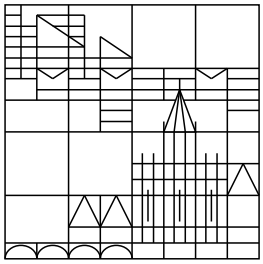

Universität Konstanz

The CROQUE-Model: Formalization of the Data Model and Query Language

Holger Riedel
Marc H. Scholl

Konstanzer Schriften in Mathematik und Informatik
Nr. 23, Dezember 1996

ISSN 1430–3558

Universität Konstanz
Postfach 5560/D 188, 78434 Konstanz, Germany
Email: preprints@informatik.uni–konstanz.de
WWW: http://www.informatik.uni–konstanz.de/Schriften

The CROQUE-Model: Formalization of the Data Model and Query Language

Holger Riedel Marc H. Scholl

23/1996
Universität Konstanz
Fakultät für Mathematik und Informatik *

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 ODMG [Cat96] 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, good models offer nice features such as orthogonal subtyping, the explicit distinction of class hierarchies and type hierarchies, support for values (without object identity), and an orthogonal query language, based on formal (e.g. logic) definitions.

*authors' adress: Fakultät für Mathematik und Informatik, Universität Konstanz, Postfach 5560/D188, D-78434 Konstanz, Germany, email: {Holger.Riedel,Marc.Scholl}@uni-konstanz.de

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 instances ("subclassing") is not possible, because an object formally only belongs to *one* specific class (where it was created). The disadvantages of such an approach have been discussed e.g. in [HS91b, HH91, SLR+93].
  - While the given set of query operations of ODMG-OQL-1.2 seems rather complete for practical reasons, there is a lack of a foundation of such queries, which is essential for query optimization.

- *status of the ODMG proposal:* The ODMG proposal is far away from being an exact definition of an OODM. A lot of detail problems in the ODMG approach – both model and queries – have to be solved. This is actually complicated by the fact that in the future also SQL queries should be captured by ODMG-OQL as well.

- *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 ODMG queries executed efficiently. Many issues are open, especially the interaction of query optimization mechanisms with useful database design choices that need to be offered for the storage of object databases.

In the CROQUE project[1], we are designing and implementing an object database system that realizes a clear separation between logical and physical structures. The logical database schema is defined in CROQUE-ODL, our variant of the ODMG object definition language [Cat96]. In a separate step, the physical database design, the database administrator can use an (estimated or observed) transaction load, i.e., a set of transaction programs with their frequencies to optimize the layout of (internal) storage structures. This optimization process (taking the DB profile and a cost model as additional inputs) should be supported by a tool. Descriptive ODB queries, expressed in CROQUE-OQL/OML – our variant of ODMG-OQL/OML – over the logical ODB schema are transformed and optimized towards efficient internal execution plans by the CROQUE query optimizer. Particular emphasis lies on exploiting physical designs that can differ significantly from the logical schema's view of data: partitioning, clustering, replication of DB objects.

In order to capture ODMG-like queries within our project it was essential that we formally fixed the data model and the queries. Therefore, we tried to extend our preliminary work on the OODB models COCOON [SLR+93] and EXTREM [HH91] with the intentions of [Cat96]. So the CROQUE approach gives clean answers to the following questions:

- The CROQUE data model is based on the ideas of Beeri's OODB model [Bee90] and more directly on EXTREM [HH91] and COCOON [SLR+93]. It can handle values (immutable objects) as well as objects, and orthogonal type constructors (tuples, sets,

[1] 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).

bags, lists, and arrays) are supported. More specifically, a general typing theory 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 checked at compile-time and always terminate with a desired result. The exception handling of the ODMG proposal is replaced by the use of a null value. The typing rules also exclude some ODMG queries in order to get well typed queries. The intention of such queries can easily be achieved using another syntax.

- The ODMG proposal is rather informal for set operations on complex values (structured immutable objects) and object types. Here, CROQUE provides an orthogonal use of union, intersection, and set difference for arbitrary set and bag types.

The rest of this paper is organized as follows: Section 2 presents the formal model (ODL), an overview of CROQUE-OQL is given in Section 3. In Section 4 we examine other approaches which are related to the goals of our framework. The formal semantics is given in the Appendix.

2 The Data Model and ODL

The description and abstract syntax of ODL given in [Cat96] provides only very little detail on the exact semantics of the proposed object model ("formal semantics can easily be defined"). This section presents the CROQUE approach to define such formal semantics. Our formalization builds upon the BCOOL model presented in [LS93]. 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 attribute that in turn contains the (immutable) structure in the CROQUE model.

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 [LS93, SS90]).

Furthermore, we adopt the BCOOL approach to arrange object types into a *lattice*, 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,

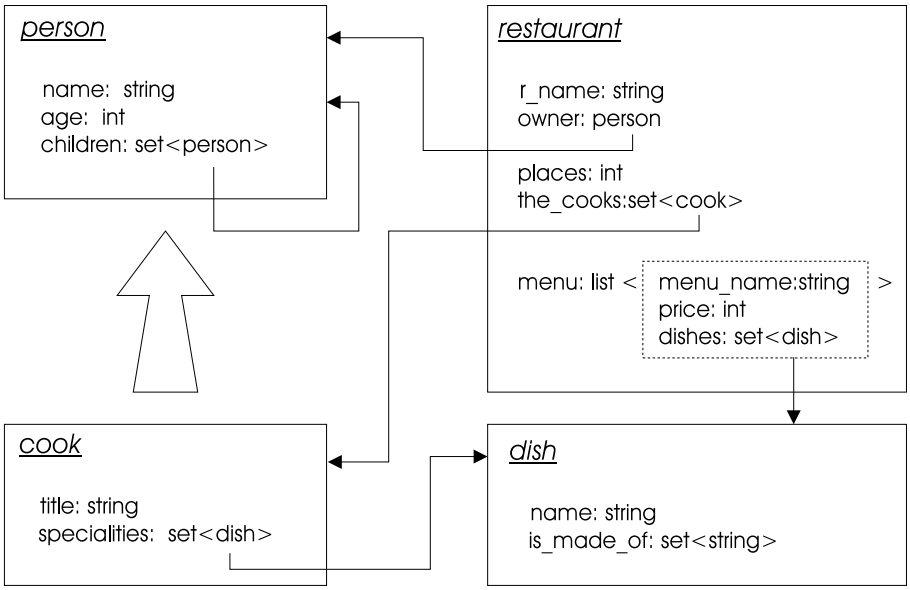

Figure 1: An ODMG schema

operations). ODL characteristics are formalized as functions (attributes are literal-valued functions, relationships are object-valued functions—possibly multi-valued). 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 plus one basic type for mutable objects, 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 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 schema information about instances 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 (here ODMG uses

the somewhat misleading notion "relationship"), while the thick one denotes the subtype relationship. Possible collection types are sets, bags, lists, and arrays.

