A Formalization of ODMG Queries

Holger Riedel and Marc H. Scholl

Department of Mathematics and Computer Science
University of Konstanz
D-78457 Konstanz
Germany

email: {Holger.Riedel,Marc.Scholl}@uni-konstanz.de

Abstract

The ODMG proposal has helped to focus the work on object-oriented databases (OODBs) onto a common object model and query language. Nevertheless there are several shortcomings of the current proposal stemming from the adaption of concepts of object-oriented programming and a lack of formalization. In this paper we present a formalization of the ODMG model and the OQL query language that is used in the CROQUE project as a basis for query optimization. An essential part is a complete, formally sound type system that allows us to reason about the types of intermediate query results and gives rise to fully orthogonal queries, including useful extensions of projections and set operations.

1 INTRODUCTION

For a long time, the evolution of OODB seemed to disperse in quite different directions: there were rather distinct object-oriented database models (OODMs) either based on (nested) relational formalisms or OOPL-like notions, and hardly any consensus about the structure and formalization of queries. Nowadays, researchers and commercial products try to find a common language, usually using the notations of the ODMG (Cattell 1996) in order to introduce their specific concepts. It seems that many ideas which appeared rather different and contradictory – like OOPL-style versus SQL-like programming or value-based databases versus object notions – can be put together in order to get a full-fledged OODBS in the future. Nevertheless, several problems remain, some of which are attacked in this paper:

- *OO data models and query languages:* Up to now there is quite a big gap between the advanced OODMs of several research projects and the rather simple OODMs used in commercial OODBS. While the latter ones are mostly restricted to the underlying OOPL, more advanced models offer nice features such as orthogonal subtyping, the explicit distinction of class hierarchies and type hierarchies, support for objects *and* values (without object identity), and an orthogonal query language, based on formal (e.g. logic) definitions.
  The ODMG model is somewhere in between. While there are some "goodies" such as arbitrarily complex types, and the distinction of values (called immutable objects or literals) and objects (also named mutable objects), several aspects are not present or clarified until now:

  - Objects are handled like in OOPLs. That is, there is only a type hierarchy, while reasoning about object collections ("subclassing") is not possible, because an object formally only belongs to *one* specific class (where it was created). Several research projects, however, have shown that, in a database context, the ability to arrange object collections in an inclusion hierarchy provides a powerful basis for concepts such as integrity constraints, views, and derivation rules (Scholl, Laasch, Rich, Schek and Tresch 1993, Kuno and Rundensteiner 1996, Ceri and Manthey 1993).
  - While the given set of query operations of ODMG-OQL-1.2 seems rather complete for practical purposes, there is a lack of a formalization of such queries, which is essential for query optimization. In its current form the ODMG standard is more like a skeleton for data and query models rather than a sound formal model itself.
  - The ODMG standard does not provide any declarative object manipulation operations, except for object creation. As a consequence, update transactions completely rely on the OOPL the OODB is bound to (e.g. C++). Obviously, the OODBMS can hardly be expected to provide any help in analyzing transactions, for example to identify conflicts for semantic concurrency control, to check preservation of integrity constraints, or to derive update propagation rules, if they are coded in an imperative language. The standard completely lacks an object manipulation language (OML), e.g. in the style of update, delete, and insert statements of SQL (see also (Laasch and Scholl 1992)).

- *OO optimization and query processing:* While relational query processing has been investigated thoroughly, there are only initial frameworks for OO query processing and optimization. Up to now, it is not clear how to integrate these first ideas in order to get efficiency for all ODMG queries. Many issues are open, especially the interaction of query optimization mecha-

nisms with useful physical database design choices that need to be offered for the storage of object databases.

In the CROQUE project*, we are designing and implementing an object database system. We use the syntax of the ODMG proposal and added the missing formalization. In order to get a clean formal approach, we changed minor parts of the original ODMG proposal (Cattell 1996), which are explained in detail in the next section. More specifically, we extended our preliminary work on the OODB models COCOON (Scholl et al. 1993) and EXTREM (Hörner and Heuer 1991) in order to cover the concepts of (Cattell 1996). We added a general typing theory for mutable and immutable objects which is used as a basis of the formalization of the query language. Our rigid type system allows only well-typed ODMG queries and we could extend the applicability of set operations for collections of both mutable and immutable objects. Also there is a cast operation (to supertypes) based on the type structure which allows a flexible kind of projection of complex structured values. In the case of mutable objects, this is similar to "object-preserving" projections of other OOQLs. Such a concept is not present in the current ODMG proposal.

Very recently, the ODMG standard 2.0 was released. We plan to integrate the concepts of the new proposal into the CROQUE model. In this paper, all references to the ODMG proposal refer to the ODMG standard 1.2, but we give an overview of how the ODMG 2.0 standard may be combined with our proposal in the conclusion. Here we only state that the changes of ODMG 2.0 can be integrated into our approach in a somewhat straightforward manner.

The rest of this paper is organized as follows: In the next section we explain the differences between our approach and the ODMG proposal. Additionally, we give an overview of formalization efforts in the field of object-oriented database languages related to the concepts of the ODMG approach. Section 3 presents the formal model (ODL), an analysis of the formalization of CROQUE-OQL is given in Section 4, while an excerpt of the formalization of the query language is given in the Appendix. A complete formalization can be found in (Riedel and Scholl 1996).

2 COMPARISON AND RELATED WORK

The ODMG query language is rather rich of concepts, so that the claim of its inventors that it "can easily be formalized" (Cattell 1996) must be seen rather critical. We made the following observations:

- Because the concepts of the query language are only introduced by (rather

*CROQUE (a shorthand for Cost and Rule-based Optimization of object-database QUEries) is a joint project of U Konstanz with U Rostock, partially funded by the German National Research Fund (DFG).

simple) examples, it is not always clear what the exact intended semantics would be for more complex queries.
- The use of run-time exceptions in the query language seems to be a bad design choice for a query language, because further control strategies are necessary, when OQL is used within application programs. It seems rather odd to allow queries, which behave harmful in certain circumstances. Additionally, the analysis of the potential for query optimization is rather difficult, if exceptional cases have to be considered all over the place.

In order to get a clean formalization, we took a special view the ODMG model and query language guided by commonly accepted design principles for OOQLs (Atkinson, Bancilhon, DeWitt, Dittrich, Maier and Zdonik 1989, Yu and Osborn 1991, Heuer and Scholl 1991):

- The CROQUE data model is based on the ideas of Beeri's OODB model (Beeri 1990) and more directly on COCOON and EXTREM. It can handle values (immutable objects) as well as objects, and orthogonal type constructors (tuples, sets, bags, lists, and arrays) are supported. More specifically, a rigorous, strict type system for mutable and immutable objects is the basis for a closed query language where each query result is a part of the data model.
- The query language is statically typed. Thus, queries can be type-checked at compile-time. Only well-typed queries are allowed, so that some ODMG queries are excluded in our approach. But the intention of those queries can easily be achieved using a different syntactical form. Furthermore we had to introduce null values in query results to capture the `element` operator applied to non-singletons.
- In CROQUE-OQL, as far as presented in this paper, the treatment of methods in queries is incomplete. A method call in a query is currently treated like an access to an attribute, thus we only check whether it is well-typed using the presented type system, and thus make sure that there are no runtime exceptions. This is enough for our initial purpose, namely building a query optimizer for declarative OQL queries. A more complete treatment of methods is part of our future work.
- The ODMG proposal is rather informal for set operations on complex values (structured immutable objects) and object types. Here, CROQUE provides orthogonal use of union, intersection, and set difference for arbitrary collections of objects and for all literals where the types have a common supertype (`union` and `except`) or a common subtype (`intersect`).

