Informatik-Forschungsbericht 10/94, Universität Bremen, ISSN-0722-8996 (1994)

On a Better Formal Basis for Stating SQL-like Queries in Value- and Object-Based DBS

Rudolf Herzig and Martin Gogolla

Universität Bremen
Fachbereich Mathematik und Informatik
Arbeitsgruppe Datenbanksysteme
Postfach 33 04 40, D-28334 Bremen, Germany
e-mail: {herzig|gogolla}@informatik.uni-bremen.de

Abstract

We present a formalism whose purpose is to serve as a rigid basis for describing SQL-like queries and constraints in context of both value- and object-based data models. The formalism, whose major contribution lies in its inherent orthogonality and rigorous mathematical foundation, is defined independently of any concrete database model. Instead it offers a general facility for the ad-hoc manipulation of structured values. In an improvement to the well-known relational domain or tuple calculi (or their extensions towards extended relational models) the formalism (1) guarantees safe and computable queries in an immanent way, (2) allows to explain duplicates in query results, and (3) supports the composition of queries from subqueries without the need to name intermediate query results. Hence it should be seen closer to concrete SQL than classical query calculi.

1 Introduction

Object-oriented database systems (OODB) are usually embedded in a programming language environment providing full computational power. Consequently, designers of OODB at first did not feel the need to integrate ad hoc query facilities into their systems. However, it has been recognized that some associative retrieval is of importance even for OODB [BCD89, Bee90, BNPS92].

A well-tried ad-hoc query language is SQL. SQL came up with relational database systems so that many people regard SQL as a typical relational query language. On the other hand SQL has been successfully adapted to semantic data models [HK87, EGH$^+$92, HE92], and there have been many proposals for applying SQL to OODB [BCD89, CDLR90, DGJ92, GV92, KKS92, BH93]. A number of these SQL variants have already found their way into commercial products like OSQL [Bee88] in IRIS, Object SQL [HD91] in ONTOS, RELOOP [CDLR90] and its successor O_2Query [BDK92] in O_2, CQL++ [DGJ92] in Ode, XSQL [KKS92] in ORION. A derived form of O_2Query is being considered as the query language OQL of the ODMG-93 standards proposal [Cat94, Kim94].

Unfortunately the syntax of many of these query languages is illustrated by some examples only, and their semantics is mostly described in an informal way. In other cases the semantics is defined by an involved translation into an *algebra-based* framework (e.g., RELOOP [CDLR90]) or a *rule-based* (logic programming oriented) formalism (e.g., ESQL2 [GV92], XSQL [KKS92]). However, with regard to SQL *calculus-based* formalisms seem to be more helpful because of their closeness to the basic select-from-where construct found in SQL (cf. [NPS91]).

In this paper we present a query formalism called *QSV* (Queries against Structured Values) the aim of which is to provide a general framework for SQL-like query languages. The formalism is defined in such a way that it can be easily applied to relational, extended relational, or object-based data models. In comparison with other query language proposals we stress the following advantages:

Completely orthogonal SQL: In contrast to many proposals which allow for nested queries only in the *from* and *where* clause of a select query, our query formalism allows for arbitrary terms, in particular further select terms, to be used also in the *select* clause of a select query.

Abstract syntax: The syntax of the query formalism is precisely defined by mathematical notions being independent from a concrete language and thereby covering both context-free and context-sensitive rules in a formal and compact manner.

Straightforward semantics: Each syntactical category is directly given a rigorous formal semantics based on sets.

A concrete SQL-like language proposal which allows for nested queries in the *select*, *from*, and *where* clause of a select term is RELOOP (and O_2Query resp.). It is defined in context of an object model with structured values [LR89]. However we are not aware of a complete formal description of this language.

The presented query formalism is essentially useful for the ad-hoc manipulation of structured values. With that it stands in tradition of the query calculi proposed for complex object models [KV84, BK86, RKS88, ABGG89]. Most of these calculi turn out to be direct extensions of relational domain and tuple calculus, thereby showing the same deficiencies in explaining some important facets of concrete SQL:

1. *Missing simple characterization of safe queries:* Standard SQL queries are always safe, meaning that they always define finite and computable answers. By default, relational calculi allow to define unsafe queries, or syntactical restrictions to define safe queries are quite involved.

2. *Missing bag handling:* Answers to SQL queries may contain duplicates. This situation cannot be explained in relational calculi which are generally defined in a context of sets, but what would be needed is a framework of multisets. Even the query mechanism itself is not appropriate to explain duplicates in query results because a query is generally considered to be given as a formula $\varphi(t)$ with a free variable t, the answer being defined to be the set $\{t \mid \varphi(t)\}$.

3. *Missing means for query composition:* SQL queries can be nested. This situation cannot be adequately mirrored in relational calculi because in contrast to stored relations which are treated as predicates they do not provide means to calculate with computed relations. Of course, one way out would be to name intermediate query results (as possibly parameterized) predicates, but this is not in the spirit of SQL.

