QAL: A Query Algebra of Complex Objects

Iztok Savnik
University of Ljubljana
Faculty of Computer and Information Sciences
Tržaška 25, 1000 Ljubljana, Slovenia
savnik@fri.uni-lj.si

Zahir Tari
Royal Melbourne Institute of Technology
Department of Computer Science
GPO Box 2476V, Melbourne 3001, Australia
zahirt@cs.rmit.edu.au

Tomaž Mohorič
University of Ljubljana
Faculty of Computer and Information Sciences
Tržaška 25, 1000 Ljubljana, Slovenia
Tomaz.Mohoric@fri.uni-lj.si

Abstract

The main motivation for the development of a query algebra of complex objects QAL is the study of the operations needed to query the structural aspects of object-oriented databases which are due to the advantages of the object-oriented database model over conventional relational and functional database models. The basic operations of the QAL query algebra evolved from relational algebras and the FQL family of functional query languages by refining their operations for the manipulation of objects. In order to support the features specific to object-oriented data models, QAL offers: (i) a set of operations which provide the means for querying conceptual schemata, and (ii) an operation which provides a simple and efficient way of querying nested components of complex objects. We show through a case-study consisting of a comprehensive set of examples how these operations can be employed to express a class of queries specific to object-oriented databases.

Index terms: Database algebras, object algebra, query languages, complex objects, conceptual schema.

1 Introduction

The characteristics of a database algebra have to reflect the properties of the algebraic structures for which algebraic operations are intended. From this perspective, the differences between an object algebra [55, 2, 47, 53, 32, 29, 33, 51] and an algebraic language defined for the manipulation of relations [16, 41, 1], for instance, can be regarded as the consequences of differences in the underlying data models. The features that have to be included in the object algebra are operations for handling the modelling constructs which are specific to object-oriented data models, namely: the structural constructs including object identity, types/classes, complex objects, and inheritance; and the behavioral constructs including methods, encapsulation, overriding, overloading and late binding [6, 10].

In this paper, we study the operations of object algebras needed for querying the structural aspects of objects which are due to the specific features of object-oriented data models compared to the flat relational [16], nested relational [41, 1] and functional [48, 11] data models. In particular, we examine the operations needed to express the following types of queries, which emerge from the need to deal with the modelling constructs specific to the object-oriented database models: (i) queries by which the conceptual schema of a database can be explored, (ii) queries which use the conceptual schema to retrieve portions of the extensional database, and finally (iii) queries that efficiently manipulate complex objects.

The results of our work are presented in the form of a query algebra of complex objects referred to as QAL. The QAL query algebra bears close resemblance to relational algebras, and the FQL family of functional query languages [11, 9, 34]. Most QAL operations are generalisations of the operations of the relational algebras for the manipulation of objects. Further, due to its close relation to the FQL family of query languages, QAL embodies the advantages of functional query languages: it has a simple functional semantics and it allows the construction of complex queries incrementally from simpler queries [11].

Let us now present the background and motivation for our work in more detail. First, the reasons for the use of the conceptual schema to query object-oriented databases, and the basic ideas behind the proposed operations for querying conceptual schemata, are given in the next sub-section. Second, the problems with existing operations for querying complex objects, and a brief overview of our approach to querying complex objects, are given in Section 1.2.

1.1 Using Conceptual Schema to Query Databases

In an object-oriented database, some aspects of the modelled environment can be represented and stored as part of the database's conceptual schema. One example of such information is the adherence of an object to its parent class, from which one can reason about its relationships with other classes forming the database schema, its static and behavioural properties, and its relationships with other database instances [43]. As a result, an object-oriented database system should provide facilities which allow inquiries and reasoning about the properties of the individual objects which derive from their relationships to the conceptual schema.

Furthermore, due to the rich semantic relationships between the modelled concepts, or because of the

large number of different concepts appearing in the modelled environment, the conceptual schema may be complex. Consequently, users often need facilities which allow inquiring about the relationships among the elements of the schema, in short, conceptual-schema querying facilities [38]. For example, a user may be interested in the relationships between the classes `course` and `assistant`, the properties of the class `student`, and so on. Using such facilities, the user can obtain the information needed to plan conventional queries, or simply obtain answers to inquiries about the conceptual aspects of a database.

The QAL query algebra includes a set of operations for inquiring about the relationships between individual objects and a conceptual schema, and the relationships among the elements of the conceptual schema. These operations are based on the constructs of the QAL data model which, like the data model of F-Logic [26], unifies the representation of the intensional and extensional portions of an object-oriented database. Hence, the operations are referred to as *model-based* operations. When used with other operations of the QAL query algebra, model-based operations can serve as a basis for: (i) querying the conceptual schema of an object-oriented database, and (ii) employing the relationships between the individual objects and the conceptual schema in conventional queries.

1.2 Querying Nested Components of Complex Objects

There are two main approaches to the manipulation of complex objects used by recent object algebras [54, 2, 47, 32, 33, 51]. The first approach evolved from the nested relational algebras [41, 1]. The operations used in this approach include variants of the NF^2 operations **nest** and **unnest** [41], together with operations for creating and flattening a set of sets [47]. This approach is used in some recent object algebras, including Query Algebra [47], Object Algebra [33] and Aqua [32].

The second approach to the manipulation of complex objects evolved from functional programming languages [39]. The operations for the manipulation of complex objects used in this approach are most often variants of the operations **apply_to_all** and the operation (tuple) **construction** [7]. The database algebras that have been influenced by the functional languages include: Complex Object Algebra [2], Object-Oriented Query Algebra [53] and Query Algebra [47].

The above two types of operations together are commonly referred to as *restructuring operations* [2]. A given object algebra either uses one of the approaches, or combines both of them. In general, the restructuring operations are appropriate for expressing queries that: extract selected components of complex objects; change the structure of complex objects at the upper level of the composition hierarchy; or, build entirely new complex objects from existing complex objects. We argue in the paper that restructuring operations are not suitable for expressing a class of queries that manipulate the nested components of complex objects and retain the original structure of the argument complex object, at least, up to the manipulated component. An example of such type of a query is filtering a set of objects which is nested in an argument complex object.

The major drawbacks of using the restructuring operations to query nested components of complex objects are as follows. The use of the restructuring operations which evolved from nested relational algebras requires

either additional properties to be satisfied by the manipulated complex objects, or additional indexing techniques to be used in order to prevent the loss of information during restructuring. Further, the queries which manipulate nested components are complex, and their evaluation is computationally expensive if they are expressed by restructuring operations. A more detailed description of the disadvantages of using restructuring operations to query nested components of complex objects is given in Section 6.2.

An operation called **apply_at** is proposed for the evaluation of queries on the nested components of complex objects [44]. The proposed operation is a generalisation of the well-known operation **apply_to_all** [7]. The operation **apply_at** overcomes problems that arise when using restructuring operations to query nested components of complex objects. We show that the proposed operation provides a simple and efficient way to query arbitrarily nested components of complex objects. In addition to the operation **apply_at**, which serves primarily for querying nested components of complex objects, the QAL query algebra includes a set of restructuring operations that are primarily intended for restructuring complex objects.

1.3 Paper Organisation

The paper is organised as follows. Section 2 presents the formalisation of the data model of QAL. Next, the operations of the QAL query algebra are presented in Section 3. Each of the operations is defined formally. Their functionality is illustrated by example. In Section 4, we present the use of QAL operations for expressing queries specific to object-oriented databases. In particular, we study their use for querying the intensional portion of a database, and for querying complex objects. Later, the experiences obtained with two prototype implementations of QAL are presented in Section 5. In Section 6 we present the algebras and query languages which are related to the QAL query algebra. Finally, in Section 7 we give concluding remarks and some directions for further work.

2 QAL Data Model

In this section, we present the formalisation of the QAL data model [42] which serves as a basis for the definition of the QAL query algebra. The data model of QAL is an object-oriented data model which provides, in addition to the basic constructs of object-oriented database models [6], a uniform view of the database by treating classes as abstract objects. In this respect, the data model is based on ideas introduced by the logic-based declarative language F-Logic [26].

The formalisation of the QAL data model serves as a framework for the definition of simple operations intended for the manipulation of the properties of individual and class objects. Each of these operations – which are introduced in the next section – is defined using constructs introduced by the formalisation of QAL data model. Further, we observe that the uniform treatment of the schema and data portions of the database provides a simple means of defining semantic relationships among the basic concepts of the data model, such as class and type interpretations, and the partial ordering of values.

In the following sub-section, we define identifiers and present some of their properties. Section 2.2 gives

the definitions of o-values and types and presents the basic structures they form. Finally, the structural aspects of objects are presented in Section 2.3. Only the basic constructs of QAL data model, which are essential for the development of the algebra, are presented in this paper. The complete formalisation of QAL data model, along with a formal description of the object behaviour, can be found in [42].

2.1 Identifiers

Let us first define some of the basic terms used in the paper. We assume the existence of a predefined infinite set of identifiers $\mathcal{O}$. An *identifier*[1] is a unique symbol which represents an abstract or concrete entity from the real world. Let us present some examples of identifiers.

The string "`Tom`", for instance, is an identifier which represents the name of a person. Next, the symbol "`4`" is an identifier which represents the integer four. Such identifiers are usually called *constants*. Furthermore, the identifier `tom`, for example, serves as a unique identification of a person whose name is "Tom". Next, the identifier `student` is a unique identification of the abstract representation of a student, which actually stands for a class of individual students. Finally, the term `string` will denote an identifier which represents the concept of a string as used in programming languages. The identifier `string` can also be seen as an abstract constant which stands for a class of constants.

As suggested by the above examples, the set $\mathcal{O}$ is further divided into the set of individual identifiers $\mathcal{O}_D$, representing concrete entities such as person `tom` or number `4`, and the set of class identifiers $\mathcal{O}_C$, representing abstract concepts such as identifiers `person` or `string` which stand for a group of individual entities. In some cases, we will refer to the individual and class identifiers simply as individuals and classes.

The most significant difference between class identifiers and individual identifiers lies in their interpretations. While the interpretation of an individual identifier is the individual itself, the interpretation of a class identifier is the set of individuals. The interpretation of class identifiers is defined as follows. Let $\texttt{id} \in \mathcal{O}_C$. The interpretation of `id`, denoted $I_c(\texttt{id})$, has the following properties [3]:

1. $I_c(\texttt{id}) \subseteq \mathcal{O}_D$, and
2. $\forall \texttt{id}_1 (\texttt{id}_1 \in \mathcal{O}_C \wedge \texttt{id}_1 \neq \texttt{id} \Longrightarrow I_c(\texttt{id}) \cap I_c(\texttt{id}_1) = \emptyset)$.

As can be seen from the above definition, we use the "common engineering intuition", as stated in [3], by treating individuals as members of the interpretations of single class identifiers. This design decision leads to disjunctive sets of individuals that represent the interpretations of class identifiers. Therefore, an individual identifier is an element of the interpretation of exactly one class identifier.

The class interpretation specifies the *membership* relationship among individual and class identifiers. Let $\texttt{id}_1 \in \mathcal{O}_D$ and $\texttt{id}_2 \in \mathcal{O}_C$. The identifier $\texttt{id}_1$ is a member of the identifier $\texttt{id}_2$ if $\texttt{id}_1 \in I_c(\texttt{id}_2)$. The membership relationship should not be confused with the *instantiation* relationship which is defined shortly.

A binary relation among class identifiers, denoted as $(\texttt{id}_1 \ \texttt{subclass} \ \texttt{id}_2)$ where $\texttt{id}_1, \texttt{id}_2 \in \mathcal{O}_C$, is used to represent the inheritance hierarchy of classes. We assume that this relation is given by the definition of

[1] The identifiers in QAL correspond to the so-called *logical id-s*, as introduced by Kifer et al. in [26].

the conceptual schema of an object-oriented database. Also, we assume the existence of a class called `object` which is the root of the hierarchy defined by the relationship `subclass`. Using the `subclass` relationship, we define a relationship $\preceq_i$. Let $\mathtt{id}_1, \mathtt{id}_2 \in \mathcal{O}$ then $\mathtt{id}_1 \preceq_i \mathtt{id}_2$ if one of the following holds:

1. $\mathtt{id}_1 = \mathtt{id}_2$,
2. $\mathtt{id}_1, \mathtt{id}_2 \in \mathcal{O}_C \Longrightarrow \exists \mathtt{id}_3 (\mathtt{id}_3 \in \mathcal{O}_C \wedge (\mathtt{id}_1 \ \mathtt{subclass} \ \mathtt{id}_3) \wedge \mathtt{id}_3 \preceq_i \mathtt{id}_2)$, or
3. $\mathtt{id}_1 \in \mathcal{O}_D \wedge \mathtt{id}_2 \in \mathcal{O}_C \Longrightarrow \exists \mathtt{id}_3 (\mathtt{id}_3 \in \mathcal{O}_C \wedge \mathtt{id}_1 \in I_c(\mathtt{id}_3) \wedge \mathtt{id}_3 \preceq_i \mathtt{id}_2)$.