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.[2] 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 $[name, age, children]$ ($\equiv$ *person*) and $[name, age, children, title, specialities]$ ($\equiv$ *cook*).

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*). □

Syntax. The syntax of formal CROQUE type expressions is given by the following list:

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

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

Semantics. As shown before, 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, while other types are defined by their internal structure. We use a set $\mathcal{LABELS}$ as the name space for attributes and classes.

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,
$\mathcal{F} = \mathcal{D}_{Object} \rightarrow_{fin} \mathcal{V}$,
$\mathcal{S} = P_{fin}(\mathcal{V})$,
$\mathcal{B} = \mathcal{V} \rightarrow_{fin} \{0, 1, 2, \ldots\}$,
$\mathcal{L} = \{0, 1, 2, \ldots\} \rightarrow_{fin} \mathcal{V}$,
$\mathcal{A} = \{0, 1, 2, \ldots\} \rightarrow_{fin} \mathcal{V}$,
$\mathcal{T} = \mathcal{LABELS} \rightarrow_{fin} \mathcal{V}$.

$\mathcal{D}_i$ 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. 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 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 [BdV91, BF91, MCB90]: i.e., the denotations 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 $\mathcal{V}$
$[\![\tau_1 \rightarrow \tau_2]\!] = \{f \in \mathcal{F} \mid x \in [\![\tau_1]\!] \implies f(x) \in [\![\tau_2]\!] \}$
$[\![\{ \tau \}]\!] = \{x \in \mathcal{S} \mid x \subseteq [\![\tau]\!]\}$
$[\![\{\!\{ \tau \}\!\}]\!] = \{f | f : [\![\tau]\!] \rightarrow_{fin} \{0, 1, 2, \ldots\}\}$
$[\![\langle \tau \rangle]\!] = \{f | f : \{0, 1, 2, \ldots\} \rightarrow_{fin} [\![\tau]\!]\}$
$[\![\tau[\cdot]]\!] = \{f | f : \{0, 1, 2, \ldots\} \rightarrow_{fin} [\![\tau]\!]\}$
$[\![(L_1 : \tau_1, \ldots, L_n : \tau_n)]\!] = \{f : \{L_1, L_2, \ldots, L_n\} \rightarrow \bigcup_i [\![\tau_i]\!] | f(L_j) \in [\![\tau_j]\!] \ (j = 1, \ldots, n)\}$

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:[3]

Definition Subtyping:

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

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

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

Object Types. We will not give the full the semantics of object types, it is mainly a repetition of the definitions given in [LS93] 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 $\{f_1, ..., f_n\}$;
- it is not possible to refer to arbitrary instances of object types, rather only to those that are currently part of the active domain. Therefore, the 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 ($[f_i]$) are explicitly maintained. 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 $\{f_1'', ..., f_l''\} = \{f_1, ..., f_n\} \cup \{f_1', ..., f_m'\}$.

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

- 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 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 instances can be built for a certain type. These can be bound either to a variable or to a class. Using classes allows us to analyze the dependencies between certain instances either according to the type (subtyping) or the instance sets (subclassing). As in COCOON there exists the possibility that the instances of classes are restricted or determined by constraints. We do not work out these topics here but refer to earlier work done for COCOON [SLR+93] and in the EXTREM model [HS91a].

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;
class Businessmen : person some Persons;
class Restaurants : restaurant;
class Dishes : dish;
```

It should be noticed that type definitions are completely separated from the class definitions. That means that classes cannot be used within type expressions. As an example, the *owner* of an restaurant is of type *person*, but there is no notion that he/she should belong to the class *Businessmen*. This seems restrictive, but has the advantage that substitutibility of subtypes can always be supported. Also, using classes in type definitions would preclude static type checking. Additionally, the class *Businessmen* is an example of a subclass where the type is the same as in its direct superclass *Persons*. □

3 Some Facets of CROQUE-OQL

In the CROQUE approach, queries can be formulated using the ODMG-OQL language. For our purposes it was necessary to give an formal semantics to all clauses. Although ODMG

[Cat96] claims that *the semantics of (ODMG-)OQL can easily be defined*, we had to tackle a lot of detail problems to get it right. In this section we show several aspects of OQL which seem to be interesting, because they show the additional possibilities of CROQUE-OQL compared with the ODMG approach, while the technical formalization can be found in the appendix.

Typed Queries. In contrast to the ODMG proposal we can use the subtype hierarchy within in 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 unsatisfying how partial states of an object can be accessed, especially when compared with the possibilities of other object query languages [HS91b, LS93] and nested relational languages. According to the type system of CROQUE it is possible to formulate additional queries:

- due to the object type lattice, an object can be casted in an object-preserving way to each subtype in a direct way. 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 ODMG framework, 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

$T \equiv$ `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 instance of S as given in Figure 2. Then the following query is possible:

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

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

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: Instance and result for Example 4.

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 the CROQUE approach 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 immutable objects: A `union` or an `intersect` operation can be applied if the element types of the operands are in a subtype relationship, $\tau_2 \preceq \tau_1$. The result type of a `union` is $collection(\tau_1)$, while an `intersect` results in a $collection(\tau_2)$ type. The `except` operation, Q_1 `except` Q_2, results in literals of type of Q_1.
- for mutable objects: object types are always compatible. The result type depends on the set operation and is given by
  - `union`: $collection(\tau_1 \sqcup \tau_2)$,
  - `intersect`: $collection(\tau_1 \sqcap \tau_2)$,
  - `except`: $collection(\tau_1)$.

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*

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

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)}

Figure 3: Instances and results for Example 6.

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*. □

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.[4] 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 following operations are well-defined: Q_1 `union` Q_2, Q_1 `intersect` Q_2, and Q_1 `except` Q_2. □

Exceptions versus nulls. ODMG-OQL uses exceptions at run-time in order to handle some queries which do not behave senseful. These cases are treated differently within our approach, because we reject such queries or give an explicit semantics.

Example 7 ODMG 1.2 allows the following query

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

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

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. This query is not allowed in the CROQUE approach, because the cast into 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 (Persons intersect Cooks) as c
where condition
```

□

Sometimes it is not possible to avoid a conflict. The `element` operator picks one element of a collection. This is only well-defined when the specified collection has only one element. In the other cases CROQUE-OQL works as follows:

- the specified collection is empty: The null value $\perp_T$ of the underlying domain is chosen;
- the specified collection has more than one element: a value is taken randomly.