Now QSV gives the following answers to these problems: (1) Safe queries are guaranteed in QSV by introducing a special syntactic category called *declarations* by which all variables are forced to be bound to finite constant, stored, or computed ranges. (2) The notion of variable declaration shows also a certain impact on query evaluation. Query evaluation is done by looking for all possible assignments of values to declared variables. The fact that different assignments may lead to the same value in the target term of a select query makes it possible to explain duplicates in query results. (3) In contrast to classical calculi where queries are understood as formulas with free variables, in QSV select queries are defined as terms which are built on the basis of other terms, formulas, and declarations. Hence QSV does not preserve the traditional hierarchical structure of predicate calculus, where terms are used to build formulas, but not vice versa.

The paper is organized as follows. In Section 2 we introduce a general model of structured values. In Section 3 we define QSV as a basic formalism for the SQL-like manipulation of structured values. The application of this formalism to database states is sketched in Section 4. We give some concluding remarks in Section 5.

2 A Model of Structured Values

The underlying data model of our query formalism to be presented in Section 3 is given by a model of structured values. A structured (or complex) value is either an *atomic value* being an instance of some standard or user-defined data sort, a *set* or *bag value*[1] gathering equally structured complex values, or a *tuple value* combining complex values with possibly different structure. Both atomic and composite values are accompanied by specific operations.

Notation: Let sets $S, S_1, \ldots, S_n$ be given. Then $\boldsymbol{\mathcal{F}(S)}$ denotes the restriction of the powerset of S to finite sets, $\boldsymbol{\mathcal{B}(S)}$ the set of finite multisets (bags) over S, and $\boldsymbol{S_1 \times \ldots \times S_n}$ the Cartesian product of the sets $S_1, \ldots, S_n$. Finite sets are written as $\boldsymbol{\{c_1, \ldots, c_n\}}$, bags as $\boldsymbol{\{\!\{ c_1, \ldots, c_n \}\!\}}$, and elements of the Cartesian product as $\boldsymbol{(c_1, \ldots, c_n)}$.

2.1 Atomic sorts

We define values as instances of data sorts. Data sorts are connected with specific operations and predicates. All these are summarized in a data signature.

Definition 1 (data signature). A data signature is given by a triple $\Sigma = (S, \Omega, \Pi)$ in which S denotes a set of sorts, $\Omega = \{\Omega_s\}_{s \in S^* \times S}$ a family of $S^* \times S$-indexed operation symbols and $\Pi = \{\Pi_s\}_{s \in S^*}$ a family of S^*-indexed predicate symbols. □

$\omega_{s_1 \ldots s_n, s} \in \Omega$ is also written as $\omega : s_1 \times \ldots \times s_n \rightarrow s$ and $\pi_{s_1 \ldots s_n} \in \Pi$ as $\pi : s_1 \times \ldots \times s_n$.

Examples.

1. The following sorts and associated operations and predicates are standard.

   $S \supseteq \{$ nat, int, real, bool, char, string, $\ldots \}$
   $\Omega \supseteq \{ +, -, *, \mathtt{DIV}, \mathtt{MOD} : \text{int} \times \text{int} \rightarrow \text{int}, \ldots \}$
   $\Pi \supseteq \{ <, \leq, =, \geq, >: \text{int} \times \text{int}, \ldots, \}$

   Data constants are considered as nullary operation symbols, e.g., $42 :\rightarrow$ int or true $:\rightarrow$ bool.

2. A data signature may also contain *non-standard* data types.

[1] A bag (or multiset) is a set in which the occurrences of each element are counted, i.e., for bags we have $\{\!\{ 1,1,2 \}\!\} \neq \{\!\{ 1,2 \}\!\}$ while there is $\{1,1,2\} = \{1,2\}$ for sets.

$S \supseteq$ { polygon, ... }
$\Omega \supseteq$ { area : polygon $\longrightarrow$ real, ... }
$\Pi \supseteq$ { cuts : polygon $\times$ polygon, ... }

For a data signature Σ we assume an interpretation structure $I(\Sigma)$ to be given as a many-sorted algebra.

Definition 2 (interpretation of a data signature). For a given data signature Σ an interpretation structure is defined as a triple $I(\Sigma) = (I(S), I(\Omega), I(\Pi))$.

- $I(S)$ associates each sort $s \in S$ with a set $I(s)$ such that $\perp_s \in I(s)$.
- $I(\Omega)$ associates each operation symbol $\omega : s_1 \times \ldots \times s_n \longrightarrow s \in \Omega$ with a total function $I(\omega) : I(s_1) \times \ldots \times I(s_n) \longrightarrow I(s)$.
- $I(\Pi)$ associates each predicate symbol $\pi : s_1 \times \ldots \times s_n \in \Pi$ with a relation $I(\pi) \subseteq I(s_1) \times \ldots \times I(s_n)$.