The relationship $\preceq_i$ is called *more_specific* or, the opposite, *more_general* relationship. An example of a set of identifiers ordered by the relationship $\preceq_i$ is defined by the following terms: `student` $\preceq_i$ `person`, `employee` $\preceq_i$ `person`, `instructor` $\preceq_i$ `employee`, `ta` $\preceq_i$ `student`, `ta` $\preceq_i$ `instructor`, `jim` $\preceq_i$ `instructor`, `tom` $\preceq_i$ `student`, and `john` $\preceq_i$ `ta`. The relationship $\preceq_i$ forms a partial ordering of identifiers [42, 45].

The ordinary class interpretation maps class identifiers onto sets of individual identifiers called the members of the class. By taking into account the previously defined partial ordering of identifiers, another interpretation is introduced. The *inherited interpretation* [3, 55] of the class identifier `c` includes the members of the class `c` and the members of the subclasses of class `c`. Let $\mathtt{c} \in \mathcal{O}_C$. The inherited interpretation of `c`, denoted $I_c^*(\mathtt{c})$, is defined as: $I_c^*(\mathtt{c}) = \bigcup_{\mathtt{p} \in \mathcal{O}_C \wedge \mathtt{p} \preceq_i \mathtt{c}} I_c(\mathtt{p})$.

Using the above definition of the inherited interpretation, we define the *instantiation* relationship commonly used to represent the associations between individual objects and classes. Let $\mathtt{id}_1 \in \mathcal{O}_D$ and $\mathtt{id}_2 \in \mathcal{O}_C$. The identifier $\mathtt{id}_1$ is an *instance* of $\mathtt{id}_2$ if $\mathtt{id}_1 \in I_c^*(\mathtt{id}_2)$.

2.2 O-Values

So far we have presented identifiers and their structural properties. In this section, we extend the concept of identifier to the notion of *o-value* [3]. Let us first present the basic terms used in this section. We assume the existence of an infinite set of o-values $\mathcal{V}$. As shown by the following definition, the set of o-values $\mathcal{V}$ subsumes the previously defined set of identifiers $\mathcal{O}$. Just as the set of identifiers $\mathcal{O}$ is partitioned into individual and class identifiers, we divide the set of o-values into the set of ground o-values $\mathcal{V}_D$, and the set of o-values $\mathcal{V}_T$ that represent types. Further, we assume the existence of a set of attribute names $\mathcal{A}$. The o-value is one of the following:

1. $\mathtt{id} \in \mathcal{O}$,
2. $\{\mathtt{o}_1, \ldots, \mathtt{o}_n\}$, where $\mathtt{o}_i \in \mathcal{V}$, or
3. $[\mathtt{A}_1{:}\mathtt{o}_i, \ldots, \mathtt{A}_n{:}\mathtt{o}_n]$, where $\mathtt{o}_i \in \mathcal{V}$ and $\mathtt{A}_i \in \mathcal{A}$.

An example of an o-value is [name: "Jim",age:50,kids:{ana,tom},works_at:cs], representing the properties of a person. The string "Jim" and the integer number 50 are the individual identifiers which denote the name and the age of a person. The component {ana,tom} is the set of individual identifiers which represent kids, that is, the instances of the class identifier `person`. Finally, the component `cs` denotes an individual identifier which stands for an object representing the Computer Science Department.

The o-values which include only the individual identifiers are called *ground o-values*. When the o-values are composed solely of class identifiers, we refer to them as *structural types*, or simply *types*. The o-values

which include both individual identifiers and class identifiers are not permitted in the QAL data model. The properties of such "mixed" o-values as well as their integration in an object-oriented database model are presented in [45].

By analogy with our perception of class identifiers, types stand for the abstract representation of a set of o-values. Formally, a type is defined as follows. The o-value t is a type, that is, $\mathtt{t} \in \mathcal{V}_T$, if one of the following holds:

1. $\mathtt{t} \in \mathcal{O}_C$,
2. $\mathtt{t} = \{\mathtt{s}\}$, where $\mathtt{s} \in \mathcal{V}_T$,
3. $\mathtt{t} = [\mathtt{A}_1{:}\mathtt{t}_1, \ldots, \mathtt{A}_n{:}\mathtt{t}_n]$, where $\mathtt{t}_i \in \mathcal{V}_T$ and $\mathtt{A}_i \in \mathcal{A}$.

Let us present an example of a type. The tuple [name:string,age:int,kids:{person},works_at:organisation] represents the properties of person. The values of the attributes name and age are the class identifiers string and int which have the role of *primitive types*. Next, the value of the attribute works_at is a class identifier organisation which has the role of a *reference type* [55]; the instances of organisation are individual identifiers representing different organisations. Finally, the value of the attribute kids is {person} the instances of which are sets of individual identifiers representing persons.

2.2.1 Properties of o-values

Close integration of the concepts of the class identifier and the structural type provides a clear method for the definition of type interpretation, which can now be defined as a straightforward extension of the class interpretation. The interpretation of a type $\mathtt{t} \in \mathcal{V}_T$, denoted by $I(\mathtt{t})$, is [3, 30]:

1. $\mathtt{t} \in \mathcal{O}_C \Longrightarrow I(\mathtt{t}) = I_c^*(\mathtt{t})$,
2. $\mathtt{t} = \{\mathtt{s}\} \Longrightarrow I(\mathtt{t}) = \{\mathtt{o};\ \mathtt{o} \subseteq I(\mathtt{s})\}$,
3. $\mathtt{t} = [\mathtt{A}_1{:}\mathtt{t}_1, \ldots, \mathtt{A}_n{:}\mathtt{t}_n] \Longrightarrow I(\mathtt{t}) = \{[\mathtt{A}_1{:}\mathtt{v}_1, \ldots, \mathtt{A}_n{:}\mathtt{v}_n];\ \mathtt{v}_i \in I(\mathtt{t}_i)\}$.

This definition of the type interpretation specifies the *membership* relationship between ground o-values and types. Let $\mathtt{v} \in \mathcal{V}_D$ and $\mathtt{t} \in \mathcal{V}_T$. O-value v is a member of t if $\mathtt{v} \in I(t)$. Therefore, the *members* of the particular type t are the elements of the interpretation of type t.

The relationship $\preceq_i$ defined on identifiers is extended to relate o-values. The new relationship is denoted $\preceq_o$. As with the relationship $\preceq_i$, we call the relationship $\preceq_o$ the *more_specific* relationship. Intuitively, o-values that are more specific, or "below" in the ordering defined by the relationship $\preceq_o$, refine more general o-values that are "higher" in the set of o-values $\mathcal{V}$ with regard to the relationship $\preceq_o$. Just as the relationship $\preceq_i$ organises identifiers into a partially ordered set, the relationship $\preceq_o$ forms a partial ordering of o-values [45]. Formally, the relationship $\preceq_o$ is defined as follows. Let $\mathtt{v}_1, \mathtt{v}_2 \in \mathcal{V}$ be o-values. The o-value $\mathtt{v}_1$ is more specific then the o-value $\mathtt{v}_2$, denoted by $\mathtt{v}_1 \preceq_o \mathtt{v}_2$, if one of the following holds:

1. $\mathtt{v}_1, \mathtt{v}_2 \in \mathcal{O}_C \Longrightarrow \mathtt{v}_1 \preceq_i \mathtt{v}_2$,
2. $\mathtt{v}_1, \mathtt{v}_2 \in \mathcal{V}_T \wedge \mathtt{v}_1 = \{\mathtt{s}\} \wedge \mathtt{v}_2 = \{\mathtt{t}\} \Longrightarrow \mathtt{s} \preceq_o \mathtt{t}$,
3. $\mathtt{v}_1, \mathtt{v}_2 \in \mathcal{V}_T \wedge \mathtt{v}_1 = [\mathtt{A}_1{:}\mathtt{a}_1, \ldots, \mathtt{A}_n{:}\mathtt{a}_n] \wedge \mathtt{v}_2 = [\mathtt{B}_1{:}\mathtt{b}_1, \ldots, \mathtt{B}_k{:}\mathtt{b}_k] \Longrightarrow n \geq k \wedge \forall i(i \in [1..k] \Rightarrow \exists j(j \in [1..n] \wedge \mathtt{A}_j = \mathtt{B}_i \wedge \mathtt{a}_j \preceq_o \mathtt{b}_i))$, or
4. $\mathtt{v}_1 \in \mathcal{V}_D \wedge \mathtt{v}_2 \in \mathcal{V}_T \Longrightarrow \exists \mathtt{v}_3(\mathtt{v}_3 \in \mathcal{V}_T \wedge \mathtt{v}_1 \in I(\mathtt{v}_3) \wedge \mathtt{v}_3 \preceq_o \mathtt{v}_2)$.

The above definition of the o-value poset subsumes the notion of partial ordering of types as defined by Cardelli in [13], or Vandenberg in [55]. It can be obtained by restricting the set of all o-values $\mathcal{V}$ to types $\mathcal{V}_T$.

The type interpretation introduced by the above definition maps a type `T` onto a set of its members whose structure is strictly the same as the structure of a given type. We remove this constraint by defining the *inherited interpretation* of a type to be the union of the interpretation of a given type and the interpretations of all types which are more specific than a given type. Let $\mathtt{t} \in \mathcal{V}_T$. The inherited interpretation of `t`, denoted $I^*(\mathtt{t})$, is defined as: $I^*(\mathtt{t}) = \bigcup_{\mathtt{s} \in \mathcal{V}_T \wedge \mathtt{s} \preceq_o \mathtt{t}} I(\mathtt{s})$.

Further, the above definition of the inherited interpretation of types specifies the *instantiation* relationship between ground o-values and types. Formally, the instantiation relationship can be defined as follows. Let $\mathtt{v} \in \mathcal{V}_D$ and $\mathtt{t} \in \mathcal{V}_T$. The o-value `v` is an *instance* of type `t` if $\mathtt{v} \in I^*(\mathtt{t})$.

The definition of the relationship $\preceq_o$ gives a syntactical means for checking the relationship $\preceq_o$ between o-values. The correspondence between the syntactical definition of $\preceq_o$ and the inherited interpretation function I^* can be shown by the following equivalence. Let $\mathtt{t}_1, \mathtt{t}_2 \in \mathcal{V}_T$. It follows from the definitions of the relationship $\preceq_o$ and the inherited interpretation function I^* that:

$\mathtt{t}_1 \preceq_o \mathtt{t}_2 \iff I^*(\mathtt{t}_1) \subseteq I^*(\mathtt{t}_2)$.

Finally, we define the *extended interpretation* of structural types which will be used for the definition of the semantics of QAL variables when a variable is required to store a ground o-value or a type. The extended interpretation of a type `T`, denoted by $I^\diamond(\mathtt{T})$, includes all o-values that are more specific than type `T` with respect to the relationship $\preceq_o$; hence, the extended interpretation of a type includes ground o-values and types. Let $\mathtt{T} \in \mathcal{V}_T$. The extended interpretation of `T`, denoted as $I^\diamond(\mathtt{T})$, is defined as:

$I^\diamond(\mathtt{T}) = \{\mathtt{o};\ \mathtt{o} \in \mathcal{V} \wedge \mathtt{o} \preceq_o \mathtt{T}\}$.

2.3 Objects

The data model of QAL distinguishes between two aspects of objects. First, every object has an identifier, also called object identifier (abbr. oid), which distinguishes it from all other objects in a database. Second, every object has an o-value which describes its state. The two basic object aspects are linked by means of a *value assignment function* which is defined shortly. Formally, an object is defined as a pair `(id,v)`, where $\mathtt{id} \in \mathcal{O}$ and $\mathtt{v} \in \mathcal{V}$.