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 has to be handled in queries. Here 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 approaches (e.g. SQL), 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 elaborated solution for future work.

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

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

associates the collection of businessmen with the type $[name, age, children, \#business]$, where $\#business$ holds the number of restaurants owned by this businessman.

SQL compatibility. ODMG-1.2 postulates that SQL queries are part of the ODMG-OQL framework. Therefore, the `select-from-where` block has to be considered especially. [5] The following cases have to be considered in greater detail:

[5] In former versions of ODMG-OQL, the formalization of `select-from-where` queries was simplified by the fact that aggregates und grouping were treated as indepedent operators.

- *wildcards in the select clause*: The asterisk * can be used either globally or bound to a variable of the `from` clause. The semantics is always given by a replacement of the asterisk by all available components resp. attributes.

  Example 8 The query

  `select * from` *Cooks c*

  has the result type $\{\!\{(name : \texttt{string}, age : \texttt{int}, children : \{person\}, title : \texttt{string}, specialities : \{dish\})\}\!\}$. The query returns a bag of structs containing the information of the *Cook* class. Equivalent notations are

  `select` *c*`.*` `from` *Cooks c*`,` or
  `select` *c.name*`,` *c.age*`,` *c.children*`,` *c.title*`,` *c.specialities* `from` *Cooks c*

  □

- *renamings*: Functions and components of a type can be renamed in the `select` clause using the notions of SQL. Any renamings which lead to name conflicts are regarded as an type error and are not allowed in CROQUE-OQL. Especially all non-specified renamings due to inexact specification in the `select` clause are not accepted by the CROQUE type checker.

  Example 9 The query `select * from` *Cooks*`,` *Dishes* is not valid, because *name* is used twice in the result. □

- *path expressions in the from clause:* ODMG-OQL allows the access of nested structures by nested use of free identifiers in the `from` clause. This is well-captured in CROOQUE approach.

  Example 10 The query `select` *c.age* `from` *Persons* `as` *p*`,` *p.children* `as` *c* has the type $\{\!\{\texttt{int}\}\!\}$. □

4 Related Work

In the past, many efforts were done to get clean formalizations of OO data models and appropiate query languages. Usually, these frameworks do not intend to give a theoretical foundation for the ODMG approach, but are based on well-defined algebraic or calculus-based approaches. Only recently, some work has been done to apply these results for ODMG-like queries. Here we give a short overview about some of the important approaches related to the goals of this paper.

- *object algebras:* Most of the work on OODB queries has been done in algebraic approaches, e.g.: [SZ89, LS93, SÖ95]. Each of the frameworks defines a specific object algebra capturing some of the reasonable queries. Among these, two categories can be classified: (i) the simple ones only have restricted access and construction capabilities, e.g. no explicit construction of complex values. Such algebras usually offer nice

algebraic properties and potential for optimization, but due to their simplicity they fail to be a base for ODMG-OQL queries. (ii) object algebras using operations as in the nested relational algebra are more adequate for ODMG queries, but it is rather intractable to capture their formalization. A simple object algebra is given by Straube and Özsu[Str91, SÖ95] which works on a type hierarchy. Properties can be simple or set-valued and are modelled as method calls and cannot directly be accessed or constructed. This object algebra makes direct use of "oid-streams". For this framework properties of the object algebra and the generation of different processing plans are studied in depth. The COOL algebra of the COCOON project has a different flavor. It supports the class and type hierarchy and allows set-oriented queries on complex types supporting some operations of the nested relational algebra. Because COCOON has a clean formalization, it was one starting point for our work on CROQUE-OQL, which is now superior to COCOON-COOL in the following sense: (i) CRQOUE-OQL supports values and objects orthogonally; (ii) CROQUE-OQL allows flexible accesses and constructions of complex structured values. (iii) CROQUE-OQL is in the flavor of ODMG-OQL. An example for a rather complex algebra is the AQUA algebra [LMS+93] proposed for the EREQ project. Here several operators are defined in a functional way to capture different meanings of set operations. Because it explizitly 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.

- *"logical languages":* There are some formal approaches [FM95, KLW93] which are 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 [Bee89]. A recent approach to integrate complex types into set expressions is done by Fegaras and Maier [FM95] 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 base for OO query optimization and are also exploited in the CROQUE project [GS96].

5 Conclusion and further research

We presented a framework how the ODMG model can be treated formally using a well-defined type system and the notions of the object lattice. We formalized ODMG-OQL in this context. Additionally, our type system allowed us to extend ODMG-OQL for cast and set operations.

The work presented in this paper is the basis for the CROQUE project where we study query processing and optimization in object-oriented databases. The idea is to translate OQL queries into a hybrid algebra- and calculus-based notation using monoid comprehensions that allow a uniform optimization framework for the rather different query types possible in OQL. Efficient query plans are generated in the form of an enhanced physical query algebra which describes the set-oriented evaluation of the queries. The mapping of OQL queries into calculus/algebra expressions has been done, while currently we investigate rewrite rules for optimization. On the storage level we support flexible fragmentation also including possi-

ble replications and the maintenance of materialized views. An important concept of the CROQUE approach is its flexibility for supporting different storage managers, which could either be based on different object-oriented or relational DBMS. A first prototype implementation using ObjectStore is under way.

The CROQUE approach seems also reasonably useful for other OODB research areas. Currently we investigate how temporal queries and schema evolution/integration can be realized in our framework.

Acknowledgements. We thank Dieter Gluche, Torsten Grust, Andreas Heuer, Joachim Kröger, and Andreas Henrich for intensive discussions.