□

Example. The interpretation of standard data sorts may be fixed as follows.

$I(\text{nat}) = \mathbb{N}_0 \cup \{\perp_{\text{nat}}\}$, $I(\text{int}) = \mathbb{Z} \cup \{\perp_{\text{int}}\}$, $I(\text{real}) = \mathbb{Q} \cup \{\perp_{\text{real}}\}$,
$I(\text{bool}) = \{\mathit{true},\ \mathit{false}\} \cup \{\perp_{\text{bool}}\}$,
$I(\text{char}) = A \cup \{\perp_{\text{char}}\}$, $I(\text{string}) = A^* \cup \{\perp_{\text{string}}\}$,

in which A denotes a finite set of characters. Standard operations and predicates have the usual meaning.

An element $\underline{s} \in I(s)$ is called a *value*. Values are written *italic*. For example the constant $42 :\longrightarrow$ int generates the value *42*.

For each sort s there is a special *bottom value*. In consideration of the fact that operations are viewed as total functions bottom values are useful to simulate partial functions. For instance we have $I(\texttt{DIV})(\mathit{1,0}) = \perp_{\text{int}}$. Bottom values are generated by $\texttt{BOTTOM}_s :\longrightarrow s \in \Omega$. For simplicity the index of the bottom value is dropped in the following.

2.2 Constructed sorts

Besides atomic data sorts we also provide *constructed* sorts by means of predefined sort constructors. Here we propose the constructors *set* and *bag* to describe multi-valued domains and *tuple* to describe composite domains. Of course, other constructors, for instance constructors to describe lists, maps, or variant records, could be added as well [HG94a].

Sort constructors can be applied iteratively to construct domains of any complexity.

Definition 3 (sort expressions). Let a set of sorts S be given. Then the set $S\text{-}Expr(S)$ of sort expressions over S is defined as follows.

i. If $s \in S$, then $\boldsymbol{s} \in S\text{-}Expr(S)$.

ii. If $s \in S\text{-}Expr(S)$, then $\mathbf{set}(\boldsymbol{s}), \mathbf{bag}(\boldsymbol{s}) \in S\text{-}Expr(S)$.

iii. If $s_1, \ldots, s_n \in S\text{-}Expr(S)$, then $\mathbf{tuple}(\boldsymbol{s_1}, \ldots, \boldsymbol{s_n}) \in S\text{-}Expr(S)$.

Let us assume that there is a fixed interpretation I of S according to Definition 2. Then the interpretation of sort expressions is defined as follows.

i. see Definition 2.

ii. $I(\text{set}(s)) := \mathcal{F}(I(s)) \cup \{\perp\}$. $\quad I(\text{bag}(s)) := \mathcal{B}(I(s)) \cup \{\perp\}$.

iii. $I(\text{tuple}(s_1, \ldots, s_n)) := (I(s_1) \times \ldots \times I(s_n)) \cup \{\perp\}$.

□

An element $\underline{s} \in I(s)$ with $s \in S\text{-}Expr(S)$ is called a *structured* or *complex value*. As in the case of atomic sorts a special bottom value belongs to the interpretation of each sort expression.

Remark (attributes in tuple expressions). In tuple expressions attributes may be added to underline the meaning of components. For example,

```
tuple(Nodes:set(int),
      Edges:set(tuple(Start:int, End:int)))
```

describes a structure for storing graphs. From a data manipulation point of view attributes in tuple expressions can be used as projection operators on tuple values. In the following we treat attributes in tuple expressions as optional items.

The defined sort constructors have been served as the basis for a wide range of *value-based* data models. A *schema* of a value-based data model can generally be expressed by a tuple expression $s = \text{tuple}(r_1{:}s_1, \ldots, r_m{:}s_m)$. The *state* of a database belonging to this schema is a structured value of sort s. Many variations exist with respect to what is allowed for the sort expressions s_i $(i = 1, \ldots, m)$.

- The **relational model** [Cod70] restricts s_i to describe *tables* of the kind $\text{set}(\text{tuple}(a_1{:}d_1, \ldots, a_n{:}d_n))$ in which d_j denote atomic data sorts.
- The **nested form (NF) models** [Mak77, SS86] allow for sort expressions s_i with alternating set and tuple constructors so that *nested tables* can be represented.
- The **complex object (CO) model** [AB88, AFS89] does not impose any restriction on the structure of s_i.