While the work presented in this paper covers the OQL language description as given in (Cattell 1996), the following open issues have to be worked out in more detail in the future:

- Although a *nil* value is mentioned in (Cattell 1996), it is open how nulls should be supported in OQL. Due to a lot of detail problems, we just took a simple approach by using a *null* just as another value of the specific domain. In our approach, selections only result in objects or values where the conditions evaluate to *true*. Thus, nulls are treated similar to *false* in this case. In other words, we return the true-result only, not the maybe-result (Biskup 1983). We mention that nulls are not newly introduced by queries, iff the database is null-free and *null* constants and the `element` operator are not present in the query.
  The main problem of a full-fledged treatment of nulls in OOQLs (and OQL in particular) is the interpretation of the database and the query results. It has to be clarified how null values can be used in the database and how the detail problems for the OQL clauses should be treated, especially when, by the use of query operations, null values migrate to object-valued attributes or represent the object identity.
- A major effort for the future will be the integration of ODMG-OQL and the upcoming SQL-3 standard. In (Cattell 1996) a rather simplistic approach is described, where SQL clauses can be realized as macros within the ODMG-OQL framework. It is not clear, though, how this may work together (efficiently) with more enhanced SQL statements, such as outer joins or the multiple use of aggregations in the `select` clause of select-from-where blocks. Also a lot of detail problems have to be solved when the ODMG proposal has to be integrated into the upcoming SQL/Object specification (Melton 1996).

Over the last years, quite a few attempts have been published that aim at providing clean formalizations for OQL, for instance (Bancilhon and Khoshafian 1989, Abiteboul and Kanellakis 1989, Straube 1991, Bertino, Negri, Pelagatti and Sbattella 1992, Kifer, Kim and Sagiv 1992, Kifer, Lausen and Wu 1993, Hohenstein and Engels 1992, Herzig and Gogolla 1994, Kamel, Wu and Su 1994, Fegaras and Maier 1995). However, only few of these integrate all concepts of the ODMG proposal into a single formal model. Usually they use a calculus-based or algebra-based paradigm as the basis of formalization, which results in covering only a subset of the OQL concepts. It is not clear whether they can be extended easily to incorporate the full spectrum. Here we comment on some of the research work in more detail.

- Early approaches like Straube's calculus and algebra (Straube and Özsu 1990, Straube 1991) are quite simple extensions of the relational approach, where the complex structures of the ODMG proposal can neither be built within the data model nor in the query language. On the other side, it was possible to derive results for the equivalence of calculus and algebra, simplification rules, and optimized evaluation plans. Nevertheless, due to

its simplicity, it seems that this approach is not appropiate for the ODMG model.

- An example of a rather complex object algebra is the AQUA algebra (Leung, Mitchell, Subramanian, Zdonik and other 1993) proposed for the EREQ project. Here several operators are defined in a functional way to capture different meanings of set operations. Because it explicitly handles different kinds of collections by different algebra operators, it is more suitable as an "internal" algebra for query processing, while OQL is more likely an interface language, where the evaluation problems are described on a different level. In fact, the EREQ project pursues quite similar goals as CROQUE. We, however, try to avoid the "explosion" of algebraic operators due to different collection types by using monoid comprehensions (Grust and Scholl 1996).
- Among different SQL-based extensions, XSQL (Kifer et al. 1992) uses F-logic (Kifer et al. 1993) to get a formal semantics. Therefore, the data model is more restricted than the ODMG approach, because the underlying data model only supports one-level set values. On the other side, queries can be defined as subclasses of other classes and path expressions are more flexible than in the ODMG-OQL approach. Also the well-typed application of method calls and the access to the meta-level are supported.
- Some formal approaches (like (Abiteboul and Kanellakis 1989, Kifer et al. 1993, Fegaras and Maier 1995)) can be seen as extensions of nested relational calculus. Usually, the clean treatment of lists, bags, and arrays within such languages is rather difficult, because the access and construction of such values is not as declarative as for sets. Also query languages for structured types face the problem that they have to be restricted syntactically in order to get a first order query language (Beeri 1989).
- A calculus-based formalization of SQL-like queries is given in (Hohenstein and Engels 1992) in the context of an extended ER model. The work is useful within the ODMG context for the calculus-based parts of OQL, like the `select-from-where` block. A similar approach (Herzig and Gogolla 1994) also treats arbitrarily structured objects, but both approaches do not explicitly support the general type system.
- A recent approach to integrate complex types into set expressions is done by Fegaras and Maier (Fegaras and Maier 1995) using so-called monoid comprehensions. This allows a generic argumentation on operations of different types. In contrast to the approach of this paper, the typing of the queries has to be done explicitly in the query expressions. Nevertheless, monoid comprehensions are rather useful to get a basis for OO query optimization and are also exploited in the CROQUE project (Grust and Scholl 1996, Grust, Kröger, Gluche, Heuer and Scholl 1997).

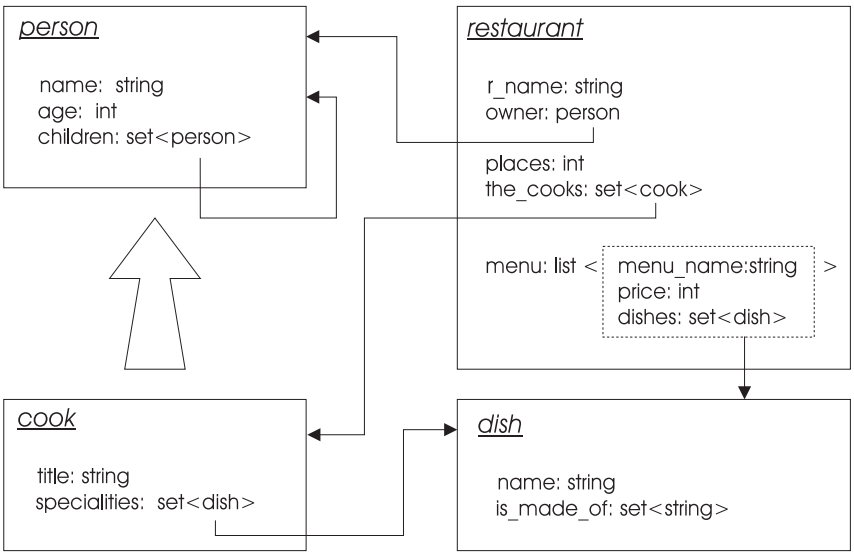

Figure 1 An ODMG schema

3 THE DATA MODEL AND ODL

This section presents the CROQUE approach to define a formal semantics of an OQL. We show the flavor of CROQUE and the differences to ODMG by the running example of the paper. The example database stores information about restaurants, their menus and employees. A graphical notation of its type structure is given in Figure 1, where we extended the graphical notations of ODMG slightly in order to capture the complex types directly.

The ODMG model is a *type-based* object model, where information about object collections need not be present in the schema. In the figure above, each box represents an object type with its attributes. Simple arrows point to the type of component objects, while the thick arrow denotes the subtype relationship. Possible collection types are sets, bags, lists, and arrays.

Our formalization builds upon the BCOOL model presented in (Laasch and Scholl 1993a). While our primary goal is to formalize the ODMG object model, we took the freedom to modify the model in the following two major respects:

1. *In CROQUE, mutable objects are atomic.*

   The replication of large parts of the ODMG (meta-) type system according to the distinction of mutable and immutable structured objects seems unnecessary to us. We rather adopt the common understanding that all

structured "objects" are literals (i.e., structured *values*). ODMG's structured mutable objects would be represented as an atomic (mutable) object with one property that in turn contains the (immutable) structure in the CROQUE model. While this simplification might be debatable from an OOPL point of view, in our context of a semantical object model it is only a minor issue.

2. *In CROQUE, objects can be an instance of multiple types (at the same time, and—by means of gain/lose operations—throughout their lifetime).*

   Both features are not included in the ODMG proposal, but they are mentioned as planned for the final release. In order to provide more flexible (object-preserving) query functionality, we added this from the beginning (see also (Laasch and Scholl 1993a, Schek and Scholl 1990)).

Furthermore, we adopt the BCOOL approach to arrange object types into a *lattice* (as opposed to just an arbitrary (multiple) hierarchy), such that (object-preserving) projections ("casts" in the OQL terminology) need not be restricted to named supertypes present in the ODB schema. ODL types are formalized as follows: there is one basic sort for mutable objects (the domain of object identifiers). Our type system builds a lattice of (named and unnamed) object types below this basic sort. Each ODL-object type defines a named type by the set of "characteristics" (attributes and relationships, operations). ODL characteristics are formalized as functions (attributes are literal-valued functions, relationships are object-valued – possibly multi-valued) functions. ODL operations represent methods, that is, computed properties (no side-effects) and update operations (with side-effects).