We distinguish between *primitive* and *structured* objects. The identity of the primitive object is the same as its value. An example of a primitive object is `(1,1)` which is a formal representation of the integer number one. The identity of "1" is the same as its value. Similarly, the object `(int,int)` stands for the formal representation of the integer type. As presented in Section 2.1, the term `int` represents an identifier.

The value of a structured object can be any of the previously defined structured o-values. An example of a tuple-structured object is `(jim,[name:"Jim",age:50,kids:{ana,tom},works_at:cs])`, where the term `jim` is an object identifier which uniquely identifies the object. This example presents an *individual object*; in

the following example, we give a description of an abstract entity represented by a *class object*. The object `(person,[name:string,age:int,kids:{person},works_at:address])` stands for an abstract representation of the person.

In the above examples, we made a distinction between individual and class objects. The *individual object* represents a single concrete entity from a modelling environment. Its value includes solely the individual identifiers. The *class object* represents an abstract entity which stands for the representation of an abstract concept. The value of a class object includes only class identifiers.

Let us now introduce the *value assignment function*, which realises the mapping between identifiers and o-values. Let `id` $\in \mathcal{O}$. The value assignment function $\nu : \mathcal{O} \rightarrow \mathcal{V}$ maps each identifier `id` to the corresponding value: ν`(id) = v`, where `v` $\in \mathcal{V}$ and `(id,v)` is an object.

3 Operations of the QAL Query Algebra

The QAL query algebra includes two types of operations: model-based operations and declarative operations. The *model-based operations* are used to access and manipulate the properties of individual and class objects which are represented by the constructs of the data model. All model-based operations are derived from the concepts used in the QAL data model formalisation. The *declarative operations* of QAL are used to express declarative queries on databases. The model-based operations are in this context used to assist the declarative operations in expressing database queries.

One of the initial aims of our work was to define a query algebra which can serve as the basis for a declarative language to be included in a C++-based database programming language (abbr. DBPL). Examples of such DBPLs include E DBPL [21], ODE [4] and ObjectStore [36]. C++-based DBPLs include the programming constructs for working with object identifiers and structures rather than for the manipulation of the abstract notion of object. For this reason, we define operations of the query algebra to deal with o-values.

The rest of this section is organised as follows. Firstly, the database environment that serves as the working example for illustrating the use of QAL operations is presented in the following sub-section. The model-based operations are defined in Section 3.2. Next, the declarative operations of QAL are presented in Section 3.3. Each of the QAL operations is defined in a formal manner. The functionality of the presented operations is illustrated by a set of examples.

3.1 Working Example

The examples in the following sub-sections are based on a conceptual schema which describes a simplified University environment. We use a simple language for the definition of classes, types and variables. Each class is defined by its name, a set of superclasses and a type. The name of the class is specified after the keyword **class**. After specifying the class name, a list of superclasses can follow the keyword **isa**. The type of the class can be stated after the keyword **type**. The static type of a class is specified by an o-value as

introduced in the previous section.

```
class department
type [ dept_name: string,
       head: employee,
       staff: {employee} ];

class course
type [ title: string,
       instructor: lecturer ];

class project
type [ proj_name: string,
       descr: string,
       attend: {employee},
       budget: integer ];

class person
type [ name: string,
       age: int,
       address: string,
       family: {person} ];

class student isa person
type [ courses: {course} ];

class employee isa person
type [ salary: int,
       manager: employee,
       projects: {project},
       dept: department ];

class lecturer isa employee;
class assistant isa lecturer;
class professor isa lecturer;
class student-assistant isa assistant,
                            student;
```

3.2 Model-Based Operations

The model-based operations are closely related to the concepts introduced by the data model formalisation presented in the previous section. They are intended for inquiring about: the associations among the individual objects, the relationships between the individual objects and class objects, and the relationships among class objects themselves.

The following model-based operations are defined: valuation operations, extension operations, comparison operations based on the o-value poset, closure operations based on the partial ordering relationship $\preceq_i$, operations for finding the nearest common more general or more specific objects of a set of objects, and equality operations. The use of operations is presented by examples of *queries* which are written as predicate calculus expressions.

3.2.1 Valuation Operator

Given an object identifier, the value of the object can be obtained using the value assignment function ν introduced in Section 2.3. The value assignment function is realised in the query algebra by the operation **val**. Let us look at an example to illustrate the use of the valuation function to obtain the properties of individual and class objects.

Example 3.1 *The first expression in this example calculates the value of the object identifier* $\mathtt{s}_1$*, which is a member of the class identifier* `student`*. The result of the second expression is the type of the class* `student`*. Note that the terms* $\mathtt{p}_1$, $\mathtt{c}_1$*, etc. represent object identifiers.*

s_1.**val** = [name:"Jim",age:23,address:"Ljubljana",family:{p_1,p_2},courses:{c_1,c_2}]
student.**val** = [name:string,age:int,address:string,family:{person},courses:{course}]

If the valuation function in a dot expression is followed by the attribute name, then the dot expression can be abbreviated using the operator "–>", as is common in procedural programming languages. For instance, the expression s_1–>age is the abbreviation for the expression s_1.**val**.age.

3.2.2 Extension Operations

In Section 2 we presented two types of class interpretations. Firstly, the ordinary interpretation I_c of a class identifier maps it onto the set of its members. This operation is denoted by **ext**. Secondly, the inherited interpretation I_c^* of a class identifier maps the class identifier onto the set of its instances. This interpretation is denoted by **exts**.

Example 3.2 *This example illustrates the use of extension operations. The query given below is based on the class hierarchy presented in Section 3.1. The result of the query is the set of identifiers referring to the objects representing persons who are younger than 22 years and who are not student assistants.*

{o; o ∈ person.**exts** ∧ o–>age < 22 ∧ o ∉ student_assistant.**ext**}

3.2.3 Comparison Operations Based on O-value Poset

The simplest and most natural way to express object properties which relate to the partial ordering of o-values is to use the partial ordering relationship $\preceq_o$ introduced in Section 2.2. The comparison operations related to the relationship $\preceq_o$ are $\prec_o, \succ_o, \succeq_o$. Their semantics is defined in the usual manner, for example, a $\prec_o$ b $\iff$ a $\preceq_o$ b ∧ a ≠ b. We call these operations *poset comparison operations.*

The poset comparison operations can be used to define a subset of the partially ordered set of o-values, such that the elements of the subset possess some properties that relate to the partial ordering of o-values. The expression x $\preceq_o$ lecturer, for example, defines a set identifiers composed of the class identifiers lecturer, assistant, professor, student_assistant, and including instances of these class identifiers as well.

Before illustrating the use of the poset comparison operations through a more complex example, the operation that maps an individual identifier onto its parent class identifier is introduced. This operation is denoted by the keyword **class_of** [10]. It is defined as x.**class_of** = c $\iff$ x ∈ I_c(c), where x is an individual identifier and I_c(c) denotes the ordinary interpretation of class identifier c. Note that any individual identifier belongs exclusively to the interpretation of one class identifier.

Example 3.3 *The query in this example selects instances of the class identifier* person *which are more specific than the class identifier* lecturer *and which are at the same time elements of either the class identifier* student_assistant *or some more general class identifier.*

{o; o ∈ person.**exts** ∧ o $\prec_o$ lecturer ∧ student_assistant $\preceq_o$ o.**class_of**}

In Example 3.3 the poset comparison operations are used to relate object identifiers. The following example shows the use of poset comparison operations to relate structured o-values.

Example 3.4 *The query presented in this example selects the values of instances of the class* `person`*, which have defined attributes* `family` *and* `manager`*. The value of the attribute* `family` *must be more specific than the type* {`employee`}*. Similarly, the value of the attribute* `manager` *is required to be more specific than the class identifier* `lecturer`*. The query is formulated as follows.*

$\{$`v; o` $\in$ `person.`**exts** $\wedge$ `v = o.`**val** $\wedge$ `v` $\prec_o$ `[family:{employee},manager:lecturer]`$\}$

3.2.4 Closure Operations

We introduce two transitive closure operations **subcl** and **supcl** which are intended for computing the set of all subclass or superclass identifiers of a given class identifier, respectively. Let `c` be a class identifier. The operations **subcl** and **supcl** are defined as follows.

`c.`**subcl** = $\{$`s; s` $\in \mathcal{O}_C \wedge$ `s` $\preceq_o$ `c`$\}$
`c.`**supcl** = $\{$`s; s` $\in \mathcal{O}_C \wedge$ `c` $\preceq_o$ `s`$\}$

Example 3.5 *The query presented in Example 3.3 can now be stated as follows.*

$\{$`o; o` $\in$ `person.`**exts** $\wedge$ `o.`**class_of** $\in$ `lecturer.`**subcl** $\wedge$ `o.`**class_of** $\in$ `student_assistant.`**supcl**$\}$

The transitive closure operations can express exactly the same relationships among class and individual identifiers as the previously presented poset comparison operations. The expression `x` $\preceq_o$ `y`, where `x` and `y` are class identifiers, can be translated to `x` $\in$ `y.`**subcl**. In a similar way, the expression `x` $\prec_o$ `y` can be translated to `x` $\in$ `y.`**subcl** $\wedge$ `x` $\neq$ `y`. While the comparison operations can serve merely for expressing relationships among object identifiers, the result of the closure operation is actually a set of identifiers, which can then be used later in the query.

3.2.5 The Nearest Common More General and More Specific Objects

Two operations are defined for computing the nearest common more general and more specific identifiers of a given set of identifiers with respect to the relationship $\preceq_i$. The operation which computes the nearest common more general identifiers of a set of identifiers with respect to the relationship $\preceq_i$ is called **lub-set**. Next, the operation called **glb-set** computes the set of nearest common more specific identifiers for a given set of identifiers.

Finding the nearest common more general (more specific) identifiers of a set of identifiers is related to the least upper bound (greatest lower bound) operation as defined on lattices. Since the object identifiers are ordered into the *partially ordered set*, there may be more than one least upper bound identifier for a given set of object identifiers. For instance, assume that the *isa* poset defined by the working example is extended by the class object `phd_student` for which the following relationships hold: `phd_student` $\preceq_i$ `student` and

phd_student $\preceq_i$ employee. In this case, the nearest common more general identifiers of the class identifiers phd_student and student_assistant are the class identifiers student and employee.

The operation **lub-set** is formally defined as follows. Given a poset $(\mathcal{O}; \preceq_i)$ and a set S such that $S \subseteq \mathcal{O}$, then the **lub-set** of S, denoted by S.**lub-set**, is the set L with the following properties:

1. $\forall o_1, \forall o_2 (o_1 \in S \wedge o_2 \in L \Longrightarrow o_1 \preceq_i o_2)$ and
2. $\forall o_1 (o_1 \in \mathcal{O} \wedge S \preceq_i o_1 \Longrightarrow \exists o_2 (o_2 \in L \wedge o_2 \preceq_i o_1))$, where $S \preceq_i o \Leftrightarrow \forall a (a \in S \Rightarrow a \preceq_i o)$.

The definition can be read as follows. Firstly, the object identifiers from the resulting set must be related to all identifiers from the argument set S by the relationship $\preceq_i$. Secondly, there does not exist an identifier which satisfies the first condition and which is more specific than an object identifier from the set of nearest common more general identifiers. Note also that the objects referenced by identifiers from the result L include all properties which are common to the objects referenced by identifiers from the set S.

The operation **glb-set** is similarly defined. Given the set of identifiers, the following two conditions must be true. First, the identifiers which are elements of the result set are more specific than every identifier from the argument set. Second, the result set includes only the most general identifiers which meet the first condition. The formal definition of **glb-set** is not presented here; it can be found in [45].

Example 3.6 *In this example, we present the use of the operation* **lub-set**. *The following query selects all members of the class identifiers which are the nearest common more general identifiers of the set* {student_assistant,professor}.

{o; c $\in$ {student_assistant,professor}.**lub-set** $\wedge$ o $\in$ c.**ext**}

The query first finds the set of class identifiers referring to the nearest more general class objects which include all properties which are common to the class objects referred by the identifiers from the set {student_assistant, professor}. *The members of these class identifiers are the result of the query.*

3.2.6 Equality

Two types of equality operation are defined. The first operation is the *identity equality* [10] denoted by the symbol "==". Two objects referenced by identifiers o_1 and o_2 are identical if the identifiers o_1 and o_2 denote the same value. The same equality operation is also used for comparing structured o-values. The o-values v_1 and v_2 are identical if they have identical components. The formal definition of identity equality is as follows. Let o_1 and o_2 denote two o-values. O-values o_1 and o_2 are identical, written o_1 == o_2, if: (i) they have the same type T; and, (ii) one of the following holds.