2.3 Generic functions

Sort expressions are usually associated with a large number of *generic* (or overloaded) functions. There are operations and predicates that

- *compose* structured values from simpler ones. For instance, `{ }` is used as a constructor for set values and `( )` is used as a constructor for tuple values.
- *decompose* structured values into simpler ones. For instance, $\texttt{PRJ}^i_{\text{tuple}(s_1,\ldots,s_n)} :$ $\text{tuple}(s_1, \ldots, s_n) \to s_i$ selects the ith component of a tuple.
- *convert* structured values. For instance, $\texttt{BTS}_{\text{bag}(s)} : \text{bag}(s) \to \text{set}(s)$ converts a bag into a set by duplicate elimination.
- *exploit* the distinct properties of certain kinds of structured values. For instance, $\texttt{CNT}_{\text{set}(s)} : \text{set}(s) \to \text{nat}$ counts the elements in a set (also defined for bags), $\texttt{OCC}_{\text{bag}(s)} : \text{bag}(s) \times s \to \text{nat}$ counts the occurrences of a given item in a bag, $\texttt{IN}_{\text{set}(s)} : \text{set(s)} \times s$ denotes the membership predicate (also defined for bags).

There are many other generic functions (for details see [GH91, Gog94, Her95]). All operations induced by sort expressions are summarized in $\Omega(S\text{-}Expr(S))$. Analogously all predicates induced by sort expressions are combined in $\Pi(S\text{-}Expr(S))$.

From now on let $\Sigma_D = (S_D, \Omega_D, \Pi_D)$ denote a data signature according to Definition 1 so that $\Sigma = (S, \Omega, \Pi)$ denotes an *extended* signature given by $S = S\text{-}Expr(S_D)$, $\Omega = \Omega_D \cup \Omega(S\text{-}Expr(S_D))$, and $\Pi = \Pi_D \cup \Pi(S\text{-}Expr(S_D))$.

3 SQL-like Manipulation of Structured Values

Having defined a model of structured values one is interested in the manipulation of such values. A large part of this work can be done in an algebra-like style by using the operations found in a data signature (e.g. arithmetic operations, string operations, etc.) or by making use of the generic functions being associated with the predefined sort constructors (e.g. typical set operations like union, intersection, difference, etc.). However, in this section we are more interested in means for formulating ad-hoc queries in a *Structured Query Language*-like manner. The basic scheme for formulating such queries in SQL is given by query blocks of the kind

SELECT	*target* (with free variables)
FROM	*range specification* (binding free variables)
WHERE	*condition*

which will be called a *SFW-block*.

In this section we present a query formalism which gives an answer to the question of what should be allowed for filling up the open parts of this scheme, i.e., the target, range specification, and condition clause. The global objective of this work is to achieve a maximal degree of orthogonality in the formulation of SQL-like queries being expressed by exact formal definitions.

The query formalism to be presented here is a revision of a calculus for querying extended Entity-Relationship schemas (called ER calculus) which was reported in [GH91]. Improvements on this calculus are expressed by the following points:

1. The calculus is lifted from a specific data model like the ER model to a more general model of structured values which also simplifies comparison with other approaches in this direction.

2. Important syntactical parts of the calculus, especially select terms and declarations, are drawn in considerably simplified terms.

3. The query formalism is compared with classical formalisms for formulating queries in the relational model (relational domain and tuple calculus and relational algebra) and more recent proposals for formulating such queries in extended relational models (the calculus of Abiteboul/Beeri [AB88] and NF^2-algebra of Schek/Scholl [SS86]).

4. The necessary enhancements when switching from queries in value-based to queries in object-based data models are discussed in Section 4.

We start by giving the formal definition of our query formalism which will be called QSV (*Queries* against *Structured Values*). We thoroughly discuss the evaluation of SFW-blocks by examples. Then we relate our query formalism to other approaches in this field.

Remark (usage of the term "calculus"). Query formalisms based on predicate logic are traditionally called query calculi but in mathematical logic the notion of calculus is reserved to denote a logic with a corresponding derivation system (proof theory).

Our query formalism defines nothing else than a formal language with a fixed interpretation (model theory)[2]. Therefore we refrain from calling our framework a query calculus. Instead we use the neutral notion of a query (and constraint) formalism.

3.1 Formal definition of QSV

The query formalism comprises the following syntactical categories:

Terms: A term is always evaluated to a structured value. The target clause of SFW-block is given by a term.

Formulas: A formula is always evaluated to a truth value. The condition clause of a SFW-block is given by a formula.

Declarations: A declaration is used to bind one or more variables to finite ranges. The range specification clause of a SFW-block is given by a declaration.

Terms may consist of variables. For the evaluation of such terms we need a notion of variable assignment.

Definition 4 (variables and variable assignments). Let a family $Var = \{Var_s\}_{s \in S}$ of S-indexed variables be given. The set of variable assignments A is defined by $A := \{\alpha \mid \alpha : Var_s \to I(s) \text{ for all } s \in S\}$. The special assignment $\epsilon : Var \to \{\perp\}$ is called the *empty assignment*. □

The index of a variable symbol denotes the sort of the variable.