In the sequel, we define a formal type system for the CROQUE-version of ODL. *Basic types* describe (pairwise) disjoint sets of instances. There are several basic types for the atomic literals (δ_{Int}, δ_{Bool}, ...) plus one basic type for mutable objects (δ_{Object}), on which *object types* can be defined by subtyping (see below). *Structured types* can be specified using the built-in type constructors set ($\{\ \}$), bag ($\{\!\{\ \}\!\}$), list ($\langle\rangle$), array ($[]$), struct ($()$), and function ($\rightarrow$). Types serve several purposes: (i) they represent a "repository" of possible values (this will be called the *domain* of the type below, an intensional notion of type); (ii) they are used by the compiler for type checking (i.e., assuring that only "(type) valid" expressions are ever executed). For example, we would not allow to compute the square root of a string. Finally, (iii) types can be used as containers (collections) for those values of that type, which are currently "in use" in the database (in ODL: "with extension"). The latter use of types (extensions) is typically only common with *mutable* types (where we will talk about the "active domain" of the type). Notice that, on the formal level, we will use active domains regardless of whether the ODB schema contains the "with extension" clause or not.

We use a denotational approach to specify the formal semantics. A formal language is defined syntactically, typing rules and semantics are then defined

for that language. As usual in the denotional approach, semantics are defined by giving denotation functions "$[\![\ldots]\!]$" that map syntactic constructs of the defined language to (operations on) the so-called "semantic domains". OQL's mapping to the formal language is straightforward and illustrated in examples throughout this paper.

Syntax. The syntax of a formal CROQUE type expression τ is given by the following list:

$$
\begin{array}{lll}
\tau = & \mid \delta_{Int} & /* \text{ INTEGER } */ \\
& \mid \delta_{Bool} & /* \text{ BOOLEAN } */ \\
& \vdots & \\
& \mid \delta_{Object} & /* \text{ mutable object sort } */ \\
& \mid [f_1, ..., f_n] & /* \text{ object types } */ \\
& \mid \{\ \tau\ \} & /* \text{ set types } */ \\
& \mid \{\!\{\ \tau\ \}\!\} & /* \text{ bag types } */ \\
& \mid \langle\ \tau\ \rangle & /* \text{ list types } */ \\
& \mid \tau[\,] & /* \text{ array types } */ \\
& \mid (\ \tau_1, \ldots, \tau_n\ ) & /* \text{ struct types } */ \\
& \mid \delta_{Object} \rightarrow \tau & /* \text{ function types } */
\end{array}
$$

Semantics. A type may be referenced by a label (e.g. [*name,age,children*] by *person*). Labelling is mandatory for structs but optional for other structured types. Therefore, we decide to define the semantics of a struct based on the given label (name equivalence), while other types are defined by their internal structure (structural equivalence). Therefore we use a set $\mathcal{LABELS}$ as the name space for such labels.

The semantic domain of values is defined by the following recursive domain equations:

$\mathcal{V} = \mathcal{D}_{Bool} \cup \mathcal{D}_{Int} \cup \mathcal{D}_{String} \cup \mathcal{D}_{Object} \cup \mathcal{F} \cup \mathcal{S} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{A} \cup \mathcal{T}$
$\mathcal{D}_{Bool} = \{\perp_{Bool}, true, false\}$,
$\mathcal{D}_{Int} = \{\perp_{Int}, 0, 1, -1, 2...\}$,
$\mathcal{D}_{String} = \{\perp_{String}, "a", "A", ...\}$,
$\mathcal{D}_{Object}$ contains countably infinite objects,
$\mathcal{F} = \mathcal{D}_{Object} \rightarrow_{fin} \mathcal{V}$,
$\mathcal{S} = \wp_{fin}(\mathcal{V})$,
$\mathcal{B} = \mathcal{V} \rightarrow_{fin} \{0, 1, 2, \ldots\}$,
$\mathcal{L} = \{0, 1, 2, \ldots\} \rightarrow_{fin} \mathcal{V}$,
$\mathcal{A} = \{0, 1, 2, \ldots\} \rightarrow_{fin} \mathcal{V}$,
$\mathcal{T} = \mathcal{LABELS} \rightarrow_{fin} \mathcal{V}$.

$\mathcal{D}_i$ are domains of basic values (e.g., boolean and integer). $\mathcal{F}$ denotes the domain of finite mappings from $\mathcal{D}_{Object}$ to $\mathcal{V}$, and $\mathcal{S}$ all finite powersets over $\mathcal{V}$. The domains for the other constructed types ($\mathcal{T}$ for structs, $\mathcal{L}$ for lists, $\mathcal{B}$ for bags, and $\mathcal{A}$ for arrays) are modelled as (finite) function domains mapping index values to elements (structs, arrays, and lists) and elements to positive integers (bags), respectively. Defining arrays like lists is sufficient for

our purposes and avoids some well-known problems with the formalization of functional arrays (Libkin, Machlin and Wong 1996). The type-specific bottom elements ($\perp_i$) denote undefined values. In order to improve readability we omit the type information and use $\perp$ instead in the sequel.

In general, equality must be defined for types on which sets are constructed (e.g., for testing set-membership). Because the equality for function types is undecidable in general, the domains of function types are restricted to objects. The equality on these restricted functions would be still undecidable, because the domain $\mathcal{D}_{Object}$ is infinite. However, since all instances of functions that can ever occur in any database state are restricted to the *active domains* of the corresponding object types (which are *finite* sets), all functions can be regarded as finite sets of pairs, such that equality is decidable. Hence, we do not need to separate types with equality from those without equality, which would be necessary otherwise.

Basic and Constructed Types. Except for object types, our semantics of types and subtyping is quite usual and follows (Balsters and de Vreeze 1991, Balsters and Fokkinga 1991, Mannino, Choi and Batory 1990): i.e., the denotations ($[\![\ldots]\!]$) of basic types are given by the following equations:

Definition Semantic Domain:

$[\![\,\delta_i\,]\!] = \mathcal{D}_i$, in case that $\mathcal{D}_i$ is a summand of $\mathcal{V}$
$[\![\,\tau_1 \rightarrow \tau_2\,]\!] = \{f \in \mathcal{F} \mid x \in [\![\,\tau_1\,]\!] \implies f(x) \in [\![\,\tau_2\,]\!]\ \}$
$[\![\,\{\ \tau\ \}\,]\!] = \{x \in \mathcal{S} \mid x \subseteq [\![\,\tau\,]\!]\}$
$[\![\,\{\!\{\tau\}\!\}\,]\!] = \{f | f : [\![\,\tau\,]\!] \rightarrow_{fin} \{0,1,2,\ldots\}\}$
$[\![\,\langle\tau\rangle\,]\!] = \{f | f : \{0,1,2,\ldots\} \rightarrow_{fin} [\![\,\tau\,]\!]\}$
$[\![\,\tau[\cdot]\,]\!] = \{f | f : \{0,1,2,\ldots\} \rightarrow_{fin} [\![\,\tau\,]\!]\}$
$[\![\,(L_1 : \tau_1, \ldots, L_n : \tau_n)\,]\!] = \{f : \{L_1, L_2, \ldots, L_n\} \rightarrow \bigcup_i [\![\,\tau_i\,]\!] | f(L_j) \in [\![\,\tau_j\,]\!]\ (j = 1, \ldots, n)\}$

Subtyping is used to describe (sub)-sets of objects with common interfaces, such that type-checking becomes more meaningful. The CROQUE definition of a subtype consists of three parts: a set of supertypes, a set of local characteristics, and (possibly) a type name. Any instance of the subtype is also an instance of its supertypes (*substitutability*), and all characteristics defined on the supertypes are applicable to the instances of the subtype (*inheritance of the interface*), in addition to the locally defined ones. Formally, object types need not be named, they are given by listing the set of characteristics.* For example, if *person* is an object type with attributes *name, age*, and a (set-valued) relationship *children*, and *cook* a subtype of *person*, with the additional attributes *title* and *specialities*, the two object types will be referred to as

*Notice that the ODMG policy of solving naming conflicts due to multiple inheritance by renaming leads to unique characteristics' names, and hence types are uniquely identified by the set of characteristics.