References

[BdV91] H. Balsters and C. C. de Vreeze. A semantic of object-oriented sets. In *Proc. of 3rd Intl. Workshop on Database Programming Languages*, pages 187–200, Nafplion, Greece, August 1991.

[Bee89] C. Beeri. Formal models for object-oriented databases. In *[KNN89]*, pages 370–395, 1989.

[Bee90] C. Beeri. A Formal Approach to Object-Oriented Databases. *Data and Knowledge Engineering*, pages 353–382, 1990.

[BF91] H. Balsters and M. M. Fokkinga. Subtyping can have simple semantics. *Theoretical Computer Science*, 87:81–96, 1991.

[Cat96] R.G.G. Cattell, editor. *Object Database Standard, ODMG–93, Rel. 1.2.* Morgan Kaufmann Publishers, Inc., 1996.

[FM95] Leonidas Fegaras and David Maier. Towards an Effective Calculus for Object Query Languages. In *1995 ACM SIGMOD International Conference on Management of Data*, May 1995.

[GS96] T. Grust and M.H. Scholl. Translating OQL into Monoid Comprehensions — Stuck with Nested Loops? Technical Report, University of Konstanz, 1996.

[HH91] C. Hörner and A. Heuer. *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, October 1991.

[HS91a] A. Heuer and P. Sander. Classifying object-oriented query results in a class/type lattice. In *Proceedings of the 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems, Rostock, MFDBS 91*, volume 495 of *Lecture Notes in Computer Science*, pages 14–28. Springer, May 1991.

[HS91b] A. Heuer and M.H. Scholl. Principles of object-oriented query languages. In *Proc. GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, pages 178–191. Springer, March 1991.

[ICD90] *Proc. 3rd Int'l Conf. on Database Theory (ICDT'90)*, Paris, France, December 1990. Springer LNCS 470.

[KLW93] M. Kifer, G. Lausen, and J. Wu. *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, 1993.

[KNN89] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases.* Elsevier, 1989.

[LMS+93] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Stanley B. Zdonik, and other. The AQUA Data Model and Algebra. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Database Programming Languages (Proc. DBPL-4, New York)*, pages 136–156. Springer, Workshops in Computing, August 1993.

[LS93] C. Laasch and M. H. Scholl. A functional object database language. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Database Programming Languages (Proc. DBPL-4, New York)*, pages 136–156. Springer, Workshops in Computing, August 1993.

[MCB90] M.V. Mannino, I.J. Choi, and D.S. Batory. The object-oriented functional data language. *IEEE Transactions on Software Engineering*, 16(11):1258–1272, November 1990.

[SLR+93] M.H. Scholl, C. Laasch, C. Rich, M. Tresch, and H.-J. Schek. The COCOON object model. Technical Report 192, ETH Zürich, Dept. of Computer Science, December 1992. Also available as Technical Report 93-02, University of Ulm, Dept. of Computer Science, February 1993.

[SÖ95] D.D. Straube and M.T. Özsu. Query Optimization and Execution Plan Generation in Object-Oriented Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):210–227, April 1995.

[SS90] M.H. Scholl and H.-J. Schek. A relational object model. In ICDT90 [ICD90], pages 89–105.

[Str91] D.D. Straube. *Queries and Query Processing in Object-Oriented Database Systems.* PhD thesis, University of Alberta, 1991.

[SZ89] G.M. Shaw and S.B. Zdonik. An object-oriented query algebra. *IEEE Data Engineering*, 12(3):29–36, September 1989.

A Formal semantics of CROQUE-OQL

In this section we give the formal semantics of our query language. For each term of CROQUE-OQL we explain the following:

1. preconditions of the applicability,
2. the syntax of the term,
3. the result type of the query expression using the $\frac{premise}{implication}$ notation meaning the logical implication that given the types in *premise* the result type is derived in the *implication* part. The notation *query* :: *type* means that the query *query* has the type *type*.

4. the instance of the query.

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:

  1. Let I be a list or an array. $I[k]$ returns the k-th value of I. If k is not a valid index for I, $\perp_T$ will be returned.
  2. Let I_1 resp. I_2 be two lists or two arrays with valid indizes $[0 \ldots n_1]$ and $[0 \ldots n_2]$. Then $I = I_1 \ + \ I_2$ denotes a list (or an array) with valid indizes $[0 \ldots n_1 + n_2 + 1]$ and $\forall j \in [0 \ldots n_1] : \ I[j] = I_1[j]$ and $\forall j \in [0 \ldots n_2] : \ I[n_1 + 1 + j] = I_1[j]$

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 refers to the type expression $\langle int \rangle$. We assume the existence of two disjoint sets:

- $\mathcal{LABELS}$ is the set of names used for labels of classes, types, functions, or components of structs.
- $\mathcal{IDENTIFIER}$ is the set of identifiers used as iterator variables in `where` or `from` clauses.

A query is constructed by the terms built above. It should be mentioned that queries can have each type and are not restricted to set queries. Thus, each correct term without free occurences of identifiers is a valid query.

We introduce our concepts using the partition into nine groups of queries as given in [Cat96]:

- *basics:*
  - 1. *precondition*: $c \in \mathcal{V}, c :: \tau$,
    2. *syntax*: c,
    3. *typing rule*: $\frac{}{c::\tau}$
    4. *instance*: $[\![\ c\ ]\!] = c$.
  - 1. *precondition*: *entry_point* is a named database instance.
    2. *syntax*: *entry_point*
    3. *typing rule*: $\frac{entry\_point::\tau}{entry\_point::\tau}$
    4. *instance*: $[\![\ entry\_point\ ]\!] = \sigma(entry\_point)$
  - 1. *precondition*: *query_name* is the name of a view defined by *query*.
    2. *syntax*: *query_name*
    3. *typing rule*: $\frac{query\_name:=query,\ query::\tau}{query\_name::\tau}$
    4. *instance*: $[\![\ query\_name\ ]\!] = [\![\ query\ ]\!]$

- 1. *precondition*: none
  2. *syntax*: $(query)$
  3. *typing rule*: $\frac{query::\tau}{(query)::\tau}$
  4. *instance*: $[\![\, (query) \,]\!] = [\![\, query \,]\!]$