Definition 5 (terms). The syntax of terms is given by an S-indexed family $Term = \{Term_s\}_{s \in S}$ and a function free : $Term \to \mathcal{F}(Var)$ defined by the following rules.

i. If $v \in Var_s$, then $\boldsymbol{v} \in Term_s$ with $\text{free}(v) := \{v\}$.

ii. If $\omega : s_1 \times \ldots \times s_n \to s \in \Omega$ and $\tau_i \in Term_{s_i}$ $(i = 1 \ldots n)$, then $\boldsymbol{\omega(\tau_1, \ldots, \tau_n)} \in Term_s$ with $\text{free}(\omega(\tau_1, \ldots, \tau_n)) := \text{free}(\tau_1) \cup \ldots \cup \text{free}(\tau_n)$.

iii. If $\tau \in Term_s$, $\delta \in Decl$ and $\varphi \in Form$, then $\boldsymbol{\{\!\!\{ \tau \mid \delta,\ \varphi \}\!\!\}} \in Term_{\text{bag}(s)}$ with $\text{free}(\{\!\!\{ \tau \mid \delta,\ \varphi \}\!\!\}) := (\text{free}(\tau) \cup \text{free}(\delta) \cup \text{free}(\varphi)) \setminus \text{decl}(\delta)$.

For a fixed interpretation I and a variable assignment $\alpha \in A$ the evaluation of terms is defined as follows.

i. $(I, \alpha)[\![v]\!] = \alpha(v)$.

ii. $(I, \alpha)[\![\omega(\tau_1, \ldots, \tau_n)]\!] = I(\omega)((I, \alpha)[\![\tau_1]\!], \ldots, (I, \alpha)[\![\tau_n]\!])$.

iii. $(I, \alpha)[\![\{\!\!\{ \tau \mid \delta,\ \varphi \}\!\!\}]\!] = \{\!\!\{ (I, \alpha')[\![\tau]\!] \mid \alpha' \in A \text{ with } \alpha'(v) = \alpha(v) \text{ for all } v \in Var \setminus \text{decl}(\delta) \text{ and } (I, \alpha') \models \delta \text{ and } (I, \alpha') \models \varphi \}\!\!\}$.

□

Variables (i) and operation symbols (ii) are standard. Terms (iii) are called *select terms*, because $\{\!\!\{ \tau \mid \delta,\ \varphi \}\!\!\}$ is merely a short-hand notation for the SQL-like query scheme *select* τ *from* δ *where* φ. τ represents the target term fixing the format of the desired result, δ is a declaration binding one or more variables to finite domains (see below), and φ is a qualifying formula.

Example (terms in concrete syntax). `2`, `2+3*7` (infix notation of operation symbols is frequently used, operations have the usual bindings), `2+x`,

[2] The difference between the two viewpoints of logic (which can be tied to the difference between syntax and semantics) and their relevance to databases have been discussed in [GMN84].

`{1,4,9}` (the curly brackets serve as constructors for set values), and `SELECT Name(x) FROM x:PERSON WHERE Age(x)<18` are possible terms.

The function free returns the free variables of a term. For select terms the free variables are given by the free variables of the target term τ, the declaration δ, and the qualifying formula φ reduced by the variables declared in δ.

A term is evaluated to a structured value the sort of which is already fixed by the index of a term. The evaluation of variables depends on the current variable assignment α, while the evaluation of operations depends on the given interpretation structure I.

Select terms are generally evaluated to bag values. Evaluation depends on the possible assignments of values to the variables declared in declaration δ. True bags are obtained when the target term τ is evaluated to the same value under different assignments. The evaluation of select terms is further discussed in Section 3.2 after having introduced formulas and declarations.

Definition 6 (formulas). The syntax of formulas is defined by a set *Form* and a function free : $Form \to \mathcal{F}(Var)$ defined by the following rules.

i. If $\pi : s_1 \times \ldots \times s_n \in \Pi$ and $\tau_i \in Term_{s_i}$ $(i = 1 \ldots n)$, then $\boldsymbol{\pi(\tau_1, \ldots, \tau_n)}$ $\in Form$ with $\text{free}(\pi(\tau_1, \ldots, \tau_n)) := \text{free}(\tau_1) \cup \ldots \cup \text{free}(\tau_n)$.

ii. If $\tau_1, \tau_2 \in Term_s$, then $\boldsymbol{\tau_1 = \tau_2} \in Form$ with $\text{free}(\tau_1 = \tau_2) := \text{free}(\tau_1) \cup \text{free}(\tau_2)$.

iii. If $\varphi \in Form$, then $\boldsymbol{\neg(\varphi)} \in Form$ with $\text{free}(\neg(\varphi)) := \text{free}(\varphi)$.

iv. If $\varphi_1, \varphi_2 \in Form$, then $\boldsymbol{(\varphi_1 \vee \varphi_2)} \in Form$ with $\text{free}((\varphi_1 \vee \varphi_2)) := \text{free}(\varphi_1) \cup \text{free}(\varphi_2)$.

v. If $\delta \in Decl$ and $\varphi \in Form$, then $\boldsymbol{\exists\delta(\varphi)} \in Form$ with $\text{free}(\exists\delta(\varphi)) := (\text{free}(\delta) \cup \text{free}(\varphi)) \setminus \text{decl}(\delta)$.