1. T $\in \mathcal{O}_C$ and, o_1 and o_2 denote the same value,
2. T = [A_1:T_1,...,A_n:T_n] and o_1->A_i == o_2->A_i, where $i \in [1..n]$, or
3. $T = \{S\}$ and there exists one-to-one mapping F from o_1.**val** onto o_2.**val**, such that for each pair (x,y) $\in F$, where x $\in$ o_1.**val** and y $\in$ o_2.**val**, the identity equality x == y holds.

The second equality operation is the *value equality*. It compares objects on the basis of their values. We distinguish between two types of value equality: *deep equality* [47] and *local equality*. Deep equality, denoted by "=", is defined as follows. The objects referenced by object identifiers o_1 and o_2 are deep equal, written o_1 = o_2, if their values are identical, that is, o_1.**val** == o_2.**val**.

The *local equality* operation, denoted "=/c", compares objects on the basis of the properties that pertain to the particular class identified by c. To be able to compare two objects using local equality "=/c", their identifiers have to be related to the class identifier c by the relationships $\preceq_i$. This, of course, does not imply that they have the same parent class. A formal definition of local equality is as follows. Let $Atr(c)$ be the function that returns the set of attributes defined for a class c. The objects referenced by identifiers o_1 and o_2 are locally equal for a class c, written o_1 =/c o_2, if the following holds:

1. $o_1 \preceq_i$ c and $o_2 \preceq_i$ c, and
2. for each A $\in Atr(c)$ the identity equality o_1->A == o_2->A holds.

Example 3.7 *Let us consider the following two objects:* (i1,[name:"Tone",age:40,address:"Titova 1", family:{p1,p4},salary:50000,manager:m1,projects:{},dept:e2]) *and* (i2,[name:"Vanja",age:24,address:"Jamova 5",family:{p6,p9},salary:50000,manager:m1,projects:{},dept:e2]). *Both objects are instances of the class* employee. *The objects are not value equal if all their properties are considered, yet they are value equal considering the local properties of* employee *represented by the attributes* salary, manager, projects *and* dept.

3.3 Declarative Operations

The QAL query algebra is a functional language. Every declarative operation is a function which is used to manipulate a set of o-values called the *argument* of the operation. Operations can be combined using the functional composition operator and/or by use of the higher-order functional operations to form *function expressions* [2] which are themselves *functions*. A function expression is a *query* when the argument and the free variables which appear in the expression are bound to o-values[2].

Formally, a QAL query is a function expression $o.f_1....f_n$, where o is an argument o-value which can be represented as a constant or a variable, the dot operator "." represents the functional composition, and f_i ($i \in [1..n]$) are QAL operations. The operations can have a set of *parameters* which are, depending on the operation, either o-values (queries), or function expressions. The result of a query is an o-value.

The set of declarative operations comprises three groups of operations. The first group includes the generalisation of the basic operations of the (flat) relational algebra for the manipulation of o-values. This group contains the operations **select**, **union**, **differ** and **intsc**. The second group of operations extends the functionality of the restructuring operations of NF^2 algebra [41, 17]. These are the QAL operations **unnest** and **group**. The last group of QAL operations is a set of higher-order functions that evolved from functional

[2] Abiteboul and Beeri [2] provide an extended discussion on the functions and the queries in algebras of complex values. The definitions of terms *function* and *query* are taken from their work.

query languages [11, 18]. This set includes the QAL operations **apply**, **tuple**, **close** and **apply_at**.

The rest of this section presents the examples illustrating the use of declarative operations for expressing queries. All examples are based on the conceptual schema presented in Section 3.1. The variables used in queries are defined in a similar manner to the definition of variables in the C (and C++) programming language. The following expression, for example, specifies the variable `pset`, the static type of which is {[name: string,age: int]}.

```
struct {[name:string,age:int]} pset;
```

3.3.1 Apply

The operation **apply**(f) is used to evaluate a parameter function expression `f` on the elements of the argument set. The operation is formally defined as follows.

$$s.\mathbf{apply}(f) = \{v;\ o \in s \wedge v = f(o)\}$$

Example 3.8 *An example of the use of the operation* **apply** *is given below. The presented query maps a set of object identifiers referring to instances of the class* `student` *onto a set of student names. The* identity *function* **id** *is used to identify the elements of the set* `students`*, which is an argument of the operation.*

```
struct {student} students;
struct {string} names;

names = students.apply( id->name );
```

As stated above, the parameter f of the operation **apply** can be any function expression. The following example illustrates the case where the parameter of the operation **apply** is another **apply** operation. This provides a tool for accessing nested sets (e.g., set of sets).

Example 3.9 *The query below first maps a set of identifiers, which are the elements of the interpretation of the class identifier* `student`*, to the set of sets which include identifiers that refer to courses studied by students. In the second step, the query replaces each identifier from the nested sets with the identifier which refers to the instructor of the particular course. Therefore, the result of the query is a set of sets, where each nested set includes references to the instructors of courses studied by a student.*

```
struct {{instructor}} insts_sets;

insts_sets = student.ext.
             apply( id->courses ).
             apply( id.apply( id->instructor ));
```

3.3.2 Selection

The operation **select**(p) is used to filter an argument set of o-values by using a parameter predicate `p`. The parameter predicate specifies the properties of selected o-values. The selection operation is defined as follows:

s.**select**(p) = {o; o ∈ s ∧ p(o)}

The type of the result is the same as the type of the input set of objects. The parameter predicate p is a boolean function. The predicate can consist of constants, typed variables and/or nested queries. The elements of the predicate can be related using the standard arithmetic relationships, the previously presented model-based operations, the set membership operation **in** and the boolean operations **and**, **or** and **not**. Let us illustrate the use of the operation **select** with an example.

Example 3.10 *The following query uses the operation* **select** *to compute the names of students who attend the course in graph theory.*

```
struct {string} stud_names;

stud_names = student.ext.
             select( "GraphTheory" in id->courses.
                                      apply( id->title )).
             apply( id->name );
```

3.3.3 Set Operations

The algebra includes the set operations: union, intersection and difference, which are denoted **union**, **intsc** and **differ**, respectively. The formal definition of the set operations is given below.

s.**union**(u) = {o; o ∈ u ∨ o ∈ s}
s.**intsc**(u) = {o; o ∈ s ∧ o ∈ u}
s.**differ**(u) = {o; o ∈ s ∧ o ∉ u}

If the types of the sets u and s are $\{T_1\}$ and $\{T_2\}$ respectively, then the type of the resulting set is $\{lub(T_1, T_2)\}$, where lub is the least upper bound operation.

Example 3.11 *This example illustrates the use of the operation* **union**. *The query given below computes the union of: the set of instructors who work in the department named* E4, *and the set of students who have at least one instructor from this department.*

```
struct {person} e4_people;

e4_people = instructor.ext.
            select( id->dept->dept_name == "E4" ).
            union( student.ext.
                   select( "E4" in id->courses.
                                   apply( id->instructor->dept->dept_name ));
```

3.3.4 Close

The operation **close**(f) is defined in order to provide the end-user with a simple tool for the manipulation of recursive data structures [52]. Its semantics is defined as follows. Given an argument set s that contains

instances of type T, the closure of this set is computed using the parameter function expression f. The result of the evaluation of the function f must be instances of type T. The operation is formally defined as follows.

$$\mathtt{s}.\mathbf{close}(\mathtt{f}) = \begin{cases} \mathtt{s} & \text{if } \mathtt{s}_1 = \{\}, \text{or} \\ (\mathtt{s} \cup \mathtt{s}_1).\mathbf{close}(\mathtt{f}) & \text{otherwise,} \end{cases}$$

where $\mathtt{s}_1 = \{\mathtt{o};\ \exists \mathtt{p}(\mathtt{p} \in \mathtt{s} \wedge \mathtt{p}.\mathbf{class\_of} = \mathtt{c} \wedge \mathtt{o} \in \mathtt{c}.\mathbf{exts} \wedge \mathtt{o}\ \mathbf{op}\ \mathtt{p}.\mathtt{f} \wedge \mathtt{o} \notin \mathtt{s})\}$.

Depending on the cardinality of the result of the function expression f, **op** is either $=$ or $\in$ operation. The type of the result is the same as the type of the input argument.

Example 3.12 *The following query selects from the interpretation of the class* employee *the set of employees who earn less than* 10000. *The resulting set is extended by computing all managers of the previously selected employees.*

```
struct {employee} empls;

empls = employee.ext.
        select( id->salary < 10000 ).
        close( id->manager );
```

3.3.5 Tuple

The operation $\mathbf{tuple}(\mathtt{A}_1{:}\mathtt{f}_1, \ldots, \mathtt{A}_n{:}\mathtt{f}_n)$ is a generalisation of the relational operation **project**. Given a set of o-values as an argument of the operation, a tuple is generated for each o-value from the set. Each component of the newly created tuple is specified by the corresponding **tuple** parameter consisting of the attribute name $\mathtt{A}_i$ and the function expression $\mathtt{f}_i$ $(1 \leq i \leq n)$. The function expression $\mathtt{f}_i$ is evaluated for each element of the argument set producing an o-value, which serves as the value of the attribute $\mathtt{A}_i$ in the resulting tuples. The operation **tuple** is formally presented below.

$$\mathtt{s}.\mathbf{tuple}(\mathtt{A}_1{:}\mathtt{f}_1, \ldots, \mathtt{A}_n{:}\mathtt{f}_n) = \{[\mathtt{A}_1{:}\mathtt{f}_1(\mathtt{o}), \ldots, \mathtt{A}_n{:}\mathtt{f}_n(\mathtt{o})];\ \mathtt{o} \in \mathtt{s}\}$$

The type of the resulting o-value is $\{[\mathtt{A}_1{:}T(\mathtt{f}_1), \ldots, \mathtt{A}_n{:}T(\mathtt{f}_n)]\}$, where $T(\mathtt{f})$ denotes the type of the result of a function expression f.

Example 3.13 *The following query constructs a tuple for each member of the class* student. *The tuple consists of the student name and the set of tuples that describe the courses studied by the student. Nested tuples include titles of courses and names of instructors.*

```
struct {[ sname: string;
          courses: {[ title: string,
                      iname: string ]}]} students;

students = student.ext.
           tuple( sname: id->name,
                  courses: id->courses.
                           tuple( title: id->title,
                                  iname: id->instructor->name ));
```

3.3.6 Group

The operation **group**(A:f,B:g) is used to group the elements of the argument set. The function expressions f and g are evaluated on every element of the input set. The results of the evaluation of g are grouped in accordance with the result of the evaluation of f.

The result of the operation **group** is an o-value whose structure is a two column table. The values of the first attribute, denoted by the label A, are all distinct values of the function f applied to the input set of o-values. The values of the second attribute, labelled B, are the groups of results of function g application for o-values which share the common value of the function f. The operation **group** is formally defined as follows.

$$\texttt{s.}\mathbf{group}\texttt{(A:f,B:g)} = \{\texttt{[A:}v_A\texttt{,B:}v_B\texttt{]};\ \exists \texttt{r}(\texttt{r} \in \texttt{s} \wedge v_A\texttt{=r.f} \wedge \forall \texttt{o}((\texttt{o} \in \texttt{s} \wedge v_A\texttt{=o.f}) \Rightarrow \texttt{o.g} \in v_B) \wedge \nexists \texttt{p}(\texttt{p} \in \texttt{s} \wedge \texttt{p.g} \in v_B \wedge \texttt{p.f} \neq v_A))\}$$

The type of the resulting object is in the form of [A:T(f),B:{T(g)}], where T(f) denotes the type of the result of the function expression f.

Example 3.14 *This example presents the use of the operation* **group** *to group the members of the class* employee *by their departments.*

```
struct {[ dept: department,
          emps: {employee} ]} depts_groups;

depts_groups = employee.ext.
               group( dept: id->dept, emps: id );
```

3.3.7 Unnest

There are three different types of restructuring operations that can be used to unnest structured objects in an argument object. These operations are: (i) producing a flat structure from a set of sets, (ii) unnesting a tuple component which is a set of o-values, and (iii) unnesting a tuple component which is a tuple. These three operations are defined in many algebras, where they are usually denoted **flatten**, **unnest** [47] and **tup_collapse** [2], respectively.