- *arithmetics and aggregate functions:*

  - 1. *precondition*: f is a function of type $\tau_1 \times \ldots \times \tau_n \mapsto \tau$.
    2. *syntax*: $f(t_1, \ldots, t_n)$
    3. *typing rule*: $\frac{f:\tau_1 \times \ldots \times \tau_n \mapsto \tau, t_1:\tau_1, \ldots t_n:\tau_n}{f(t_1,\ldots,t_n)::\tau}$
    4. *instance*: $[\![\, f(t_1, \ldots, t_n) :: \tau \,]\!] = [\![\, f \,]\!]([\![\, t_1 \,]\!], \ldots, [\![\, t_n \,]\!])$

- *comparisons and boolean expressions:*

1. (a) *precondition*: none
   (b) *syntax*: t_1 `and` t_2
   (c) *typing rule*: $\frac{t_1::bool,\ t_2::bool}{t_1 \texttt{ and } t_2::bool}$
   (d) *instance*: , $[\![\, t_1 \texttt{ and } t_2 \,]\!] = \begin{cases} false & : \ [\![\, t_1 \,]\!] = false \ \vee \ [\![\, t_2 \,]\!] = false \\ true & : \ [\![\, t_1 \,]\!] = [\![\, t_2 \,]\!] = true \end{cases}$
2. (a) *precondition*: none
   (b) *syntax*: t_1 `or` t_2
   (c) *typing rule*: $\frac{t_1::bool,\ t_2::bool}{t_1 \texttt{ or } t_2::bool}$
   (d) *instance*: $[\![\, t_1 \texttt{ or } t_2 \,]\!] = \begin{cases} true & : \ [\![\, t_1 \,]\!] = true \ \vee \ [\![\, t_2 \,]\!] = true \\ false & : \ [\![\, t_1 \,]\!] = [\![\, t_2 \,]\!] = false \end{cases}$
3. (a) *precondition*:
   (b) *syntax*: `not` t
   (c) *typing rule*: $\frac{t::bool}{\texttt{not } t::bool}$
   (d) *instance*: $[\![\, \texttt{not } t \,]\!] = \begin{cases} true & : \ [\![\, t \,]\!] = false \\ false & : \ [\![\, t \,]\!] = true \end{cases}$
4. (a) *precondition*: t_1 and t_2 have the same type τ.
   (b) *syntax*: $t_1 \ = \ t_2$
   (c) *typing rule*: $\frac{t_1::\tau,\ t_2::\tau}{t_1 \ = \ t_2::bool}$
   (d) *instance*: $[\![\, t_1 \ = \ t_2 \,]\!] = \begin{cases} true & : \ [\![\, t_1 \,]\!] = [\![\, t_2 \,]\!] \\ false & : \ otherwise \end{cases}$
5. (a) *precondition*: t_1 and t_2 have the same type τ.
   (b) *syntax*: $t_1 \ != \ t_2$
   (c) *typing rule*: $\frac{t_1::\tau,\ t_2::\tau}{t_1 \ != \ t_2::bool}$
   (d) *instance*: $[\![\, t_1 \ != \ t_2 \,]\!] = \begin{cases} true & : \ [\![\, t_1 \,]\!] \neq [\![\, t_2 \,]\!] \\ false & : \ otherwise \end{cases}$
6. (a) *precondition*: $id \in \mathcal{IDENTIFIER}$ is free in $query_2$. τ is a collection type.
   (b) *syntax*: `for all` id `in` $query_1 : query_2$

(c) *typing rule*: $\frac{query_1::\tau, query_2::bool}{\texttt{for all } id \texttt{ in } query_1 : query_2::bool}$

(d) *instance*: $[\![\texttt{for all } id \texttt{ in } query_1 : query_2]\!] =$

$$\begin{cases} true & : \quad \forall\, t \in [\![query_1]\!] \;:\; [\![query_2(t)]\!] = \texttt{true} \\ false & : \quad otherwise \end{cases}$$

7. (a) *precondition*: $id \in \mathcal{IDENTIFIER}$ is free in $query_2$. τ is a collection type.

(b) *syntax*: $\texttt{exists } id \texttt{ in } query_1 : query_2$

(c) *typing rule*: $\frac{query_1::\tau,\ query_2::bool}{\texttt{exists } id \texttt{ in } query_1 : query_2::bool}$

(d) *instance*: $[\![\texttt{exists } id \texttt{ in } query_1 : query_2]\!] =$

$$\begin{cases} true & : \quad \exists\, t \in [\![query_1]\!] \;:\; [\![query_2(t)]\!] = \texttt{true} \\ false & : \quad otherwise \end{cases}$$

8. (a) *precondition*:

(b) *syntax*: $query_1 \texttt{ in } query_2$

(c) *typing rule*: $\frac{query_1::\tau,\ query_2::collection(\tau)}{query_1 \texttt{ in } query_2::bool}$

(d) *instance*: $[\![query_1 \texttt{ in } query_2]\!] = \begin{cases} true & : \quad [\![query_1]\!] \in [\![query_2]\!] \\ false & : \quad otherwise \end{cases}$

- *constructors and conversions:*

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

type of τ_2	*result*	*further conditions*
basic type	v	$: [\![I]\!] = v$
struct	$(L_1 : (\tau_{2,1})\ c_1, \ldots, L_m : (\tau_{2,m})\ c_m)$	$: [\![I]\!] = (L_1 : c_1, \ldots, L_m : c_m, \ldots, L_n : c_n) \ \wedge\ \tau_2 = (L_1 : \tau_{2,1}, \ldots, L_m : \tau_{2,m})$
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	$o \rightarrow (\tau_2')\ v$	$: [\![I]\!] = o \rightarrow v \wedge \tau_2 = \mathcal{D}_{Object} \rightarrow_{fin} \tau_2'$
object type	$\sigma([f_1, \ldots, f_m])$	$: [\![I]\!] = \sigma([f_1, \ldots, f_m, \ldots, f_n])$