$[name, age, children]$ ($\equiv$ *person*) and $[name, age, children, title, specialities]$ ($\equiv$ *cook*). An example for structured values is the following.

Example 1 The literal type

$S \equiv$ `struct<`*r_name* : `string`,
places : `int`,
menu : `list<struct<`*menu_name* : `string`, *price* : `int>>`
`>`

is a subtype of

$T \equiv$ `struct<`*r_name* : `string`,
menu : `list<struct<`*menu_name* : `string>>`
`>`,

because the components of S are either the same as in T (here: *r_name*), or subtypes w.r.t. the same component of T (here: `list<struct<`*menu_name*: `string`,*price* : `int>>` is a subtype of `list<struct<`*menu_name*`:string>>` applying the subtype rules recursively), or (as a special case of struct types) there are components not present in the supertype (here: *places* and *price* in the component *menu*). □

The subtype relationship ($\preceq$) is based on set inclusion: i.e., if a type is defined as a subtype of another, then every instance of the subtype is also an instance of its supertype (which allows *substitutability*):

$$\tau_1 \preceq \tau_2 \implies [\![\, \tau_1 \,]\!] \subseteq [\![\, \tau_2 \,]\!].$$

This leads to the following inference rules for constructed types:*

Definition Subtyping:

[SETS] $\dfrac{\tau_1 \preceq \tau_2}{\{\tau_1\} \preceq \{\tau_2\}}$ [FUNS] $\dfrac{\tau_1^{dom} \preceq \tau_2^{dom},\ \tau_2^{rng} \preceq \tau_1^{rng}}{\tau_2^{dom} \to \tau_2^{rng} \preceq \tau_1^{dom} \to \tau_1^{rng}}$

[BAGS] $\dfrac{\tau_1 \preceq \tau_2}{\{\!\{\tau_1\}\!\} \preceq \{\!\{\tau_2\}\!\}}$ [ARRAYS] $\dfrac{\tau_1 \preceq \tau_2}{\tau_1[\cdot] \preceq \tau_2[\cdot]}$ [LISTS] $\dfrac{\tau_1 \preceq \tau_2}{\langle\tau_1\rangle \preceq \langle\tau_2\rangle}$

[STRUCTS] $\dfrac{\tau_1 \preceq \tau_1', \ldots, \tau_n \preceq \tau_n'}{(L_1 : \tau_1, \ldots, L_n : \tau_n, \ldots, L_m : \tau_m) \preceq (L_1 : \tau_1', \ldots, L_n : \tau_n')}$

---

* The horizontal bar corresponds to logical implication. Notice the antimonotonicity (contravariance) in [FUNS], which is needed for the set-inclusion semantics of the subtype relationship. Also be aware that structs are subtyped using labels and not by the sequential order of components.

Object Types. We will not give the full the semantics of object types, it is mainly a repetition of the definitions given in (Laasch and Scholl 1993a) for BCOOL. The extensions w.r.t. the type constructors not contained in BCOOL have been given above or will be given together with the operations below. In summary, the semantics of object types is defined such that the following holds:

- the *semantic domains* of all object types are the same, in order to allow for object evolution, such that objects can gain and lose instance relationships dynamically. Formally: $[\![[f_1, ..., f_n]]\!] = [\![[\,]]\!] = \mathcal{D}_{Object}$, for $\{f_1, ..., f_n\} \subseteq F$.
- nonetheless, object types are arranged in a lattice: there is an object type $[f_1, ..., f_n]$ for any subset of F that contains the functions defined in a database schema. Intuitively, applications of the function f on instances of $[f_1, ..., f_n]$ pass static type-checking, iff f is contained in $\{f_1, ..., f_n\}$. In the programming language community this kind of subtyping is called F-bound polymorphism (Canning, Cook, Hill and Olthoff 1989).
- it is not possible to refer to arbitrary instances of object types, rather only to those that are currently part of the active domain (i.e. that have been introduced by explicit creation and have not been deleted since). Therefore, the (semantic) domains of object types (denoted by $[\![[f_1, ..., f_n]]\!]$) have to be distinguished from the *active domains* (denoted by $\sigma([f_1, ..., f_n])$) that contain the instances of these types in the current state σ;
- only the active domains of the type without functions ($[\,]$) and the types with only one function ($[f_i]$) need to be explicitly maintained by an implementation. The other active domains (of types with more functions $[f_i, f_j, \ldots]$) are derived from the former according to the type lattice;
- **Definition Subtyping:**
  $[f_1, ..., f_n] \preceq [f'_1, ..., f'_m] \iff \{f_1, ..., f_n\} \supseteq \{f'_1, ..., f'_m\}$.
- **Definition Object Type Lattice:**
  The set of object types forms a lattice, where the subtype relationship is the partial order. The least upper bound ($\sqcup$), and the greatest lower bound ($\sqcap$) are defined as follows:
  $[f_1, ..., f_n] \sqcup [f'_1, ..., f'_m] = [f''_1, ..., f''_l]$,
  where $\{f''_1, ..., f''_l\} = \{f_1, ..., f_n\} \cap \{f'_1, ..., f'_m\}$.
  $[f_1, ..., f_n] \sqcap [f'_1, ..., f'_m] = [f''_1, ..., f''_l]$,
  where $\{f''_1, ..., f''_l\} = \{f_1, ..., f_n\} \cup \{f'_1, ..., f'_m\}$.
- Notice that the definitions guarantee the subset relationship between the active domains of a subtype and its supertypes, because of the superset relationship between their function sets:

  $[f_1, ..., f_n] \preceq [f'_1, ..., f'_m] \implies \{f_1, ..., f_n\} \supseteq \{f'_1, ..., f'_m\} \implies \sigma([f_1, ..., f_n]) \subseteq \sigma([f'_1, ..., f'_m])$.

Example 2 The object functions present in Figure 1 are *name*, *age*, *children*, *title*, *specialities*, *r_name*, *owner*, *places*, *the_cooks*, *menu*, and *is_made_of*.

The object type [*name*, *specialities*] is not directly present in the schema, but is an element of the CROQUE object lattice. Obviously, [*name, specialities*] is a useful type in the running example, because it is related to names and specialities of the cooks stored in the database. □

Classes. In our framework, several extents can be built for a certain type and are maintained by the system. These can be bound either to a variable or to a "class". For the purpose of this paper, our syntax of defining "classes" is equivalent to the "`with extension <` *class name* `>`" clause in ODL interface (i.e. type) definitions. A class in CROQUE is a named object container (collection) that might be arranged in subcollections-hierarchies with other classes. Using classes allows us to analyze the dependencies between certain instances either according to the type (subtyping) or the extents (subclassing). As in COCOON there exists the possibility that classes are restricted or determined by constraints. We do not work out these topics here but refer to earlier work done for COCOON (Scholl et al. 1993) and in the EXTREM model (Heuer and Sander 1991).

Example 3 The CROQUE database schema for the running example is as follows:

```
type restaurant = interface{r_name: string, owner: person,
                    places: int, the_cooks: set<cook>,
                    menu: list< struct<menu_name: string, price: int,
                    dishes: set<dish>>>};
type person = interface{name: string, age: int,children: set<person>};
type cook = interface:person{title: string, specialities: set<dish>};
type dish = interface{name : string, is_made_of : set<string>};

class Persons : person;
class Cooks : cook some Persons;    /* subset of Persons */
class Businessmen : person some Persons;    /* subset of Persons */
class Restaurants : restaurant;
class Dishes : dish;
```

The database schema shows how the notion of a class can be used in a flexible manner to assign the same type to different object collections. The class *Businessmen* has the same type as *Persons*, but describes a (usually proper) subset of persons. This can be compared with the notion of *named sub-extents* mentioned as a possible future revision of ODMG 1.2. □

4 ASPECTS OF CROQUE-OQL

In this section we show several aspects of CROQUE-OQL to elude the additional possibilities of CROQUE-OQL compared with the ODMG approach. The technical formalization can be found in the Appendix. The CROQUE approach is characterized by the following:

- For each query statement there is a static type check. When a query matches the preconditions, the type check defines the result type of the query. Moreover, each type-correct query succeeds, that is, there are no exceptions raised within our framework.
- All ODMG-OQL queries can also expressed in our framework. If its original formulation in ODMG-OQL does not pass the type-check, it can easily be rewritten into a similar type-safe CROQUE-OQL query.
- Moreover, some query expressions can be applied in our approach but not with ODMG-OQL, because we fully exploit the additional possibilities offered by our type lattice.
- We add some functionality which is not present in ODMG-OQL, but nevertheless is considered useful due to our experience (Laasch and Scholl 1993a, Heuer, Fuchs and Wiebking 1990), namely the manipulation of complex structured attributes within object-preserving queries.