In the QAL algebra, the above operations are defined in the frame of a single operation **unnest**. This operation has different behaviour depending on the type of object to which the operation is applied. The advantage of such a definition is that it provides a uniform view of the previously presented operations. Let us present the formal definition of the QAL operation **unnest**. The term s in the following definition denotes a set of o-values, and the function T(o) returns the type of the o-value o.

(1) T(s) = {{S}} $\Longrightarrow$ s.**unnest**= {o; $\exists s_i$($s_i \in$ s $\wedge$ o $\in$ s_i)}

(2) T(s) = {[A_1:T_1,...,A_i:{T_i},...,A_n:T_n]} $\Longrightarrow$
s.**unnest**(A_i) = {t; $\exists$p(p $\in$ s $\wedge$ t.A_i $\in$ p.A_i $\wedge$ $\forall j(j \neq i \Rightarrow$ p.A_j = t.A_j))}

(3) T(s) = {[A_1:T_1,...,A_i:[B_1:S_1,...,B_k:S_k],...,A_n:T_n]} $\Longrightarrow$
s.**unnest**(A_i) = {t; $\exists$p(p $\in$ s $\wedge$ $\forall l(l \in [1..k] \Rightarrow$ t.B_l = p.A_i.B_l) $\wedge$
$\forall j(j \neq i \Rightarrow$ t.A_j = p.A_j))},

Example 3.15 *In this example, we illustrate the use of the operation* **unnest**. *The first query computes all courses studied by the students. The second query computes the set of all pairs composed of object identifiers referring to the employees and their departments.*

```
struct {course} courses;
struct {[ dept: department,
          emp: employee ]} depts_empls;

courses = student.ext.
          apply( id->courses ).
          unnest;

depts_empls = department.ext.
              tuple( dept: id, emp: id->staff ).
              unnest( emp );
```

3.3.8 Apply_at

To be able to apply a QAL function expression to any nested component of an o-value, the functionality of the operation **apply** is extended by a new parameter which serves as a component selector. The operation **apply_at**(p,f) identifies the desired components by applying the *aggregation path* p to the o-values which are the elements of the argument set. The function f is evaluated on the identified components.

The aggregation path is specified by a sequence of attributes separated by dot operators. The evaluation of the aggregation path serves solely for the identification of the component, and does not restructure the argument o-value. The only manipulation activity that results from the evaluation of the operation **apply_at** is the result of the argument function expression f evaluation. Let us present an example to illustrate the use of the operation **apply_at**.

Example 3.16 *The following query filters the nested components of o-values which describe departments. The nested components are identified by the attribute* staff. *The filtered sets of object identifiers include references to employees older than 45.*

```
struct {[ dept_name: string,
          head: employee,
          staff: {employee} ]} depts;

depts = department.ext.
        apply( id.val ).
        apply_at( staff,
                  select( id->age > 45 ));
```

The operation **apply_at** can be formally described by a set of rules. Let s be a set of o-values of type $\{T_1\}$, o be an o-value of type T_2, p be an aggregation path, f be a function expression, and *Atr* be a function which maps a tuple structured type T to the set of attributes defined by T. The operation **apply_at** is defined as follows.

(1) s.**apply_at**(A.p,f) = {o'; $\exists$o(o $\in$ s $\wedge$ o'.A = o.A.**apply_at**(p,f) $\wedge$
$\forall$B((B $\in$ *Atr*(T_1) $\wedge$ A $\neq$ B) $\Rightarrow$ o'.B = o.B))}
(2) o.**apply_at**(A.p,f) = o', such that o'.A = o.A.**apply_at**(p,f) $\wedge$
$\forall$B((B $\in$ *Atr*(T_2) $\wedge$ A $\neq$ B) $\Rightarrow$ o'.B = o.B)
(3) s.**apply_at**(null,f) = s.**apply**(f)
(4) o.**apply_at**(null,f) = o.f

The above definition should be read as follows. If the input argument of the operation is a set s and the path p is not empty, then the first rule is applied. Next, if the argument of the operation is a set s and the path p is empty, then the ordinary operation **apply** is used to evaluate function expression f on the elements of s by rule 3. If the argument is a single o-value, rule 2 is used to evaluate the function expression f on the selected component. Finally, rule 4 is used to evaluate f on the single argument o-value in cases where the path p is empty. Let us now illustrate the use of operation **apply_at** by another example.

Example 3.17 *In this example we present a more complex query which, given a data structure describing departments, filters the component identified by the path expression* staff.family *by selecting only those object identifiers more specific than the class* employee*, that is, only those who are the members of the class* employee *or some more specific class. Note that the type of the argument o-value* depts *is not changed.*

```
struct {[ dept_name: string,
          head: employee,
          staff: {[ emp_name: string,
                    address: string,
                    projects: {project},
                    family: {person} ]} ]} depts;

depts = depts.apply_at( staff.family,
                        select( id ≺o employee ));
```

The prototype implementations of **apply_at** show that it can be realised as an iterator [23] which recursively accesses the target nested components of complex objects, passes them to the query tree which realises the parameter function expression of **apply_at**, and, finally, assembles the results of the parameter function expression. In other words, each complex objects passed to the iterator of **apply_at** is individually decomposed and later assembled after the parameter function expression is evaluated on its nested components. It is important to note that the operation **apply_at** does not require restructuring of the argument complex objects in the way they are restructured when using the restructuring operations (presented in Section 1.2) for accessing the nested components of complex objects[3].

By the previous definition of the aggregation path, the operation **apply_at** can access any component of complex values constructed by the set and the tuple constructors, with the exception of the elements of multilevel nested sets, for example the elements nested in {{1,2},{3}}. To be able to access the elements of such components using the operation **apply_at**, an operation "**in**" is defined. The operation can be included in the aggregation path as any other ordinary attribute. Formally, its semantics can be defined by adding the following rule to the previous definition of operation **apply_at**.

[3]The use of restructuring operations for querying nested components of complex objects is presented by Example 4.12.

```
(5) s.apply_at(in.p,f) = {o'; ∃o(o ∈ s ∧ o' = o.apply_at(p,f) )}
```

As can be seen from the above rule, the function of attribute **in** is analogous to the function of operation **apply**. The argument of the operation **in** is a set of objects. The attribute that follows the operation **in** (or, the argument query, where **in** is the last component of the aggregation path) is applied to each element of the **in**'s argument set. The use of the attribute **in** is demonstrated by the following example.

Example 3.18 *Suppose that the following variable, named* `depts_groups`*, represents the result of some grouping of employees from each department; the value of the attribute* `groups` *is a set of sets containing tuples describing employees. The query presented below filters nested sets which are the values of the attribute* `projects` *by selecting only those projects with a budget higher than 6000.*

```
struct {[ dept_name: string,
          groups: {{ [emp_name: string,
                      projects: {project}] }} ]} depts_groups;

depts_groups = dept_groups.apply_at( groups.in.projects,
                                     id.select( id->budget > 6000 ));
```

4 Queries Specific to Object-Oriented Data Models

So far, we have presented the operations of the QAL query algebra. In this section, we focus on the use of operations to express queries which are specific to object-oriented databases. Two types of queries are presented: (i) queries that manipulate conceptual schemata, and (ii) queries that manipulate complex objects. Queries that manipulate conceptual schemata are presented in Section 4.1. The queries are classified according to the kind of objects which they manipulate, and on the type of the operation they perform. We show through a comprehensive set of examples that the model-based operations of QAL can serve as a tool for expressing queries with which one can inquire about the properties of the conceptual schemata, and about the relations between the individual objects and the conceptual schemata. Next, in Section 4.2 the queries used for the manipulation of complex objects are classified depending on the type of the operation they perform. The presented classification is based on the classification of the operations on complex objects presented by Kim et al. in [27]. We demonstrate that QAL operations can express the main types of queries on complex objects.

4.1 Using Conceptual Schemata for Querying Databases

Let us first define two general types of queries which will later serve for the classification of queries which use the conceptual schemata. Firstly, queries in which the structure and the meaning of data stored in a database is explored, and the answers to which consist of the elements of the conceptual schema, are called the *intensional queries* [38]. Secondly, the conventional queries by which the elements of the extensional portion of a database are manipulated, and answers to which consist solely of the extensional portion of the database are referred to as *extensional queries.*

In this sub-section we study two types of queries that use the conceptual schemata. Firstly, intensional queries which use the model-based operations for inquiring and reasoning about the properties of the conceptual schemata are presented. The second type of queries presented in this sub-section are the extensional queries, where the model-based operations are used for querying the relationships between the individual objects and the elements of the database conceptual schemata.

4.1.1 Intensional Queries

Intensional queries allow for browsing the schema using the navigational facilities of the query language, posing questions about the properties of the schema elements, and inquiring about the associations among the elements of the conceptual schema. The first two examples in this section present the use of intensional queries for querying the inheritance hierarchy of classes. The third example presents an inquiry about the structural properties of classes. The last example presents a query that searches for the common properties of a set of objects.

The semantics of variables used in the examples and the semantics of the intermediate results of the QAL queries is defined using the extended interpretation of types $I^\diamond$ which is presented in Section 2.2.1. Given a type T the interpretation $I^\diamond(T)$ includes all o-values which are more specific than T with respect to the relationship $\preceq_o$. Therefore, the interpretation of a type T can include ground o-values as well as types.

Example 4.1 *The following query selects a set of class identifiers which have the following properties. Firstly, the selected class identifiers are the subclasses of the class identifier* `person`. *Secondly, the selected identifiers are more specific or equal to the class* `employee` *and more general than or equal to the class* `student-assistant`. *In general, such queries use the poset comparison operations and/or the operations* **subcl** *and* **supcl** *to select the subset of class identifiers representing the conceptual schema.*

```
struct {person} classes;

classes = person.subcl.
          select( id ⪯o employee and id ⪰o student-assistant );
```

Let us at this point give a detailed explanation of the type of the result which is {`person`}. *Firstly, the operation* **subcl** *takes an argument class identifier* `person` *and returns a set of class identifiers which are more specific than (or subclasses of)* `person`. *The type of this set is* {`person`}. *Namely, the natural interpretation* $I^\diamond$({`person`}), *which is used for the definition of the semantics of types, includes all subsets of the set of identifiers which are more specific than the class identifier* `person`. *Secondly, in accordance with the definition of the operation* **select**, *the type of result of the operation* **select** *is the same as the type of its argument.*

Example 4.2 *The query presented below retrieves all classes at the top level of the class hierarchy*[4], *that is, the only superclass they have is* `object`. *Note that the role of the operator* **supcl** *in such queries is similar to the role of the extension operator.*

[4] This query was originally presented in [22].

```
struct {object} top_classes;

top_classes = object.subcl.
              select( id.supcl == {object} );
```

Example 4.3 *The following query constructs a tuple for every superclass of the class* `student-assistant`*. Each of the resulting tuples comprises the class object identifier and the value of the class object. Therefore, the static structure of the superclasses of the class* `student-assistant` *is returned by the query.*

```
struct {[ pclass: person,
          ptype: [ name: string,
                   age: int,
                   address: string,
                   family: {person} ] ]} stat_struct;

stat_struct = student-assistant.supcl.
              tuple( pclass: id,
                     ptype: id.val );
```

Note that because of the use of the extended interpretation $I^\diamond$ for the definition of the semantics of types, the result of the above query can be any o-value which is more specific than the type of the variable `stat_struct`*. For instance, the resulting set includes the tuple* `[pclass:student,ptype:[name:string,age:int,address:string,family:{person},courses:{course}]]` *which is constructed for the class* `student`*.*

Example 4.4 *The query presented below finds all common superclasses of the classes from the set* {`student-assistant,professor`}*. Firstly, the operation* **lub-set** *computes the set of the closest common superclasses of the argument set of class identifiers. Let us denote this set by* `s`*. Secondly, the set* `s` *is extended by the class identifiers which are more general than the class identifiers from* `s`*. This is achieved by applying the operation* **supcl** *to each class identifier from the set* `s` *and then collapsing the set of sets using the operation* **unnest**. *Note that the operation* **lub-set** *has to be used in order to assure that the resulting set includes only the class identifiers which are more general than or equal to both* `student-assistant` *and* `professor`*.*

```
struct {person} common_supclasses;

common_supclasses = {student-assistant,professor}.lub-set.
                    apply( id.supcl ).
                    unnest;
```

4.1.2 Relating Extensional and Intensional Portions of a Database

The main characteristic of the extensional queries presented by the following examples is that they relate individual objects to the information represented by the conceptual schema. The first example presents a query that relates individual objects to the inheritance hierarchy of classes by using the poset comparison operations and the operation **class_of**. The query in the second example presents the use of poset comparison operations to relate ground o-values with structural types. Next, the query in Example 4.7 presents the use

of model-based operations to generalise the results of extensional queries. Finally, the last example in this section demonstrates the use of local equality to compare individual objects on the basis of the properties that pertain to their relationships with particular classes.