For a fixed interpretation I and a variable assignment $\alpha \in A$ the validity of formulas is defined as follows.

i. $(I, \alpha) \models \pi(\tau_1, \ldots, \tau_n)$ iff $((I, \alpha)[\![\tau_1]\!], \ldots, (I, \alpha)[\![\tau_n]\!]) \in I(\pi)$.

ii. $(I, \alpha) \models \tau_1 = \tau_2$ iff $(I, \alpha)[\![\tau_1]\!] = (I, \alpha)[\![\tau_2]\!]$.

iii. $(I, \alpha) \models \neg(\varphi)$ iff not $(I, \alpha) \models \varphi$.

iv. $(I, \alpha) \models (\varphi_1 \vee \varphi_2)$ iff $(I, \alpha) \models \varphi_1$ or $(I, \alpha) \models \varphi_2$.

v. $(I, \alpha) \models (\exists\delta(\varphi))$ iff there is a variable assignment $\alpha' \in A$ with $\alpha'(v) = \alpha(v)$ for all $v \in Var \setminus \text{decl}(\delta)$ and $(I, \alpha') \models \varphi$ and $(I, \alpha') \models \delta$.

□

There is nothing special about formulas. Note that the constant formulas `TRUE` and `FALSE`, the other logical connectives $\wedge$, $\Rightarrow$, $\Leftrightarrow$, and the quantifier $\forall$ might be defined by the given material.

Example (formulas in concrete syntax). `x<4` (infix notation of predicate symbols is frequently used), `x=y`, `NOT x=y`, `x=y AND x=z OR y=z` (logical connectives have the usual bindings), and `EXISTS p:PERSON WHERE Name(p)="Smith"` are possible formulas.

The function free returns the free variables of a formula.

Remark (relation between quantified formulas and select terms). There exists a notable correspondence between quantifiers and the from- and where-clause of select terms. With the aggregate function CNT it is always possible to replace the formula $\exists\delta(\varphi)$ by $\texttt{CNT}\{\!\!\{\, c \mid \delta, \varphi \,\}\!\!\} \geq 1$ (in which c represents an arbitrary constant term). In this respect quantifiers can be regarded as syntactic sugar in our query formalism [Gog94].

Definition 7 (declarations). The syntax of declarations is given by a set *Decl* and the functions free, decl : $\mathit{Decl} \rightarrow \mathcal{F}(\mathit{Var})$ defined by the following rules.

i. If $v \in \mathit{Var}_s$, $\tau \in \mathit{Term}_{\mathrm{set}(s)}$ and $v \notin \mathrm{free}(\tau)$, then $(\boldsymbol{v} : \boldsymbol{\tau}) \in \mathit{Decl}$, $\mathrm{free}((v : \tau)) := \mathrm{free}(\tau)$ and $\mathrm{decl}((v : \tau)) := \{v\}$.

ii. If $v \in \mathit{Var}_s$, $\tau \in \mathit{Term}_{\mathrm{set}(s)}$, $\delta \in \mathit{Decl}$ and $v \notin \mathrm{free}(\tau) \cup \mathrm{free}(\delta) \cup \mathrm{decl}(\delta)$, then $(\boldsymbol{v} : \boldsymbol{\tau};\ \boldsymbol{\delta}) \in \mathit{Decl}$, $\mathrm{free}((v : \tau;\ \delta)) := (\mathrm{free}(\tau) \setminus \mathrm{decl}(\delta)) \cup \mathrm{free}(\delta)$ and $\mathrm{decl}((v : \tau;\ \delta)) := \{v\} \cup \mathrm{decl}(\delta)$.

For a fixed interpretation I and a variable assignment $\alpha \in A$ the validity of declarations is defined as follows.

i. $(I, \alpha) \models (v : \tau)$ iff $\alpha(v) \in (I, \alpha)[\![\, \tau \,]\!]$.

ii. $(I, \alpha) \models (v : \tau;\ \delta)$ iff $\alpha(v) \in (I, \alpha)[\![\, \tau \,]\!]$ and $(I, \alpha) \models \delta$.

□

Following the definition declarations are generally given as *declaration sequences* of the kind $(x_1{:}\tau_1;\ (x_2{:}\tau_2;\ \ldots;\ (x_n{:}\tau_n)\ \ldots\ ))$. Declarations are used in select terms or in quantified formulas to bind variables to finite sets of values.