  – 1. *precondition*: f_{n+1} is a new function $f_{n+1} : \mathcal{D}_{Object} \mapsto \tau_1$ and $query_2$ is a collection of mutable objects.
  2. *syntax*: $\texttt{extend}[f_{n+1} : query_1](query_2)$
  3. *typing rule*: $\frac{query_1::\tau_1, query_2::\tau([f_1,\ldots,f_n])}{\texttt{extend}[f:query_1](query_2)::\tau([f_1,\ldots,f_n,f_{n+1}])}$
  4. *instance*: $[\![\texttt{extend}[f_{n+1} : query_1](query_2)]\!] = [\![query_2]\!]$, $o \in [\![\texttt{extend}[f_{n+1} : query_1](query_2)]\!] \ \Rightarrow [\![o.f_{n+1}]\!] = [\![query_1(o)]\!]$

  – 1. *precondition*: none
  2. *syntax*: $\texttt{struct } (label_1 : query_1, \ldots, label_n : query_n)$

3. *typing rule*: $\frac{query_1::\tau_1,\ldots,query_n::\tau_n}{\texttt{struct}\ (label_1:query_1,\ldots,label_n:\ query_n)::(label_1::\tau_1,\ldots,label_n::\tau_n)}$
4. *instance*: $[\![\ \texttt{struct}\ (label_1 : query_1, \ldots, label_n : query_n)\ ]\!] = (label_1 : [\![query_1\ ]\!], \ldots, label_n : [\![\ query_n\ ]\!])$

– 1. *precondition*: none
2. *syntax*: $\texttt{set}(query_1, \ldots, query_n)$
3. *typing rule*: $\frac{query_1::\tau,\ldots,\ query_n::\tau}{\texttt{set}(query_1,\ldots,query_n\})::\{\tau\}}$
4. *instance*: $[\![\ \texttt{set}(query_1, \ldots, query_n)\ ]\!] = \{[\![\ query_1\ ]\!], \ldots, [\![\ query_n\ ]\!]\}$

– 1. *precondition*: none
2. *syntax*: $\texttt{bag}(query_1, \ldots, query_n)$
3. *typing rule*: $\frac{query_1::\tau,\ldots,\ query_n::\tau}{\texttt{bag}(query_1,\ldots,query_n\})::\{\!\{\tau\}\!\}}$
4. *instance*: $[\![\ \texttt{bag}(query_1, \ldots, query_n)\ ]\!] = \{\!\{[\![\ query_1\ ]\!], \ldots, [\![\ query_n\ ]\!]\}\!\}$

– 1. *precondition*: none
2. *syntax*: $\texttt{list}(query_1, \ldots, query_n)$
3. *typing rule*: $\frac{query_1::\tau,\ldots,\ query_n::\tau}{\texttt{list}(query_1,\ldots,query_n)::\langle\tau\rangle}$
4. *instance*: $[\![\ \texttt{list}(query_1, \ldots, query_n)\ ]\!] = \langle\ [\![\ query_1]\!], \ldots, [\![\ query_n\ ]\!]\ \rangle$

– 1. *precondition*: none
2. *syntax*: $\texttt{array}(query_0, \ldots, query_n)$
3. *typing rule*: $\frac{query_0::\tau,\ldots,\ query_n::\tau}{\texttt{array}(query_1,\ldots,query_n)::\tau[\ ]}$
4. *instance*: $[\![\ \texttt{array}(query_0, \ldots, query_n)\ ]\!] = [\ [\![\ query_0\ ]\!], \ldots, [\![\ query_n\ ]\!]\ ]$

- *accessors:*

– Properties of an object can be accessed using the dot notation. Valid dots are . and $\rightarrow$.

– 1. *precondition*: *query* is an object type or a struct type. p is a component of τ.
2. *syntax*: *query dot p*
3. *typing rule*: $\frac{query::\tau_1,\ p::\tau_2}{query\ dot\ p::\tau_1}$
4. *instance*: $[\![\ query\ dot\ p\ ]\!] = [\![\ query\ ]\!]\Big|_{[\![\ p\ ]\!]}$

– 1. *precondition*: τ is a array of list type.
2. *syntax*: $\texttt{first}(query)$
3. *typing rule*: $\frac{query::\tau(\tau_1)}{\texttt{first}(query)::\tau_1}$
4. *instance*: $[\![\ \texttt{first}(query)\ ]\!] = [\![\ query[0]\ ]\!]$

– 1. *precondition*: *query* has a list or array type and n is the highest valid index of *query*.
2. *syntax*: $\texttt{last}(query)$
3. *typing rule*: $\frac{query::\tau(\tau_1}{\texttt{last}(query)::\tau_1}$
4. *instance*: $[\![\ \texttt{last}(query)\ ]\!] = [\![\ query[n]\ ]\!]$

– 1. *precondition*: none
2. *syntax*: $\texttt{listtoset}(query)$
3. *typing rule*: $\frac{query::\langle\tau\rangle}{\texttt{listtoset}(query)::\{\tau\}}$

4. *instance*: $[\![\texttt{listtoset}(query)]\!] = \{l | l \in [\![query]\!]\}$

– 1. *precondition*: τ is a collection type.
2. *syntax*: `element` $(query)$
3. *typing rule*: $\frac{query::\tau(\tau_1)}{\texttt{element } (query)::\tau_1}$
4. *instance*: $[\![\texttt{element } (query)]\!] =$
$$\begin{cases} v & : \quad v \in [\![query]\!] \quad \text{(Nondeterminism if } card([\![\text{query }]\!]) > 1) \\ \bot_\tau & : \quad [\![query]\!] = \{\} \end{cases}$$

- *set and collection expression:*

– 1. *precondition*: τ_1 and τ_2 are collection types. It holds that $\neg(\tau_2 = \langle\tau_3\rangle \ \vee \tau_2 = \tau_3[\,]) \wedge (\tau_1 = \{\tau_2\} \vee \tau_1 = \{\!\{\tau_2\}\!\})$ [6]
2. *syntax*: `flatten` $(query)$
3. *typing rule*: $\frac{query::\tau_1(\tau_2(\tau_3))}{\texttt{flatten } (query)::\tau_2}$
4. *instance*: $[\![\texttt{flatten } (query)]\!] =$
$$\begin{cases} \{t | \exists t_1 \in [\![query]\!] : t \in t_1\} : \tau_2 = \{\tau_3\} \\ \{\!\{t | \exists t_1 \in [\![query]\!] : t \in t_1\}\!\} : \tau_2 = \{\!\{\tau_3\}\!\} \\ \langle [\![query[1]]\!], \ldots, [\![query[n]]\!] \rangle : \tau_2 = \langle\tau_2\rangle \ \wedge (\tau_1 = \langle\tau_2\rangle \vee \tau_1 = \tau_2[\,]) \\ [\![query[1]]\!] \ + \ \ldots \ + \ [\![query[n]]\!] : \tau_2 = \tau_3[\,] \wedge (\tau_1 = \langle\tau_2\rangle \vee \tau_1 = \tau_2[\,]) \end{cases}$$