Typed Queries. In contrast to the ODMG proposal we can use the subtype hierarchy within the query formalism. From this point of view, some queries generate subtypes (e.g. `intersect`), while other operations lead to supertypes (e.g. casts, `union`). In the ODMG approach it is rather unsatisfactory how partial states of an object can be accessed, especially when compared with the possibilities of other object query languages and nested relational languages. According to the type system of CROQUE it is possible to formulate additional queries:

- an object can be cast in an object-preserving way to each supertype. Especially, this leads to queries which are not expressible in ODMG-OQL. For instance, the query

  `([`*name*`,`*specialities*`])` *Cooks*

  is not valid in ODMG-OQL, because the result type of the query is not present in the ODL schema, but well-defined in the object type lattice of the CROQUE model and retrieves each cook with his name and his set of specialities.
- immutable objects can be projected to any supertype.

Example 4 As shown in Example 1, the immutable object type

Q		
r_name	*places*	*menu:*$\langle$(*menu_name*, *price*)$\rangle$
Theo's	32	$\langle(Standard, 21), (Best, 29)\rangle$
Villey's	20	$\langle(Breakfast, 10), (Breakfast, 11)\rangle$

The result of Example 2	
r_name	*menu:*$\langle$(*menu_name*)$\rangle$
Theo's	$\langle(Standard), (Best)\rangle$
Villey's	$\langle(Breakfast), (Breakfast)\rangle$

Figure 2 Extension and result for Example 4.

$T \equiv$ `struct<`*r_name* : `string`, *menu* : `list<struct<`*menu_name* : `string>>>`

is a supertype of

$S \equiv$ `struct<`*r_name* : `string`, *places* : `int`, *menu* : `list<struct<`*menu_name* : `string`, *price* : `int>>>`.

Let Q be the extension of S as given in Figure 2. Then the following query is possible:

(`struct<`*r_name* : `string`,
menu : `list<struct<`*menu_name* : `string>>>`) Q,

which "projects" Q onto the type T. It should be noted that the query cannot directly be realized with an OQL `select-from-where` statement, because the construction of list-type results via `select-from-where` statements is too restricted. □

Set operations. The ODMG approach handles only rather simple cases of the collection operations `union`, `intersect`, and `except`, where the operands have the *same* type. Instead, the type system of the CROQUE model allows further, well-defined set operations.

In CROQUE, each collection of objects consists either solely of mutable or solely of immutable objects. Thus, it is not possible to build a collection which consists of both mutable and immutable objects. The set operations `union`, `intersect`, and `except` can be applied to sets and bags in our approach. If both operands are set types, the result is a set, otherwise it is a bag. For

bags we use the addition (resp. minimum) of the cardinalities to determine the semantics of a `union` (resp. `intersect`). The typing rules are:

- for mutable objects: object types are always compatible. The result type depends on the set operation and is given by the corresponding (type) lattice operations:

  - `union`: $collection(\tau_1 \sqcup \tau_2)$,
  - `intersect`: $collection(\tau_1 \sqcap \tau_2)$,
  - `except`: $collection(\tau_1)$.

- for immutable objects: The typing rules for immutable objects as given in the last section do not imply a lattice structure. So set operations are only allowed when the common subtype of the element types (for the `intersect` operation), or the common supertype of the element types (for `union` and `except`) exist. Then the result consists of

  - `union`: elements of the input instances casted to the common supertype,
  - `intersect` all elements of the domain of the common subtype which can be casted to an element of both input instances,*
  - `except`: all elements of the first collection which do not have a corresponding element in the second collection when both are casted to the common supertype.

Example 5 Two instances of mutable objects can always be combined, e.g. the query

Businessmen `intersect` *Cooks*

returns all businessmen who are also cooks. The result type is

$person \sqcap cook = [name, age, children, title, specialities]$,

while the query

Businessmen `union` *Cooks*

returns anyone who is either a businessman or a cook. The result type is $person \sqcup cook = person = [name, age, children]$. The query *Businessmen* `except` *Cooks* returns all businessmen who are not cooks. The return type is *person*. □

*This is in fact an extension of the definition of a natural join.

Q_1	
name	*restaurants*:{(*r_name*)}
Theo	{(Theo's),(Starlight Cafe)}
Mary	{}
Peter	{(Peter's Inn)}

Q_2		
name	*restaurants:*{(*r_name*, *places*)}	*age*
Theo	{(Theo's,32),(Starlight Cafe,15)}	37
Mary	{(Mary's Inn,25),(Villey's,20)}	38

Figure 3 Extensions for Example 6.

Example 6 Figure 3 shows two collections of literals, Q_1 and Q_2, which contain certain information about persons and the restaurants they own.* The type of Q_2 is

```
set<struct<name: string,
           restaurants: set<struct<r_name: string, places: int>>
           age: int>>
```

and the type of Q_1 is a supertype of the type of Q_2. So the set operations combining Q_1 and Q_2 are well-defined, because in this special case the common subtype (supertype) is the type of Q_2 (of Q_1). The results are given in Figure 4 and are constructed as follows

- Q_1 `union` Q_2: The result type is the type of Q_1. The result consist of the tuples which are either in Q_1 or a cast of tuples of Q_2 onto the type of Q_1. It should be mentioned that

  - duplicates are removed, because the input instances are both sets,
  - the construction of result tuples is only based on the recursive cast operation, which can be seen as an extension of a projection in the nested algebra. There is no unification of "similar" structured values as in the PNF nested algebra (Roth, Korth and Silberschatz 1988).

- Q_1 `intersect` Q_2: Here, the result type is the maximal common subtype,

*Both collections can be seen as special views (using immutable objects) of the database schema of Figure 1 realized by OQL-queries.

Q_1 `union` Q_2	
name	*restaurants*: {(*r_name*)}
Theo	{(Theo's),(Starlight Cafe)}
Mary	{(Mary's Inn), (Villey's)}
Mary	{}
Peter	{(Peter's Inn)}

Q_1 `intersect` Q_2		
name	*restaurants*:{(*r_name*,*places*)}	*age*
Theo	{(Theo's,32),(Starlight Cafe,15)}	37

Q_1 `except` Q_2	
name	*restaurants*:{(*r_name*)}
Mary	{}
Peter	{(Peter's Inn)}

Q_2 `except` Q_1		
name	*restaurants:*{(*r_name*, *places*)}	*age*
Mary	{(Mary's Inn,25),(Villey's,20)}	38

Figure 4 Results for Example 6.

i.e. the type of Q_2. The result tuples have to be present in Q_2 and there must be corresponding tuples for the common supertype in Q_1.

- Q_1 `except` Q_2: The result type is the type of Q_1, because it is the type of the first operand. For each tuple of Q_1, it will be checked whether there is a corresponding value in Q_2 using the common supertype, in this case Q_1. □

We mention that the existence of the common subtype also implies the existence of the common supertype, but not vice versa, as the following example shows:

Example 7 The types

T_1 : `struct<`A : `string,`B : `set<string>>` and
T_2 : `struct<`A : `string,`B : `bag<string>>`

have the common supertype `struct<`A : `string>`, but there is no common subtype because the component B is present in both types, but its different typings are not comparable by the subtype inference rules. □

The problem of the last example arises, because subtyping of structs is based on the labels in the CROQUE model. So this problem can be avoided by the additional restriction that all components of structs which are not part of the common supertype must have pairwise distinct labels.

Exceptions versus nulls. ODMG-OQL uses exceptions at run-time in order to handle some queries which are not "well-behaved". These cases are treated differently within our approach, because we usually reject such queries due to type-checking errors.