Example (declarations in concrete syntax). `x:{1,2,3}`, `p:PERSON` (in which `PERSON` denotes a set-valued term), `p:PERSON; c:CAR`, and `h:Hobbies(p); p:PERSON` (in which `Hobbies(p)` denotes a set-valued attribute of persons) are possible declarations.

For a declaration δ the function free returns the free variables of δ and the function decl gives the set of variables declared within δ. For a declaration sequence $(x_1{:}\tau_1;\ (x_2{:}\tau_2;\ \ldots;\ (x_n{:}\tau_n)))$ this is $\{x_1, \ldots, x_n\}$. In such a declaration sequence a variable x_i must not be declared twice and is not allowed to be free to the right, i.e., to be free in τ_j with $j \geq i$.

Finally we come to the central notion of queries and constraints. Queries are defined to be closed terms, i.e., terms without free variables, and constraints are given by closed formulas.

Definition 8 (queries and constraints).

1. Every term $\tau \in \mathit{Term}$ with $\mathrm{free}(\tau) = \emptyset$ defines a query. The evaluation of queries is determined by $(I, \epsilon)[\![\, \tau \,]\!]$.

2. Every formula $\varphi \in \mathit{Form}$ with $\mathrm{free}(\varphi) = \emptyset$ defines a constraint. The validity of constraints is determined by $(I, \epsilon) \models \varphi$.

□

In the following brackets are dropped as far as unambiguity is preserved. Infix notations are used where appropriate. $\{\!\!\{\, \tau \mid \delta \,\}\!\!\}$ is used as a short-hand for $\{\!\!\{\, \tau \mid \delta,\ \texttt{TRUE} \,\}\!\!\}$.

3.2 Evaluation of select terms

According to Definition 5 a select term of the general kind

$$\{\!\!\{\ \tau \mid x_1{:}\tau_1;\ x_2{:}\tau_2;\ \ldots;\ x_n{:}\tau_n,\ \varphi\ \}\!\!\}$$

is evaluated by examining all possible assignments of values to the variables $x_1, \ldots, x_n$. Variable assignments have to fulfill the declarations and the qualifying formula φ. Having found a possible assignment this assignment contributes to the query result by evaluating the term τ. Assuming all terms τ_i $(1 \leq i \leq n)$ to be evaluated to finite sets there can only exist a finite number of possible assignments to variables $x_1, \ldots, x_n$. Hence evaluation of select terms always results in finite sets. The query formalism is *safe* [Ull88] (which can be formally proved by induction over term construction [GH91]). Importantly there is no need to define safe queries in a semantic way as in the case of relational domain calculus.

There is a simple scheme of describing the semantics of a declarative select term by an imperative program. More precisely a select term of the above kind can be evaluated by the following procedure:

result := $\{\!\!\{\ \}\!\!\}$;
for each $x_n \in [\![\ \tau_n\ ]\!]$ **do**
 ...
  for each $x_1 \in [\![\ \tau_1\ ]\!]$ **do**
   if $\models \varphi$ **then** result := result $\cup$ $[\![\ \tau\ ]\!]$ **fi**;
  od;
 ...
od;

The order of the for-each statements reflects the scope of variables in declaration sequences extending from the right to the left.

Next we try to explain the semantics of select terms by giving some examples.

Examples.

1. The first example demonstrates the basic *filter* function of select terms.

   $\{\!\!\{$ `x | x:{-2,-1,0,1,2}, x>0` $\}\!\!\}$

   The term is evaluated to $\{\!\!\{\ \mathit{1}, \mathit{2}\ \}\!\!\}$.

2. In general select terms are evaluated to true bags. For instance, the term

   $\{\!\!\{$ `x^2 | x:{-2,-1,0,1,2}` $\}\!\!\}$

   results in $\{\!\!\{\ \mathit{0}, \mathit{1}, \mathit{1}, \mathit{4}, \mathit{4}\ \}\!\!\}$. The aggregate function `BTS` can be used to convert bags into sets by eliminating duplicates.

3. Declarations may be given as declaration sequences. This can be used to describe the *cross product*. For example

   $\{\!\!\{$ `(x,y) | x:{1,2}; y:{1,2,3}` $\}\!\!\}$

   is evaluated to $\{\!\!\{\ (\mathit{1},\mathit{1}),\ (\mathit{1},\mathit{2}),\ (\mathit{1},\mathit{3}),\ (\mathit{2},\mathit{1}),\ (\mathit{2},\mathit{2}),\ (\mathit{2},\mathit{3})\ \}\!\!\}$. Together with a qualifying formula φ this allows the formulation of *joins*.