– 1. *precondition*: τ_1 and τ_2 are collection types. $(\tau_2 = \langle\tau_3\rangle \ \vee \tau_2 = \tau_3[\,]) \wedge (\tau_1 = \{\tau_2\} \vee \tau_1 = \{\!\{\tau_2\}\!\})$
2. *syntax*: `flatten` $(query)$
3. *typing rule*: $\frac{query::\tau_1(\tau_2(\tau_3))}{\texttt{flatten } (query)::\{\tau_2\}}$
4. *instance*: $[\![\texttt{flatten } (query)]\!] = \{t | \exists t_1 \in [\![query]\!] : t \in t_1\}$

– "set operations": The typing rules are different for mutable and immutable objects:

∗ mutable objects:

— 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*:
$$t \in [\![query_1 \ \texttt{intersect} \ query_2]\!] :\Leftrightarrow \ t \in [\![query_1]\!] \ \wedge \ t \in [\![query_2]\!],$$
$$card(t, [\![query_1 \ \texttt{intersect} \ query_2]\!]) = min(card(t, [\![query_1]\!]), card(t, [\![query_2]\!])$$

— 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*:
$$t \in [\![query_1 \ \texttt{union} \ query_2]\!] :\Leftrightarrow \ t \in [\![query_1]\!] \ \vee \ t \in [\![query_2]\!],$$
$$card(t, [\![query_1 \ \texttt{union} \ query_2]\!]) = card(t, [\![query_1]\!]) + card(t, [\![query_2]\!])$$

[6] This special case is formalized in the next item.

— 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:

— 1. *precondition*:$\tau_1(\tau_1')$ and $\tau_2(\tau_2')$ are set or bag types. $\tau_1' \preceq \tau_2'$. If τ_1 and τ_2 are both sets, then $\tau_3 = \{\tau_1'\}$, otherwise $\tau_3 = \{\!\{\tau_1'\}\!\}$.

2. *syntax*: $query_1$ `intersect` $query_2$, $query_2$ `intersect` $query_1$

3. *typing rules*: $\frac{query_1::\tau_1(\tau_1'),\ query_2::\tau_2(\tau_2')}{query_1\ \texttt{intersect}\ query_2::\tau_3}$ $\frac{query_1::\tau_1(\tau_1'),\ query_2::\tau_2(\tau_2')}{query_2\ \texttt{intersect}\ query_1::\tau_3}$

4. *instance*:

$$t \in\ [\![\ query_1\ \texttt{intersect}\ query_2\ ]\!] :\Leftrightarrow$$
$$t \in\ [\![\ query_2\ \texttt{intersect}\ query_1\ ]\!] :\Leftrightarrow$$
$$t \in [\![\ query_1\ ]\!]\ \wedge\ (\tau_2)\ t \in [\![\ query_2\ ]\!]$$
$$card(t,[\![\ query_1\ \texttt{intersect}\ query_2\ ]\!]) =$$
$$min(card(t,[\![query_1\ ]\!]), card((\tau)\ t,[\![query_2]\!]))$$

— 1. *precondition*:$\tau_1(\tau_1')$ and $\tau_2(\tau_2')$ are set or bag types. $\tau_1' \preceq \tau_2'$. If τ_1 and τ_2 are both sets, then $\tau_3 = \{\tau_2'\}$, otherwise $\tau_3 = \{\!\{\tau_2'\}\!\}$.

2. *syntax*: $query_1$ `union` $query_2$, $query_2$ `union` $query_1$

3. *typing rules*: $\frac{query_1::\tau_1(\tau_1'),\ query_2::\tau_2(\tau_2')}{query_1\ \texttt{union}\ query_2::\tau_3}$ $\frac{query_1::\tau_1(\tau_1'),\ query_2::\tau_2(\tau_2')}{query_2\ \texttt{union}\ query_1::\tau_3}$

4. *instance*:

$$t \in [\![\ query_1\ \texttt{union}\ query_2]\!]\ :\Leftrightarrow\ t \in [\![\ query_2\ \texttt{union}\ query_1]\!]\ :\Leftrightarrow$$
$$t \in [\![\ (\tau_2)\ query_1\ ]\!]\ \vee\ t \in [\![\ query_2\ ]\!],$$
$$card(t,[\![query_1\ \texttt{union}\ query_2]\!]) =$$
$$card(t,[\![\ (\tau_2)\ query_1\ ]\!]) + card(t,[\![\ query_2\ ]\!])$$

— 1. *precondition*:$\tau_1(\tau_1')$ and $\tau_2(\tau_2')$ are set or bag types. $\tau_2' \preceq \tau_1'$. 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*:

$$t \in [\![query_1\ \texttt{except}\ query_2\ ]\!] :\Leftrightarrow\ t \in [\![\ query_1\ ]\!] \wedge \neg (t \in [\![\ (\tau_1) query_2\ ]\!]),$$
$$card(t,[\![query_1\ \texttt{except}\ query_2]\!]) =$$
$$max(card(t,[\![\ \ query_1\ ]\!]) - card(t,[\![\ (\tau_1)\ query_2\ ]\!]), 0)$$

– 1. *precondition*: For each p_j there is a label L_j that is unique for $p_1, \ldots, p_m$, i.e. L_j is a name of an identifier, an attribute or of an instance. Renamings within p_j can be expressed using the syntax *new_label* : *query*. The use of the wildcard '*' is substituted by a set of explicit references. $Q_1, \ldots, Q_n$ are collection queries and may use an identifier using the syntax Q_j `as` *id* or Q_j *id*. Furthermore, $query_3$ is a condition that is well-typed for $Q_1, \ldots, Q_n$.