The following two queries are equivalent to the queries from Examples 3.3 and 3.4, which are now restated to illustrate the use of poset comparison operations in the context of QAL declarative operations.

Example 4.5 *The query given below uses poset comparison operations to relate object identifiers. It retrieves the members of the class identifiers which are more specific than or equal to the class identifier* `lecturer` *and, at the same time, more general than or equal to the class identifier* `student_assistant`.

```
struct {person} lecturers;

lecturers = person.exts.
            select( id ≺o lecturer and
                    id.class_of ≻o student_assistant );
```

The parent class identifier of each instance of the class identifier `person` *has to be computed (using the operation* **class_of***) in order to select the members of class identifiers that are more general than or equal to* `student-assistant`. *However, the parent class identifier does not need to be computed in order to determine if an instance of* `person` *is an element of a class identifier which is more specific than or equal to* `lecturer`. *Namely, in accordance with the definition of the relationship* $\preceq_o$ *in Section 2.2.1, the relationship* **id** $\prec_o$ `lecturer` *implies the relationship* **id.class_of** $\preceq_o$ `lecturer` *and vice versa.*

Example 4.6 *The following query demonstrates the use of the poset comparison operations for relating tuple structured o-values. It returns instances of the type of the class* `person` *which are more specific than the o-value* `[family:{employee},manager:instructor]`.

```
struct {[ name: string,
          age: int,
          address: string,
          family: {person} ]} people;

people = person.exts.
         apply( id.val ).
         select( id ≺o [ family: {employee},
                         manager: instructor ]);
```

The result of the above query is the set of o-values of type {`[name:string,age:int,address:string,family:{person}]`}. *The use of the extended interpretation of types* $I^\diamond$ *allows the instance of a given type* T *to be any o-value more specific than* T. *Therefore, the instances can include – in addition to the attributes defined by* T *– some further attributes.*

Example 4.7 *The query given below answers the following question: Which are the parent classes of employees who work in CSD and who are younger than 25? The query first filters the set of individual identifiers*

denoting employees and, then applies the operation **class_of** *to each identifier from the filtered set. The resulting set of class identifiers acts as the generalisation of the filtered set of individual identifiers. The result could be further generalised using the operation* **lub-set**, *which computes the closest common more general class identifiers of a set of identifiers.*

```
struct {employee} empls_classes;

empls_classes = employee.exts.
                select( id->dept == csd and
                        id->age < 25 ).
                apply( id.class_of );
```

Example 4.8 *The comparison of individual objects based on properties inherited from particular classes is illustrated by the following query. The query uses local equality as defined in Section 3.2 to select student assistants who have the properties which relate to their role of being employees equal to the properties of an employee referenced by the variable* `peter`.

```
struct {student_assistant} assistants;
struct employee peter;

assistants = student_assistant.exts.
             select( id.val =/employee peter.val );
```

4.2 Querying Complex Objects

In this section, we present the use of QAL operations for the manipulation of complex objects. The presentation is based on the classification of operations on complex objects defined by Kim et al. in [27]. The following types of queries which manipulate complex objects are presented in Sections 4.2.1 – 4.2.4. Firstly, the set of complex objects can be filtered by specifying a selection condition on the values of simple and complex attributes of argument complex objects. The values of complex attributes are accessed using path expressions and/or nested queries. The queries of second type retrieve the selected attributes from the flat and/or nested levels of complex objects from the argument set. Next, we present queries which manipulate nested components of complex objects. Finally, the use of QAL operations for restructuring complex objects is demonstrated. All examples presented in this section use the complex object `depts` representing departments and their employees introduced by Example 3.17.

4.2.1 Retrieving Complete Complex Objects

The operation which filters a set of complex objects and retrieves the entire complex object is sometimes called *object selection* [33]. In the simple case – when the selection condition is based on the values of the single-valued attributes of complex objects – retrieving the entire complex objects can be expressed using the operation **select** and ordinary path expressions [10]. Let us give an example of such a query.

Example 4.9 *As in other examples presented in this section, we use the variable* `depts` *representing departments and their employees from Example 3.17. The following query selects all departments whose head is a member of the class* `professor` *or some more specific class.*

```
depts = depts.select( id.head.class_of ≼o professor );
```

A more complex case is retrieving complex objects by expressing the predicates on the nested components of complex objects. The predicates are specified by the use of nested queries. A query of this type is presented by the following example.

Example 4.10 *The query given below retrieves departments which are involved in the project identified by object identifier* `pr4`.

```
depts = depts.select( pr4 in id.staff.
                          apply( id.projects ).
                          unnest );
```

4.2.2 Projecting Components of Complex Objects

The second type of queries which retrieve complex objects is similar to the nested-relational operation **project** as defined in [1, 17, 33]. Queries of this type project the selected attributes from the flat level and/or from the nested levels of the argument complex objects. They can be realised using the QAL operations **tuple** and **apply**. The operation **tuple** is used for projecting the selected attributes. However, it can realise much more complex operations by using arbitrary functional expressions to specify the components of the constructed tuples. The operation **apply** is in this context used to access the elements of the arbitrarily nested sets. An example of a query that projects the selected components of the complex object is given below.

Example 4.11 *Given a set of o-values describing departments, the presented query projects the values of the attribute* `head` *and the values of attributes that describe personal information about employees.*

```
struct {[ head: employee,
          staff: {[ emp_name: string,
                    address: string,
                    family: { person } ]}]} depts_personal;

depts_personal = depts.tuple( head: id.head,
                              staff: id.staff.
                                     tuple( emp_name: id.emp_name,
                                            address: id.address,
                                            family: id.family ));
```

4.2.3 Querying Nested Components of Complex Objects

In this section, we present queries that manipulate nested components of the argument complex objects and retrieve the complete complex objects that have changed target components. Such queries can be

manipulated using the operations of the two approaches presented in Section 1.2. The first approach, which is most commonly employed in object algebras, is the use of restructuring operations which evolved from NF^2 algebras [41, 1]. The second approach to querying complex objects is the use of higher-order functional operations which evolved from functional languages [7, 11].

The queries which manipulate nested components of complex objects can be in the QAL query algebra expressed using the operation **apply_at**. Let us present an example of a query which manipulates the nested components of the complex object and express it by the operations of different approaches.

Example 4.12 *The first query demonstrates the use of the operation* **apply_at** *for querying nested components of complex objects. The complex objects from the set, which is the value of the variable* `depts`*, are manipulated by filtering the nested sets identified by the attribute* `family`*. The original set is replaced by the set of family members who are younger than 30.*

```
depts = depts.apply_at( staff.family,
                        id.select( id->age < 30 ));
```

The simplest way to express the above query using the previously stated first type of restructuring operations is to use the ordinary NF^2 restructuring operations **nest** *and* **unnest** *as they are defined, for instance, in [41]. Suppose that the relation* `r` *is defined by the set of attributes* $\{A_1, \ldots, A_n\}$*. In brief, the operation* `r.nest(S,`$B_1, \ldots, B_k$`)`*, where* $B_i \in \{A_1, \ldots, A_n\}$*, creates a nested relation named S which includes the values of the attributes* $B_1, \ldots, B_k$*. Next, the operation* `r.unnest(A)` *unnests the nested relation represented by the attribute* $A \in \{A_1, \ldots, A_n\}$*. The above query can now be expressed as follows.*

```
depts = depts.unnest( staff ).
              unnest( family ).
              select( id->age < 30 ).
              nest( family, family ).
              nest( staff,emp_name,address,projects,family );
```

In order to express this query with the operations used by the previously stated second approach to querying complex objects, we can use the QAL higher-order functional operations **tuple** *and* **apply**. *The query that is equivalent to the above query can be expressed as follows.*

```
depts = depts.tuple( dept_name: id.dept_name,
                     head: id.head,
                     staff: id.staff.
                            tuple( emp_name: id.emp_name,
                                   address: id.address,
                                   projects: id.projects,
                                   family: id.family.
                                           select( id->age < 30 )));
```

4.2.4 Restructuring Complex Objects

In this section, we present the use of QAL operations for restructuring complex objects. The first example demonstrates the use of QAL restructuring operations to realise the typical restructuring operations on NF^2

relations. Next, we present a type of query which uses restructuring operations together with the operation **apply_at** to realise restructuring of the nested components of complex objects.

Example 4.13 *As in other examples in this section, we use the variable* `depts` *defined in Example 3.17. The query first unnests the attributes* `staff` *and* `projects`*. The operation* **unnest** *has to be used twice in order to flatten the nested sets of tuples which are the values of the attribute* `staff` *(see the definition of the operation* **unnest** *in Section 3.3.7). The first operation* **unnest** *flattens the nested sets of tuples, and the second* **unnest** *collapses the nested tuples. Next, the query groups the names of employees and departments by the projects in which they participate.*

```
struct {[ project: project,
          group: {[ emp_name: string,
                    dept_name: string ]}]} proj_groups;

proj_groups = depts.unnest( staff ).
                    unnest( staff ).
                    unnest( projects ).
                    group( project: id.projects,
                           group: id.tuple( emp_name: id.emp_name,
                                            dept_name: id.dept_name ));
```

If the parameter function expression of the operation **apply_at** includes a restructuring operation, then the complex object is restructured at the component determined by the aggregation path. In this way, a restructuring operation can be applied to any nested component of a complex object. The use of the restructuring operation **tuple** as the parameter of the operation **apply_at** is illustrated by the following example.

Example 4.14 *Given a set of complex objects describing departments, the following query first identifies the components that include a set of identifiers referring to projects and then replaces every occurrence of an oid with a tuple that contains the name and the budget of each project.*

```
struct {[ dept_name: string,
          head: employee,
          staff: {[ emp_name: string,
                    address: string,
                    projects: {[ proj_name: string,
                                 budget: int ]},
                    family: {person} ]} ]} depts_projects;

depts_projects = depts.apply_at( staff.projects,
                                 id->tuple( proj_name: id->proj_name,
                                            budget: id->budget ));
```

5 Prototype Implementations

Two prototypes of the QAL query algebra were developed. The first prototype was implemented using Sicstus Prolog [49]. The prototype serves as an experimental environment for studying theoretical aspects

of object-oriented data models and object algebras. The second prototype is implemented as an extension of the database programming language E [21]. It is used for studying the problems of QAL integration with the C++-based database programming language. In both prototypes we did not address the problems of efficient implementation and query optimisation. Some aspects of the Prolog and E-based implementations are presented in the following sub-sections.

5.1 The Prolog-based Prototype

We aimed to study previously presented theoretical aspects of the QAL algebra using the prototype which is flexible enough to be able to experiment with the data model and the operations of QAL. The Prolog-based prototype is not tied to any particular type system or subject to any similar constraint, unlike, for example, our second prototype, which extends the E database programming language, and is thus constrained by the E type system. The basic motives in the Prolog-based implementation of QAL were the study of: (i) the operations for the efficient manipulation of complex objects, (ii) the implementation of the substitutability principle [35] in the context of a query language, (iii) the optimisation of queries which include model-based operations, and, (iv) the implementation of static type checking of QAL queries.

The prototype comprises a simple user-interface module, a query evaluation module, and a storage manager module. The query evaluator consists of the query interpreter, query optimisation, and type-checking sub-modules. The query interpreter is designed similarly to the query execution sub-system of Postgres extended relational database system [12, pp.78]. The query optimiser is currently under development. In the current version of the prototype, we implemented some ad-hoc optimisation rules necessary to speed up the evaluation of the most critical types of nested queries. Static type checking is implemented to provide the user with diagnostics for type errors [42]. The type checking algorithm is based on the ideas presented by Cardelli in [13]. The QAL storage manager is implemented on top of Postgres DBMS. The external representation of objects is based on the subset of the structural part of F-Logic [26]. A more detailed description of the prototype implementation can be found in [42].

