_name ::=	identifier
ot_name ::=	identifier
at_name ::=	identifier
ut_name ::=	identifier
user_name ::=	identifier
real_number ::=	REAL_NUMBER
int_number ::=	NUMBER
identifier ::=	letter { letter_or_digit }
letter ::=	“a” | ... “z” | “A” | ... “Z” | “_”
letter_or_digit ::=	letter | digit
digit ::=	“0” | ... “9”
NUMBER ::=	digit { digit }
REAL_NUMBER ::=	[NUMBER] “.” NUMBER | NUMBER
string_constant ::=	“““ { char } “““
char ::=	letter | digit | “ “ | ...
char_constant ::=	““” char ““”