Example 8 ODMG-OQL 1.2 allows the following query

```
select (Cook) p.specialities
from Persons as p
where condition
```

If the specified *condition* restricts the set of persons to a subset of cooks, this query works well in the ODMG approach, otherwise a run-time exception occurs. Notice that determining whether *condition* returns only cooks is not possible at compile-time in general. Consequently, this query is not allowed in the CROQUE approach, because the cast onto a subtype is not type-safe. Nevertheless, a similar query can be realized in CROQUE-OQL in different ways, e.g.:

```
select c.specialities
from Cooks as c
where condition
```

Another possibility is the use of type guards as described in (Laasch and Scholl 1993a). □

The only case, where it is not possible to avoid a conflict, is the element operator applied to a non-singleton collection.* In this case, CROQUE-OQL works as follows:

- the specified collection is empty: The null value $\perp_T$ of the underlying domain is chosen;

*The problem might be considered more of theoretical nature, since in practice, the `element` operator will usually be applied to a singleton.

- the specified collection has more than one element: a value is taken randomly by non-determinism.

Nulls in queries. Although there is a specific $\perp_T$ for each type T in the ODMG model as well as in the CROQUE approach, it is rather unclear how this value will be handled in OQL queries. CROQUE uses the following approach:

- $\perp_T$ is not treated in a special way, e.g. in our approach there is no semi-lattice of values using $\perp_T$ as the bottom element.
- In boolean conditions $\perp_{Bool}$ is treated similar to `false` meaning that only the values are selected where the condition evaluates to `true`.

As in other frameworks, this approach to nulls is not very satisfying, because its application-dependent semantics is rather difficult to capture. But due to a lot of detail problems we leave a more elaborate solution for future work.

Extend. Building new subtypes of object types is not directly possible in the ODMG approach. In CROQUE-OQL, we provide an `extend` function for such queries. For instance,

```
extend[#r:count(
                    select * from Restaurants r
                    where r.owner = b)
      ](Businessmen as b)
```

assigns the type $[name, age, children, \#r]$ to the collection of businessmen, where $\#r$ holds the number of restaurants owned by each businessman.

5 CONCLUSION AND FURTHER RESEARCH

The proposed ODMG standard for object databases has proven very useful in unifying the attempts to define object database models and query languages. The commercial market as well as the research community adopt the standard in one way or the other. However, one of the major deficiencies w.r.t. research activities around OQL is the lack of a formal definition. The standard as described in (Cattell 1996) is in fact more a syntactical skeleton than a sound definition. In this paper we have presented the CROQUE approach to define rigid formal semantics for OQL 1.2. We have argued that in some respects certain deviations from the standard are adequate: to name one, we have simplified the type system (or you could equivalently say, the meta-schema) by avoiding the duplication of mutable (object) and immutable (value) structures. For us, (mutable) objects are atomic. ODMG's structured

mutable objects will be modeled as atomic objects with a structure-valued property (attribute) in CROQUE.

Another characteristic of our proposal is the strong and static type system that we introduced for ODL. The type system provides fully orthogonal type constructors (sets, lists, bags, arrays, tuples), arranges object types in a complete type lattice (that allows for exact typing of more flexible collection operations, such as unions, differences, and intersections), and separates type ("interface") definitions from extent definitions (called class definitions in CROQUE). This, among others, does allow for multiple type extents that are mentiond as "Possible Future Revisions" in the ODMG proposal.

The main advantage of our clean and powerful type system is its applicability for the formalization of the query language OQL. In our approach, only type-safe queries are considered any further. This can be checked at compile-time, using the subtyping rules presented in this paper. So we need not bother with run-time exception in our approach. A second result is the more elaborate and flexible use of cast (i.e., type changing) and set operations. Every object or structured value can be cast to any supertype. Set operations are applicable for all collections of objects, because the result type can always be constructed by the corresponding (up- or down-) cast operations. Additionally, set operations on collections of differently structured values are applicable, if the corresponding types are comparable by the use of the subtyping rules.

The work presented in this paper is part of the CROQUE project, where we implement a prototype OODBMS with particular focus on query optimization, physical database design, and the maintenance of partially replicated storage structures. CROQUE will offer (our variant of) an ODMG user interface; it will provide a wide variety of design choices to the database administrator for selecting internal storage structures, particularly including replication schemes; and finally, it will contain a powerful cost- and rule-based query optimizer that generate efficient execution code. The work presented in this paper represents the front-end part of CROQUE: OQL queries (and updates in an OML, an object manipulation language) are translated into an internal representation for further manipulation. In order to clearly define our starting point, we have given a formal semantics to OQL version 1.2. This work, even though related to CROQUE is also useful for others, since the standard document itself does not provide enough details on the intended semantics.

Currently, ODMG standard 2.0 has been released with some changes to the object model and the query language. While we still have to work out the changes in detail, we already made the following preliminary observations:

- ODMG's object model has been extended by adding a "class" concept (as instantiable object types) changing the policy about instantiations and extents. Presumably we will integrate the consequences for several extents using our class concept which describes the extent of a type and not its behavior.

- ODMG proposes a new type constructor "dictionary" which can be added to our data model without difficulties.
- ODMG proposes a new approach for null values. For each domain (without a null value) there exists another domain with a type-specific null value. In our approach, we already have a type-specific null value in every domain, because this is necessary in our formal approach.
- Querying nulls in ODMG-OQL looks now more similar to SQL2, although it is restricted to a two-valued logic (like in our approach). Here we think a more elaborate approach using three-valued logic has to be worked out in the future, also looking into SQL3.
- Now ODMG-OQL allows some enhanced set operations where always the common super type of the instances is used in the result. While this solution that always upcasts (e.g. intersections, too) does not return a "wrong" result type, it eliminates a lot of potentially useful properties/methods from the result. Hence, we will probably stick with our the more flexible lattice-based approach.

Further work in the CROQUE context is primarily concerned with optimization and implementation (a first prototype has been demonstrated already, parts of it are available online on the Web). For the parts presented in this paper, a suite of generic update operations including a bulk update facility in the style of our earlier work (Laasch and Scholl 1992, Laasch and Scholl 1993b) are next to come, so as to offer a basis for the specification of declarative update transactions. This will constitute the CROQUE-OML. Furthermore, some details that have already been mentioned throughout the paper need further elaboration, for example the complete treatment of null values.

Acknowledgements. We like to thank Dieter Gluche, Torsten Grust, Andreas Heuer, Joachim Kröger, and Andreas Henrich for intensive discussions and the anonymous referees for useful comments.

REFERENCES

Abiteboul, S. and Kanellakis, P.: 1989, Object identity as a query language primitive, *Proc. ACM SIGMOD Conference on Management of Data*.

Abiteboul, S. and Kanellakis, P.: 1990, *Intl. Conf. on Database Theory*, number 470 in *Lecture Notes in Computer Science*, Springer.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S.: 1989, The object-oriented database system Manifesto, *(Kim, Nicolas and Nishio 1989)*, pp. 40–57.

Balsters, H. and de Vreeze, C. C.: 1991, A Semantics of Object-Oriented Sets, *Intl. Workshop on Database Programming Languages*, Nafplion, Greece, pp. 187–200.

Balsters, H. and Fokkinga, M. M.: 1991, Subtyping can have simple semantics, *Theoretical Computer Science* **87**, 81–96.

Bancilhon, F. and Khoshafian, S.: 1989, A Calculus for Complex Objects, *Journal of Computer and System Sciences* **38**, 326–340.

Beeri, C.: 1989, Formal Models for Object-Oriented Databases, *(Kim et al. 1989)*, pp. 370–395.

Beeri, C.: 1990, A Formal Approach to Object-Oriented Databases, *Data and Knowledge Engineering* pp. 353–382.

Bertino, E., Negri, M., Pelagatti, G. and Sbattella, L.: 1992, Object-Oriented Query Languages: The Notion and the Issues, *IEEE Transactions on Knowledge and Data Engineering* **4**(3), 223–237.

Biskup, J.: 1983, A Foundation of Codd's Relational Maybe-operations, *ACM Transactions of Database Systems* **8**(4), 608–636.

Canning, P., Cook, W., Hill, W. and Olthoff, W.: 1989, F-Bounded Polymorphism for Object-Oriented Programming, *ACM Intl. Conf. On Functional Programming and Computer Architecture*.