5.2 Extending the Database Programming Language E

The basic intention of the second prototype implementation is to study QAL integration with a C++-based database programming language [21, 31, 4] (abbr. DBPL), which itself provides programming constructs for the manipulation of collections of database objects. The integration of QAL in a C++-based DBPL extends the functionality of the DBPL by adding to it facilities for declarative manipulation of database objects. The problems that arise in the integration of procedural programming languages and declarative query languages are the consequences of the different syntax and semantics of the two languages. They are often referred to as the *impedance mismatch problem* [5].

The extension of the E DBPL with the constructs of the query language QAL is realised by extending the semantics of *dot expressions* which are commonly used for accessing record components in procedural programming languages (e.g. C programming language). The ordinary dot expressions are extended to

subsume functional queries expressed by the QAL query algebra. From our experience, the functional nature of QAL and its comprehensible operations make the language suitable for integration with an object-oriented programming language.

The integration of the E language with the QAL query algebra is implemented by a preprocessor that is realised as an extension of the existing C++ parser `cppp` [19]. The preprocessor is written using the standard Unix tools Yacc and Lex. The C++ parser `cppp` is extended to: (i) recognise E constructs, and (ii) compile QAL expressions into E code. The currently implemented features of the query language are: support for class extensions and QAL sets, the operation **apply** restricted to the evaluation of the class data members and member functions, and the operation **select**.

The sets are implemented using the class template **setof**`<T>`. The generic type `T` can be any legal E **dbtype**. Further, each E **dbclass** is extended by the concept of a *class extension*. The class extension is a set of object identifiers that refer to the class *members*. Class extensions obey the following restrictions: (i) they are defined only for **dbclass**-es[5], (ii) a class extension is a persistent set of **dbpointer**-s, (iii) they include only the persistent instances of the **dbclass**, and (iv) an instance object can be an element of the single class extension. The operations **ext** and **exts**, which are defined in Section 3.2, are used for accessing the class extensions.

Example 5.1 *This example presents the use of the operation* **select** *in the E language extended with the constructs of the QAL query algebra. The operation* **select** *filters the set* `p` *by selecting from it persons who are younger than the value of the variable* `max_age` *and do not work in Ljubljana. The variable* **this** *denotes an implicit variable which iterates through the argument set of the operation* **select**.

```
int max_age;
setof<person> p;
...
setof<person> ps = p.select( this->age < max_age &&
                            !(this->works in institution.ext.
                                          select( this->address = "Ljubljana" )));
```

6 Related Work

Almost all recent database algebras have evolved from the relational database algebra proposed by Codd in [16]. Although some algebras do not include the operations of relational algebra, each of them includes the ability to compute selection, standard set operations, projection and Cartesian product. This is somehow natural, since the algebraic structures of recent algebras subsume or extend flat relations. In the QAL algebra, some of the relational algebra operations are generalised to be able to manipulate objects, while some have retained their original semantics. The QAL operation **tuple**, for instance, generalises the relational operation **project**. On the other hand, the QAL set operations (e.g. **union**) are defined as in the relational algebra.

The work on the QAL algebra has been influenced by the early functional query language FQL proposed

[5]Class extensions are not defined for ordinary C++ classes.

by Buneman and Frankel [11], by some of its descendants, that is, GDL [9] and O^2FDL [34], and by the functional database programming language FAD [18]. The algebra is by its nature a functional language, where the operations can be combined by the use of functional composition and higher-order functions to form the language expressions. The algebraic structures manipulated by FQL are entities represented by the functional data model. The QAL algebra can be treated as a generalisation of FQL for the manipulation of objects. It subsumes the operations of FQL, i.e. operations **extension**, **restriction**, **selection** and **composition**, as well as the extensions of FQL provided by the query languages GDL [9] and O^2FDL [34].

Further, the QAL algebra is closely related to the Query Algebra originally proposed by Shaw and Zdonik in [47]. This algebra has been further used as the basis for the EXCESS/EQUAL algebra [35] and its query optimiser, and for the algebra Aqua [32]. QAL shares many similarities with these algebras in some of the basic operations (i.e. **select**, **tuple** and set manipulation operations), while it differs in the operations for restructuring complex objects and for querying nested components of complex objects. In addition, QAL includes a rich set of primitive operations which serve for querying database schemata.

In the following sub-sections, we present the approaches of other algebras and query languages to the two problems addressed in this work. First, the existing approaches to querying conceptual schemata are presented. Next, the existing operations for querying nested components of complex objects are overviewed.

6.1 The Use of Schema for Querying Databases

While the problem of querying conceptual schema of object-oriented databases has, to our knowledge, not been addressed by recent database algebras, there are some query languages which include facilities to manipulate schema.

In this respect, QAL was significantly influenced by F-Logic as proposed by Kifer, Lausen and Wu in [24] and [26]. F-Logic is a declarative language which incorporates the constructs of object-oriented data model and the frame-based knowledge representation languages into logic. The possibility of querying data and schema portions of a database emerged from frame-based knowledge representation languages which use the same language constructs for the representation of the schema and the data portions of the database. In brief, F-Logic provides the following possibilities for querying database schema. Firstly, the relationships between instances and classes, which are based on the *isa* hierarchy of database objects, can in F-Logic be queried using the predefined predicates for testing class membership and the subclass relationship. Secondly, F-Logic provides facilities to explore the properties of individual and class objects by treating attributes and methods as objects that can be manipulated in a similar manner to other database objects. This allows, among other things, inquiring about some types of non-trivial relationships among objects such as the analogy and the similarity relationships.

Similarly to F-Logic, the query language XSQL [25] includes a set of constructs for querying database schema. XSQL queries can include variables that range over class objects. Therefore, classes can be queried on the basis of their properties and the properties of their instances. The XSQL operation **subclassOf** can, in this context, be used to inquire about the relationships among classes which are based on the inheritance

hierarchy of classes. In a similar manner to F-Logic, XSQL also treats attributes and methods as objects that can be queried; hence, a user can inquire about the properties of individual and class objects.

Further, the work on QAL was influenced by the work of Papazoglou presented in [38]. He suggests a set of high-level operations for expressing intensional queries which aid the user in understanding the meaning of stored data. The proposed operations can express the following types of queries: relate individual objects to classes, browse the *isa* hierarchy of classes, inquire about the class properties, compute associations among classes which are not related by *isa* relationship, locate objects on the similarity basis, and inquire about the dynamic evolution of objects represented by *roles*.

A query language which is also closely related to the QAL query algebra is the query language P/FDM [22]. P/FDM is based on the Functional Database Model [48] (abbr. FDM). It extends FDM by incorporating in the language the constructs of the object-oriented database model. P/FDM is implemented using the Prolog programming language. The database comprises the data and metadata entities. Every metadata entity, a class or a function for example, is described using a set of predefined predicates. The Prolog predicates which serve as the interface to the database, and the query language P/FDM which is implemented by the use of these predicates, provide uniform access to the data and metadata entities. These facilities allow for querying the properties of metadata entities and some types of relationships among them.

The operations for querying database schema provided by the query languages of recent object-oriented databases management systems (i.e., query languages of ORION [28], O_2 DBMS [8, 20], Postgres [12, 40] or ObjectStore [36, 37]) are briefly overviewed. Most recent query languages allow the use of class extensions in queries through a function which maps a class onto: the set of its members and/or the set of its instances. ORION [28] provides a set of operations to modify database schema at different levels: modification of inheritance, class properties, methods, and inheritance hierarchy of classes. However, it does not include additional constructs which would allow querying of conceptual schema and the relationships between the data and schema parts of the database. Next, in [10], Bertino proposes the use of the operator **class_of**, which returns the class of an object at run-time. The resulting class can be further used in queries. Finally, the ODMG standard [14] includes a set of operations for accessing the properties of metadata. The information that can be obtained about a type is: the set of operations defined for a given type, the set of its attributes, the set of its supertypes and subtypes, and the extent of a given type. The ODMG standard does not provide uniform access to the intensional and extensional levels of the OO database in the way proposed by our model-based operations. Moreover, some of our model-based operations can not be expressed using only ODMG types. Such operations include poset comparison operations defined on structured o-values and the operations **lub-set** and **glb-set**.

6.2 Querying Complex Objects

We base our discussion on the operations for the manipulation of complex objects on the classification of operations on complex objects introduced by Kim et al. in [27]. The operations for querying complex objects are classified into *basic* and *advanced* operations. Basic operations include: (i) the retrieval of complete

complex objects, (ii) the projection of the selected attribute values, (iii) the restriction predicates on the nested components of complex objects, and (iv) three types of object update operation: insertion, deletion and modification of complex objects. The advanced operations on complex objects are: (i) the complex **join** operation which allows for joining complex objects at different nesting levels, and (ii) the recursive queries by which the recursively structured hierarchies of complex objects can be manipulated.

The operations discussed by Kim at al. [27] are closely related to the operations of nested relational algebras [1, 33, 41]. The basic types of operations presented above can be realised by the operations of most NF^2 algebras, if, however, the algebraic structures are restricted to nested relations. The use of NF^2 restructuring operations for querying nested components of complex objects is computationally expensive since it requires the reconstruction of the complex object up to the manipulated component. On the other hand, the NF^2 algebra VERSO introduced in [1] includes very expressive operations which provide the means to realise complex forms of selection, projection and join operations on complex objects. The **join** operation of VERSO, for instance, allows for the expression of some forms of complex **join** as presented in Kim's classification.

The basic queries on complex objects defined by Kim et al. can be expressed using the operations of object algebras [47, 32, 53, 35, 15]. As discussed in Section 1.2, these algebras include either the operations evolved from NF^2 algebras, the operations derived from functional query languages, or both. The first type of operation for querying complex objects includes variants of the NF^2 operations **nest** and **unnest**, which serve for creating and unnesting component relations in an argument complex object [41, 1]. The second type of operation for querying complex object includes operations which evolved from the functional operations **apply-to-all** and tuple **construction** [7]. Both types of operations are referred to as the *restructuring operations*. The restructuring operations are suitable for expressing the queries which restructure complex objects or which extract the components of complex objects. Let us now focus on the queries which manipulate nested components of complex objects.

The problems which appear when using the restructuring operations for querying nested components of complex objects are as follows. Firstly, it has been shown by Roth and Korth in [41] that the restructuring operations **nest** and **unnest** are not inverse in general. In order to safely restructure component relations, additional constraints have to be satisfied by the complex objects otherwise information may be lost during the restructuring of the target components. In particular, the NF^2 relations have to satisfy the partitioned normal form and even some more strict conditions (required for the operation **nest** to retain partitioned normal form) in order to be safely restructured [41]. Another possibility for retaining the information about the unnested groups of objects is to index unnested tuples [17]. The second problem of restructuring operations relates to the reciprocity property of operations used to create and flatten sets of sets; the operations **group** and **flatten**, as defined in [47], for instance. Such operations are not inverse in the sense that the operations **nest** and **unnest** are. Hence, it is not possible to restore the original state of a flattened set of sets unless some additional technique is used to store information about the groupings of objects. Finally, the evaluation of the query which includes a sequence of restructuring operations is computationally

expensive: it reconstructs the complex objects up to the target nested components, and it may require the creation of a sequence of temporal objects [23].

In contrast to the approach to querying nested components of complex objects using restructuring operations, the use of the operation **apply_at** does not require the evaluation of a sequence of restructuring operations on the argument complex objects in order to access theirs nested components. Namely, it can be implemented as an iterator which accesses nested components of complex objects, passes the evaluation of the parameter function of **apply_at** to the query sub-tree which realises the parameter function, and, finally, collects and assembles results of the parameter function. In this way, the above stated problems, which appear when the restructuring operations are used to access nested components of complex objects, are avoided.

An approach to querying complex objects, which could be treated as the above stated second approach to querying complex objects (i.e., including the operations which evolved from functional languages), is suggested by Abiteboul and Beeri with Complex Object Algebra [2] (abbr. COA). The operations of COA can express the basic operations on complex objects as defined by Kim et al. in [27]. The algebra introduces the operation **replace** to provide facilities to restructure complex objects and to apply queries to the nested structures of complex objects. The operation **replace** is a generalisation of the well-known functional operation **apply-to-all** [7]. It realises the application of the parameter query on the set of objects. The parameter query of the **replace** operation is specified in a functional notation called *replace specifications* which subsumes all definable queries of COA. In addition, the notation allows the queries to be structured by the arbitrary use of the *set* and *tuple* constructors. In this way, the **replace** parameter forms a "structured query" which, when applied to the set of values, can be used to restructure and query complex objects.