2. *syntax*: `select` $p_1, \ldots, p_m$ `from` $Q_1, \ldots, Q_n$ `where` $query_3$

3. *typing rule*: $\frac{p_1:\tau_1,\ldots,p_m:\tau_m,\ query_3::bool}{\texttt{select}\ p_1,\ldots,p_m\ \texttt{from}\ Q_1,\ldots,Q_n\ \texttt{where}\ query_3::\{\!\{(L_1:\tau_1,\ldots,L_m:\tau_m)\}\!\}}$
4. *instance*: $card(t, [\![\texttt{select}\ p_1,\ldots,p_m\ \texttt{from}\ Q_1,\ldots,Q_n\ \texttt{where}\ query_3\ ]\!]) =$
$\sum_{t'\in\ I'\wedge [\![\ query_3(t')\ ]\!]=\texttt{true}\wedge t=(L_1:[\![p_1(t')]\!],\ldots,L_m:[\![p_m(t')]\!])} card(t', I')$,
where I' is given as follows: $I' = \{\!\{(q_1,\ldots,q_n) | q_1 \in Q_1,\ldots,q_n \in Q_n\}\!\}$.

– 1. *precondition*: For each p_j there is a label L_j that is unique for $p_1,\ldots,p_m$, i.e. L_j is a name of an identifier, an attribute or of an instance. Renamings within p_j can be expressed using the syntax *new_label* : *query*. The use of the wildcard '*' is substituted by a set of explicit references. $Q_1,\ldots,Q_n$ are collection queries and may use an identifier using the syntax Q_j `as` *id* or Q_j *id*. Furthermore, $query_3$ is a condition that is well-typed for $Q_1,\ldots,Q_n$.
2. *syntax*: `select distinct` $p_1,\ldots,p_m$ `from` $Q_1,\ldots,Q_n$ `where` $query_3$
3. *typing rule*: $\frac{p_1:\tau_1,\ldots,p_m:\tau_m,\ query_3::bool}{\texttt{select distinct}\ p_1,\ldots,p_m\ \texttt{from}\ Q_1,\ldots,Q_n\ \texttt{where}\ query_3::\{\ (L_1,\tau_1,\ldots,L_m:\tau_m)\}}$
4. *instance*: $t \in [\![\texttt{select distinct}\ p_1,\ldots,p_m\ \texttt{from}\ Q_1,\ldots,Q_n\ \texttt{where}\ query_3\ ]\!]$
$:\Leftrightarrow\ \exists t' \in\ I' \wedge [\![\ query_3(t')\ ]\!] = \texttt{true} \wedge t = (L_1 : [\![p_1(t')]\!],\ldots,L_m : [\![p_m(t')]\!])$,
where I' is given as follows: $I' = \{\!\{(q_1,\ldots,q_n) | q_1 \in Q_1,\ldots,q_n \in Q_n\}\!\}$.

– 1. *precondition*: $\tau(\tau')$ is a set, bag, list, or array type. $\tau_1,\ldots,\tau_n$ are components of τ'. Each τ_j $(j = 1,\ldots,n)$ is a totally ordered type.
2. *syntax*: *query* `order by` $\tau_1,\ldots,\tau_n$
3. *typing rule*: $\frac{query::\tau(\tau')}{query\ \texttt{order by}\ \tau_1,\ldots,\tau_n::\langle\tau'\rangle}$
4. *instance*: $v \in [\![\ query\ \texttt{order by}\ \tau_1,\ldots,\tau_n\ ]\!]\ :\Leftrightarrow\ v \in [\![\ query\ ]\!]$;
$k_1 \le k_2 \wedge v_1, v_2 \in [\![\ query\ \texttt{order by}\ \tau_1,\ldots,\ \tau_n\ ]\!] \wedge$
$[\![\ query\ \texttt{order by}\ \tau_1,\ldots,\ \tau_n\ ]\!][k_1] = v_1 \wedge$
$[\![\ query\ \texttt{order by}\ \tau_1,\ldots,\ \tau_n\ ]\!][k_2] = v_2\ \Rightarrow\ v_1 \le_{\tau_1,\ldots,\tau_n} v_2$

– ODMG distinguishes two, slightly different versions of `group by`. Both can be formalized as macros of other CROQUE-OQL queries. [7]

∗ 1. *precondition:* $\tau(\tau_1)$ is a collection type, p is a component of τ_1, and np is a new label.
2. *syntax: query* `group by` p `with` $np : expr$
3. *typing rule:* $\frac{query::\tau,\ expr::\tau_2}{query\ \texttt{group\_by}\ p\ \texttt{with}\ np:expr::\{(\tau_1,\tau_2)\}}$
4. *instance:* $[\![\ query\ \texttt{group\_by}\ p\ \texttt{with}\ np : expr]\!] =$
$[\![\ \texttt{select distinct struct}\ (p : e.p, np : expr(e))\ \texttt{from}\ query\ \texttt{as}\ e\ ]\!]$

∗ 1. *precondition:* $\tau(\tau_1)$ is a collection type, $p_1,\ldots,p_n, np \in \mathcal{LABEL S}$.
2. *syntax: query* `group by` $(p_1 : cond_1,\ldots,p_n : cond_n)$ `with` $np : expr$
3. *typing rule:*
$$\frac{query::\tau,\ cond_1::bool,\ldots,\ cond_n::bool, expr::\tau_2}{\begin{array}{l} query\ \texttt{group by}\ (p_1 : cond_1,\ldots,p_n : cond_n) \\ \texttt{with}\ np : expr :: \{(p_1 : bool,\ldots,p_n : bool, partition : \{\!\{(query : \tau_1, np : \tau_2)\}\!\})\} \end{array}}$$
4. *instance:* $[\![\ query\ \texttt{group\_by}\ (p_1 : cond_1,\ldots,p_n : cond_n)\ \texttt{with}\ np : expr]\!] =$
$[\![\ \texttt{select distinct struct}\ (p_1 : cond_1(e),\ldots,p_n : cond_n(e),$
$np : expr(e), partition : \texttt{select}\ e_1\ \texttt{from}\ query\ \texttt{as}\ e_1$
$\texttt{where}\ cond_1(e) = cond_1(e_1) \wedge \ldots \wedge\ cond_n(e) = cond_n(e_1))$
$\texttt{from}\ query\ \texttt{as}\ e\ ]\!]$

[7] Here we differ a little from the textual description of [Cat96] and use the notions of ODMG 1.1, which are more in the flavor of the CROQUE query language.