Cattell, R. (ed.): 1996, *The Object Database Standard: ODMG-93, Release 1.2*, Morgan Kaufmann.

Ceri, S. and Manthey, R.: 1993, Chimera: A model and language for active dood systems, *Proc. of the 2nd East-West Database Workshop*, Workshops in Computing, Springer, pp. 3–16.

Fegaras, L. and Maier, D.: 1995, Towards an Effective Calculus for Object Query Languages, *Proc. ACM SIGMOD Conference on Management of Data*, pp. 47–58.

Grust, T., Kröger, J., Gluche, D., Heuer, A. and Scholl, M. H.: 1997, Query Evaluation in CROQUE – Calculus and Algebra Coincide, *British National Conference on Databases (BNCOD)*, pp. 84–100.

Grust, T. and Scholl, M.: 1996, Translating OQL into Monoid Comprehensions — Stuck with Nested Loops?, *Technical Report 3a/96*, Dept. of Mathematics and Computer Science, University of Konstanz.

Herzig, R. and Gogolla, M.: 1994, A SQL-like Query Calculus for Object-Oriented Database Systems, *Intl. Symposium on Object-Oriented Methodologies and Systems (ISOOMS)*, number 858 in *Lecture Notes in Computer Science*, Springer, pp. 20–39.

Heuer, A., Fuchs, J. and Wiebking, U.: 1990, OSCAR: An Object-Oriented Database System with a Nested Relational Kernel, *Proc. of the Intl. Conf. on Entity-Relationship Approach*, Elsevier (North-Holland).

Heuer, A. and Sander, P.: 1991, Classifying object-oriented query results in a class/type lattice, *Proc. of the 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems (MFDBS)*, Vol. 495 of *Lecture Notes in Computer Science*, Springer, pp. 14–28.

Heuer, A. and Scholl, M.: 1991, Principles of Object-Oriented Query Languages, *GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft (BTW)*, Springer, pp. 178–191.

Hohenstein, U. and Engels, G.: 1992, SQL/EER - Syntax and Semantics of an Entity-Relationship-Based Query Language, *Information Systems* **17**(3), 209–242.

Hörner, C. and Heuer, A.: 1991, *EXTREM – The structural part of an object-oriented database model*, Report 91/5 of the Department of Computer Science at the Technical University of Clausthal, Germany.

Kamel, N., Wu, P. and Su, Y.: 1994, A Pattern-Based Object Calculus, *VLDB Journal* **3**(1), 53–76.

Kifer, M., Kim, W. and Sagiv, Y.: 1992, Querying Object-Oriented Databases, *Proc. ACM SIGMOD Conference on Management of Data*, pp. 393–402.

Kifer, M., Lausen, G. and Wu, J.: 1993, *Logical Foundations of Object-Oriented and Frame-Based Languages*, Technical Report, 93/06, Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794.

Kim, W., Nicolas, J.-M. and Nishio, S. (eds): 1989, *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases*, Elsevier.

Kuno, H. and Rundensteiner, E.: 1996, The MultiView OODB view system: Design and Implementation, *Theory and Practice of Object Systems (TAPOS)* **2**(3), 202–225.

Laasch, C. and Scholl, M.: 1993a, A Functional Object Database Language, *Intl. Workshop on Database Programming Languages.*

Laasch, C. and Scholl, M. H.: 1992, Generic update operations keeping object-oriented databases consistent, *Proc. 2nd GI-Workshop on Information Systems and Artificial Intelligence*, Springer IFB 303, FAW Ulm, Germany, pp. 40–55.

Laasch, C. and Scholl, M. H.: 1993b, Deterministic Semantics of Set-Oriented Update Sequences, *Proc. of the IEEE Intl. Conf. on Data Engineering*, pp. 4–13.

Leung, T., Mitchell, G., Subramanian, B., Zdonik, S. and other: 1993, The AQUA Data Model and Algebra, *in* C. Beeri, A. Ohori and D. Shasha (eds), *Intl. Workshop on Database Programming Languages*, Springer, Workshops in Computing, pp. 136–156.

Libkin, L., Machlin, R. and Wong, L.: 1996, A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques, *Proc. ACM SIGMOD Conference on Management of Data*, pp. 228–239.

Mannino, M., Choi, I. and Batory, D.: 1990, The Object-Oriented Functional Data Language, *IEEE Transactions on Software Engineering* **16**(11), 1258–1272.

Melton, J.: 1996, An SQL3 snapshot, *Proc. of the IEEE Intl. Conf. on Data Engineering*, pp. 666–672.

Riedel, H. and Scholl, M.: 1996, *The CROQUE-Model: Formalization of the*

Data Model and Query Language, Technical Report 23/96, Dept. of Mathematics and Computer Science, University of Konstanz.

Roth, M., Korth, H. and Silberschatz, A.: 1988, Extended Algebra and Calculus for Nested Relational Databases, *ACM Transactions on Database Systems* **13**(4), 389–417.

Schek, H.-J. and Scholl, M.: 1990, A Relational Object Model, *(Abiteboul and Kanellakis 1990)*, pp. 89–105.

Scholl, M., Laasch, C., Rich, C., Schek, H.-J. and Tresch, M.: 1993, *The COCOON Object Model*, Technical Report 93-02, Dept. of Computer Science, University of Ulm.

Straube, D.: 1991, *Query Processing in Object-Oriented Database Systems*, PhD Thesis, University of Alberta, Edmonton, Canada.

Straube, D. and Özsu, T.: 1990, Queries and Query Processing in Object-Oriented Database Systems, *ACM Transactions on Office Information Systems* **8**(4), 387–430.

Yu, L. and Osborn, S.: 1991, An Evaluation Framework for Algebraic Object-Oriented Query Models, *Proc. of the IEEE Intl. Conf. on Data Engineering* pp. 670–677.

APPENDIX 1 FORMAL SEMANTICS OF SELECTED QUERY OPERATIONS

In this section we introduce the formal semantics of casting and set operations by giving

1. preconditions of the applicability,
2. the syntax,
3. the result type of the query expression using the $\frac{premise}{implication}$ notation: given the types in *premise*, we logically infer the result type in the *implication* part. The notation $query :: type$ means that the query *query* has the type *type*.
4. the result of the query, i.e. its semantics.

The interested reader can find a full definition of CROQUE-OQL in the same style in (Riedel and Scholl 1996). Some basic operations on datatypes are used as usual (without further formalization):

- for bags: Let I be an instance of $\{\!\{\ T\ \}\!\}$ and $e \in [\![\ T\ ]\!]$. The function $card(e, I) \ \mapsto \delta_{Int}$ determines how many times e is in I.
- for lists and arrays: $I[k]$ returns the k-th value of I. If k is not a valid index for I, $\perp_T$ will be returned.

Explaining the typing rules, it is sometimes necessary to refer to parts of a type structure in an abstract way. So we use the notation $\tau(\tau_1)$ where τ_1 refers to the direct sub-component of τ. For example, let $\tau = \{\!\{\langle int \rangle\}\!\}$. By the use of $\tau(\tau_1)$ the type expression τ_1 is bound to $\langle int \rangle$. We assume the existence of a set $\mathcal{LABELS}$ holding the possible names used for labels of classes, types, functions, or components of structs.

The formalization is as follows:

- *Casts:* An object or value can be casted to each supertype. The new instance is built similar to a recursive projection. The most significant changes occur in the case of structs or object types, where some components are not part of the result. In the case of sets, duplicates are removed as usual.