Finally, the recursive algebras proposed by Colby in [17] and Liu in [33] introduce another approach to querying complex objects. Both algebras evolved from nested relational algebras [41, 1]. The operations of these algebras are defined recursively so that each of the operations can be recursively applied to the components of the argument complex object. Each recursive operation includes a *list* which directs the evaluation of the operations by specifying the components of complex objects, to which the evaluation of the given operation is recursively transferred. The operations of recursive algebras are related to the QAL operation **apply_at** since both provide the means for directing the evaluation of operations to the nested components of the argument complex object. However, in the QAL query algebra a single operation (i.e. **apply_at**) serves as the "navigator" for directing the evaluation of other operations, whereas in recursive algebras *all* operations, except for the restructuring operations, include a *list* which directs their evaluation.

7 Concluding remarks

In this paper we have presented the QAL query algebra. QAL includes a set of simple operations, referred to as model-based operations, intended for inquiring and reasoning about the properties of individual and class objects. The salient feature of the model-based operations is their close relationship to the underlying data model formalisation, which unifies the representation of the intensional and the extensional parts of a

database. As a consequence of this relationship with the data model of QAL, the model-based operations provide a uniform access to the properties of individual and class objects. They can serve as the means for expressing intensional queries on object-oriented databases, as well as extensional queries that refer to the properties of a conceptual schema.

The declarative operations of the QAL query algebra are used for querying a database. They evolved from the operations of relational algebra, NF^2 algebras and functional query languages. The set of declarative operations includes the operation **apply_at**, which is used for querying nested components of complex objects. The operation generalises the well-known operation **Apply-To-All** [7] by adding to it a new parameter which specifies the path to the nested component where the parameter query is evaluated. Its semantics is simple and comprehensible, and it can serve as the basis for the efficient implementation of queries that manipulate nested components of complex objects.

There is a list of tasks that remain to be done or completed on the QAL query algebra. Here we present only the most important. Firstly, the algebraic properties of the newly proposed operations are studied in order to be utilised for query optimisation. Secondly, study of the expressive power of QAL is currently under way. We are addressing the problem by studying the relationships between QAL and other database algebras and logic based languages. Finally, the current efforts on the QAL implementation show that it can serve to extend C or C++ based database programming languages with declarative facilities. The efficient implementation of the integration of QAL and such a DBPL remains to be completed.

Acknowledgments

The authors wish to thank the anonymous referees for their helpful comments on the draft versions of this paper. In particular, we thank the referee who gave us detailed and constructive suggestions which have significantly improved the readability and the quality of the paper. Further, we thank prof. Mike Papazoglou for his advise and support during the early stages of the presented work, Vanja Josifovski for his work on the prototype of QAL based on GNU E DBPL, and prof. Michael Kifer for his suggestions on the draft version of this paper. The work of Iztok Savnik was supported by Republic of Slovenia Ministry of Science and Technology Grant J2-7627.

References

[1] S. Abiteboul, N. Bidoit, *Non First Normal Form Relations: An Algebra Allowing Data Restructuring*, Journal of Comp. and System Science 33, 1986, pp. 361-393.

[2] S. Abiteboul, C. Beeri, *On the Power of the Languages For the Manipulation of Complex Objects*, Verso Report No.4, INRIA, France, Dec. 1993.

[3] S. Abiteboul, P.C. Kanellakis, *Object Identity as Query Language Primitive*, Proc. of the ACM Conf. on Management of Data, 1989, pp. 159-173.

[4] R. Agrawal, N.H. Gehani, *ODE (Object Database and Environment): The Language and Data Model*, Proc. of the ACM Conf. on Management of Data, 1989, pp. 36-45.

[5] J. Annevelik, *Database Programming Languages: A Functional Approach*, Proc. of the ACM Conf. on Management of Data, 1991, pp. 318-327.

[6] M. Atkinson, et al. *The Object-Oriented Database Systems Manifesto*, Proc. of First Int'l Conf Deductive and Object-Oriented Databases, North Holland, 1989, pp. 40-57.

[7] J. Backus, *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*, Communications of the ACM, 21(8), August 1978, pp. 613-641.

[8] F. Banchilion, S. Cluet, C. Delobel, *A Query Language for the O_2 Object-Oriented Database System*, Proc. 2nd Workshop on Database Programming Languages, 1989, pp. 122-137.

[9] D.S. Batory, T.Y. Leung, T.E. Wise, *Implementation Concepts for an Extensible Data Model and Data Language*, ACM Trans. on Database Systems, 13(3), Sept. 1988, pp. 231-262.

[10] E. Bertino et al., *Object-Oriented Query Languages: The Notion and Issues*, IEEE Trans. on Knowledge and Data Engineering, 4(3), June 1992, pp. 223-237.

[11] P. Buneman, R.E. Frankel, *FQL- A Functional Query Language*, Proc. of the ACM Conf. on Management of Data, 1979, pp. 52-58.

[12] *Next-Generation Database Systems*, Communications of the ACM, 34(10), 1991.

[13] L. Cardelli, *A Semantic of Multiple Inheritance*, Information and Computation, 76, 1988, pp. 138-164.

[14] R.G.G. Cattell (Editor), *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, 1993.

[15] S. Ceri et al., *Algres: An Advanced Database System for Complex Applications*, IEEE Software, July 1990, pp. 68-78.

[16] E.F. Codd, *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, 13(6), June 1970, pp. 377-387.

[17] L.S. Colby, *A Recursive Algebra and Query Optimization for Nested Relations*, Proc. of the ACM Conf. on Management of Data, 1989, pp. 273-283.

[18] S. Danforth, P. Valduriez, *A FAD for Data Intensive Applications*, IEEE Trans. on Knowledge and Data Engineering, 4(1), Feb. 1992, pp. 34-51.

[19] T. Davis, *cppp documentation*, Brown University, 1993-94.

[20] O. Deux et al., *The Story of O2*, IEEE Trans. on Knowledge and Data Engineering, 2(1), March 1990, pp. 91-108.

[21] *An Introduction to GNU E, The E Reference Manual* and *The Design of the E Programming Language*, Exodus Project Documents, University of Wisconsin-Madison.

[22] S.M. Embury, Z. Jiao, P.M.D. Gray, *Using Prolog to Provide Access to Metadata in an Object-Oriented Database*, Practical Application of Prolog, 1992.

[23] G. Graefe, *Query Evaluation Techniques for Large Databases*, ACM Comp. Surveys, Vol.25, No.2, June 1993, pp. 73-170.

[24] M. Kifer, G. Lausen, *F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme*, Proc. of the ACM Conf. on Management of Data, 1989, pp. 134-146.

[25] M. Kifer, W. Kim, Y. Sagiv, *Querying Object-Oriented Databases*, Proc. of the ACM Conf. on Management of Data, 1992, pp. 393-402.

[26] M. Kifer, G. Lausen, J. Wu, *Logical Foundations of Object-Oriented and Frame-Based Languages*, Journal of the ACM, 42(4), July 1995, pp. 741-843.

[27] W. Kim, H.-T. Chou, J. Banerjee, *Operations and Implementation of Complex Objects*, IEEE Trans. on Software Engineering, 14(7), July 1988, pp. 985-996.

[28] W. Kim, et al., *Features of the ORION Object-Oriented Database System*, 11th Chapter in *Object-Oriented Concepts, Databases and Applications*, W.Kim (ed.), Sept. 1989, pp. 251-282.

[29] G.M. Kuper, M.Y. Vardi, *A New Approach to Database Logic*, Proc. of the ACM Conf. on Management of Data, 1984, pp. 86-96.

[30] C. Lecluse, P. Richard, F. Velez, *O_2, an Object-Oriented Data Model*, Proc. of the ACM Conf. on Management of Data, 1988, pp. 424-433.

[31] C. Lecluse, P. Richard, *The O_2 Database Programming Language*, Proc. of the 15th Conf. on Very Large Databases, 1989, pp. 411-421.

[32] T.W. Leung, et al., *The Aqua Data Model And Algebra*, Technical Report No. CS-93-09, Brown University, March 1993.

[33] L. Liu, *A formal approach to Structure, Algebra & Communications of Complex Objects*, Ph.D. thesis, 1992, Tilburg University.

[34] M. Mannino, I.J. Choi. D.S. Batory, *The Object-Oriented Data Language*, IEEE Trans. on Software Engineering, 16(11), Nov. 1990, pp. 1258-1272.

[35] G.A. Mitchell, *Extensible Query Processing in an Object-Oriented Database*, Ph.D. thesis, Brown University, 1993.

[36] J. Orenstein, S. Haradhvala, B. Margulies, D. Sakahara, *Query Processing in the ObjectStore Database System*, Proc. of the ACM Conf. on Management of Data, 1992, pp. 403-412.

[37] *ObjectStore User Guide*, Release 2, ObjectStore Design, Inc., 1992.

[38] M.P. Papazoglou, *Unravelling the Semantics of Conceptual Schemas*, Communications of the ACM, Sept. 1995.

[39] S.L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[40] *The Postgres Documents*, Computer Science Department, University of California at Berkeley, 1994.

[41] M.A. Roth, H.F. Korth, A. Silberschatz *Extended algebra and calculus for non 1NF relational databases*, ACM Trans. Database Systems, 13(4), 1988, pp. 389-417.

[42] I. Savnik, *A Query Language for Complex Database Objects*, Ph.D. thesis, University of Ljubljana, CSD Technical Report, Jozef Stefan Institute, CSD-TR-95-6, Jun 1995.

[43] I. Savnik, Z. Tari, *Querying Conceptual Schemata of Object-Oriented Databases*, Proc. of Database and Expert Systems Applications Workshop, IEEE Comp. Society, 1996, pp. 384-390.

[44] I. Savnik, Z. Tari, *Querying Objects with Complex Static Structure*, Proc. of Int. Conf. on Flexible Query Answering Systems, Roskilde, Denmark, To appear in Lecture Notes in Artificial Intelligence, 1998.

[45] I. Savnik, T. Mohorič, Z. Tari, *Unifying Schema and Instance Levels of Object-Oriented Database*, CSD Technical Report, Jožef Stefan Institute, May 1998, In preparation.

[46] M.H. Scholl, H.-J. Schek, *A Synthesis of Complex Objects and Object-Orientation*, OO Databases: Analysis, Design & Construction (DS-4), R.A.Meersman, W.Kent, S.Khosla (Eds.), Elsevier Science Publishers, 1991.

[47] G.M. Shaw, S.B. Zdonik, *A Query Algebra for Object-Oriented Databases*, Proc. of IEEE Conf. on Data Engineering, 1990, pp. 154-162.

[48] D.W. Shipman, *The Functional Data Model and the Data Language DAPLEX*, ACM Trans. on Database Systems, 6(1), March 1981, pp. 140-173.

[49] *SICStus Prolog Users's Manual*, Swedish Institute of Computer Science, October 1991.

[50] S.Y. Su, *Modelling Integrated Manufacturing Data with SAM**, IEEE Computer, Jan. 1986, pp. 34-49.

[51] S.Y. Su, M. Gou, H. Lam, *Association Algebra: A Mathematical Foundation for Object-Oriented Databases*, IEEE Trans. on Knowledge and Data Engineering, 5(5), Oct. 1993, pp. 775-798.

[52] J. Tillquist, F.-Y. Kuo, *An approach to the recursive retrieval problem in the relational databases*, Communications of the ACM, 32(2), Feb. 1989, pp. 239-245.

[53] B. Vance, *Towards an object-oriented query algebra*, Tech. Report CS/E91-008, Dept. Comp. Science and Eng., Oregon Graduate Institute, Jan. 1991.

[54] S.L. Vandenberg, D.J. DeWitt, *Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance*, Proc. of the ACM Conf. on Management of Data, 1991, pp. 158-167.

[55] S.L. Vandenberg, *Algebras for Object-Oriented Query Languages*, Ph.D. thesis, Technical Report No. 1161, University of Wisconsin, July 1993.