1. *precondition*: τ_1 is a subtype of τ_2.
2. *syntax*: $(\tau_2)\ I$
3. *typing rule*: $\frac{I::\tau_1, \tau_1 \preceq \tau_2}{(\tau_2)\ I::\tau_2}$
4. *instance*: $[\![\ (\tau_2)\ I\ ]\!] =$

type of τ_2	*result*	*further conditions*
basic type	v	$: [\![I]\!] = v$
struct	$(L_1 : (\tau_{2,1})\ c_1, \ldots$	$: [\![I]\!] = (L_1 : c_1, \ldots, L_m : c_m,$
	$, L_m : (\tau_{2,m})\ c_m)$	$\ldots, L_n : c_n)\ \wedge\ \tau_2 =$
		$(L_1 : \tau_{2,1}, \ldots, L_m : \tau_{2,m})$
set	$\{t' \mid \exists t \in [\![I]\!] \wedge (\tau_2')\ t = t'\}$	$: \tau_2 = \{\tau_2'\}$
bag	$card(t', [\![(\tau_2)\ I]\!]) =$	
	$\sum_{(\tau_2')t=t'}(card(t, [\![I]\!])$	$: \tau_2 = \{\!\{\tau_2'\}\!\} \wedge t' \in [\![\ \{\!\{\tau_2'\}\!\}\ ]\!]$
list	$\langle (\tau_2')\ c_0, \ldots, (\tau_2')\ c_n \rangle$	$: [\![I]\!] = \langle c_0, \ldots, c_n \rangle \wedge \tau_2 = \langle \tau_2' \rangle$
array	$[(\tau_2')\ c_0, \ldots, (\tau_2')\ c_n]$	$: [\![I]\!] = [c_0, \ldots, c_n] \wedge \tau_2 = [\tau_2']$
function	$o \to (\tau_2')\ v$	$: [\![I]\!] = o \to v \wedge$
		$\tau_2 = \mathcal{D}_{Object} \to_{fin} \tau_2'$
object type	$\sigma([f_1, \ldots, f_m, \ldots, f_n])$	$: [\![I]\!] = \sigma([f_1, \ldots, f_m, \ldots, f_n]) \wedge$
		$\tau_2 = [f_1, \ldots, f_m]$

- "set operations": The typing rules are similar for mutable and immutable objects. The result type always exists for collections of mutable objects, but not for immutable objects. So set operations are only applicable to immutable objects iff the common supertype (`union`, `except`) or subtype (`intersect`) exist.

mutable objects:

intersect:

1. *precondition*: $\tau_1(\tau_1')$ and $\tau_2(\tau_2')$ are set or bag types. If τ_1 and τ_2 are both sets, then $\tau_3 = \{\tau_1' \sqcap \tau_2'\}$, otherwise $\tau_3 = \{\!\{\tau_1' \sqcap \tau_2'\}\!\}$.
2. *syntax*: $query_1$ `intersect` $query_2$
3. *typing rule*: $\frac{query_1::\tau_1(\tau_1'),\ query_2::\tau_2(\tau_2')}{query_1 \texttt{ intersect } query_2::\tau_3}$
4. *instance*:
$$t \in [\![\, query_1 \texttt{ intersect } query_2 \,]\!] :\Leftrightarrow\ t \in [\![query_1\,]\!] \wedge\ t \in [\![query_2]\!],$$
$$card(t, [\![\, query_1 \texttt{ intersect } query_2 \,]\!] = min(card(t, [\![query_1\,]\!]), card(t, [\![query_2]\!]))$$

union:

1. *precondition*: $\tau_1(\tau_1')$ and $\tau_2(\tau_2')$ are set or bag types. If τ_1 and τ_2 are both sets, then $\tau_3 = \{\tau_1' \sqcup \tau_2'\}$, otherwise $\tau_3 = \{\!\{\tau_1' \sqcup \tau_2'\}\!\}$.
2. *syntax*: $query_1$ `union` $query_2$
3. *typing rule*: $\frac{query_1::\tau_1(\tau_1'),\ query_2::\tau_2(\tau_2')}{query_1 \texttt{ union } query_2::\tau_3}$
4. *instance*:
$$t \in [\![\, query_1 \texttt{ union } query_2 \,]\!] :\Leftrightarrow\ t \in [\![query_1\,]\!] \vee\ t \in [\![query_2]\!],$$
$$card(t, [\![\, query_1 \texttt{ union } query_2 \,]\!] = card(t, [\![query_1\,]\!]) + card(t, [\![query_2]\!])$$

except:

1. *precondition*: $\tau_1(\tau_1')$ and $\tau_2(\tau_2')$ are set or bag types. If τ_1 and τ_2 are both sets, then $\tau_3 = \{\tau_1'\}$, otherwise $\tau_3 = \{\!\{\tau_1'\}\!\}$.
2. *syntax*: $query_1$ `except` $query_2$
3. *typing rule*: $\frac{query_1::\tau_1(\tau_1'),\ query_2::\tau_2(\tau_2')}{query_1 \texttt{ except } query_2::\tau_3}$
4. *instance*:
$$t \in [\![\, query_1 \texttt{ except } query_2 \,]\!] :\Leftrightarrow$$
$$t \in [\![query_1\,]\!] \wedge \neg\ (t \in [\![query_2]\!]),$$
$$card(t, [\![\, query_1 \texttt{ except } query_2 \,]\!] = max(card(t, [\![query_1\,]\!]) - card(t, [\![query_2]\!]), 0)$$

immutable objects:

intersect:

1. *precondition*: $\tau_1(\tau_1')$ and $\tau_2(\tau_2')$ are set or bag types. The maximal common subtype of τ_1' and τ_2' exists and is named τ_3'. If τ_1 and τ_2 are both sets, then $\tau_3 = \{\tau_3'\}$, otherwise $\tau_3 = \{\!\{\tau_3'\}\!\}$.
2. *syntax*: $query_1$ `intersect` $query_2$,
3. *typing rules*: $\frac{query_1 :: \tau_1(\tau_1'),\ query_2 :: \tau_2(\tau_2')}{query_1 \texttt{ intersect } query_2 :: \tau_3}$
4. *instance*:

$$t \in [\![query_1 \texttt{ intersect } query_2]\!] :\Leftrightarrow t \in [\![\tau_3']\!] \wedge (\tau_1')\ t \in [\![query_1]\!] \wedge (\tau_2)\ t \in [\![query_2]\!]$$

$$card(t, [\![query_1 \texttt{ intersect } query_2]\!]) = min(card((\tau_1')t, [\![query_1]\!]), card((\tau_2')\ t, [\![query_2]\!]))$$

union:

1. *precondition*: $\tau_1(\tau_1')$ and $\tau_2(\tau_2')$ are set or bag types. The least common supertype of τ_1' and τ_2' exists and is named τ_3'. If τ_1 and τ_2 are both sets, then $\tau_3 = \{\tau_3'\}$, otherwise $\tau_3 = \{\!\{\tau_3'\}\!\}$.
2. *syntax*: $query_1$ `union` $query_2$
3. *typing rules*: $\frac{query_1 :: \tau_1(\tau_1'),\ query_2 :: \tau_2(\tau_2')}{query_1 \texttt{ union } query_2 :: \tau_3}$
4. *instance*:

$$t \in [\![query_1 \texttt{ union } query_2]\!] :\Leftrightarrow t \in [\![(\tau_3')\ query_1]\!] \vee t \in [\![(\tau_3') query_2]\!],$$

$$card(t, [\![query_1 \texttt{ union } query_2]\!]) = \sum_{t' \in query_1 \wedge (\tau_3')t' = (\tau_3')\ t} card((t', [\![query_1]\!]) + \sum_{t' \in query_2 \wedge (\tau_3')t' = (\tau_3')\ t} card(t, [\![query_2]\!])$$

except:

1. *precondition*: $\tau_1(\tau_1')$ and $\tau_2(\tau_2')$ are set or bag types. The least common supertype of τ_1' and τ_2' exists and is named τ_4'. If τ_1 and τ_2 are both sets, then $\tau_3 = \{\tau_1'\}$, otherwise $\tau_3 = \{\!\{\tau_1'\}\!\}$.
2. *syntax*: $query_1$ `except` $query_2$
3. *typing rule*: $\frac{query_1 :: \tau_1(\tau_1'),\ query_2 :: \tau_2(\tau_2')}{query_1 \texttt{ except } query_2 :: \tau_3}$
4. *instance*:

$$t \in [\![query_1 \texttt{ except } query_2]\!] :\Leftrightarrow t \in [\![query_1]\!] \wedge \neg(\exists t' \in [\![query_2]\!] : (\tau_4')\ t = (\tau_4')\ t'),$$

$$card(t, [\![query_1 \texttt{ except } query_2]\!]) = max(card(t, [\![query_1]\!]) - \sum_{t' \in query_2 \wedge (\tau_4')\ t' = (\tau_4')\ t} card(t', [\![query_2]\!]), 0)$$