4. In a declaration sequence like $\mathrm{x}_1{:}\tau_1;\ldots;x_n{:}\tau_n$ the variable x_i is allowed to be free in $\tau_1, \ldots, \tau_{i-1}$. This can be used to express the *union* of sets being demonstrated by

```
⦃ x | x:y; y:{{1,2}, {2,3}} ⦄
```

which is evaluated to $\{\!\{\, 1, 2, 2, 3 \,\}\!\}$. The following table shows all possible variable assignments together with the corresponding evaluations of the target term.

variable assignment		`x`
y = $\{1, 2\}$	x = 1	1
y = $\{1, 2\}$	x = 2	2
y = $\{2, 3\}$	x = 2	2
y = $\{2, 3\}$	x = 3	3

By applying the function `BTS` the result can be converted into a proper set.

5. With nested select terms *groupings* can be expressed. This is shown by the next example which groups the flat value $\{(1, 2), (1, 3), (2, 3)\}$ by the first component resulting in $\{(1, \{2, 3\}), (2, \{3\})\}$ (x.i is used as an abbreviation for $\texttt{PRJ}^i$(x)).

```
BTS⦃ (x.1,BTS⦃ y.2 | y:{(1,2),(1,3),(2,3)}, y.1=x.1 ⦄) |
      x:{(1,2),(1,3),(2,3)} ⦄
```

variable assignment		`y.1=x.1`	`(x.1,BTS⦃...⦄)`
x = $(1, 2)$	y = $(1, 2)$	true	
x = $(1, 2)$	y = $(1, 3)$	true	$(1, \{2, 3\})$
x = $(1, 2)$	y = $(2, 3)$	false	
x = $(1, 3)$	y = $(1, 2)$	true	
x = $(1, 3)$	y = $(1, 3)$	true	$(1, \{2, 3\})$
x = $(1, 3)$	y = $(2, 3)$	false	
x = $(2, 3)$	y = $(1, 2)$	false	
x = $(2, 3)$	y = $(1, 3)$	false	$(2, \{3\})$
x = $(2, 3)$	y = $(2, 3)$	true	

6. The result of the last select term can be *unnested* again by the select term

```
BTS⦃ (x.1,y) | y:x.2; x:{(1,{2,3}), (2,{3})}⦄
```

yielding the original value.

variable assignment		`(x.1,y)`
x = $(1, \{2, 3\})$	y = 2	$(1, 2)$
x = $(1, \{2, 3\})$	y = 3	$(1, 3)$
x = $(2, \{3\})$	y = 3	$(2, 3)$

3.3 A short survey of prominent query calculi

Relational domain calculus. Relational domain calculus has been considered as the formal basis of concrete query languages like QBE. A query in domain calculus is given by a formula $\varphi(x_1, \ldots, x_n)$ in which $x_1 \ldots x_n$ denote the free variables of φ. Comparing relational domain calculus with QSV one will find that there is an important difference in the usage of variables: (i) In relational domain calculus variables are restricted to be of atomic data sorts (domains). (ii) There is no notion of variable declaration, i.e., variables need not be range-restricted. Because all variables being of atomic data sorts the answer of a query $\varphi(x_1, \ldots, x_n)$ in relational domain calculus always yields a new relation defined by $\{(x_1, \ldots, x_n) \mid \varphi(x_1, \ldots, x_n)\}$. Existing relations can be referred in queries by treating them as predicates.

Example. Let a relational schema `PERSON(Name:string, Age:nat, Income:nat)` be given. Then the following query in relational domain calculus

$$\varphi(\mathrm{n}, \mathrm{i}) \equiv \exists \mathrm{a}\ (\mathrm{PERSON}(\mathrm{n}, \mathrm{a}, \mathrm{i}) \wedge \mathrm{a} < 18)$$

returns the set of pairs of name and income of persons with age less than 18.

Relational tuple calculus. Relational tuple calculus, which has been designated as the formal basis for query languages like SQL and QUEL, is very similar to relational domain calculus with the exception that domain variables are replaced by tuple variables which must be bound to relational schemas. Components of tuple variables are referred by the dot operator (projection).

Example. The same query as above is formulated in tuple calculus by

$$\begin{aligned}&\varphi(\mathrm{t} : (\mathrm{Name} : \mathrm{string}, \mathrm{Income} : \mathrm{nat})) \equiv \\ &\qquad \exists \mathrm{p{:}PERSON}\ (\mathrm{p.Name} = \mathrm{t.Name} \wedge \mathrm{p.Income} = \mathrm{t.Income} \wedge \mathrm{p.Age} < 18)\end{aligned}$$

It is a well-known fact that relational domain and tuple calculus are of same expressive power. Although tuple calculus is often referred to be the formal basis of SQL, it should be noted that many facilities of SQL, for instance the possibility to obtain duplicates in query results or grouping cannot be adequately mirrored in tuple calculus.

The calculus of Abiteboul/Beeri. In query calculi for extended relational models the requirement that answers to queries must result in new relations is obsolete [KV84, BK86, RKS88, ABGG89]. In order to allow for answers of any structure, in an approach to a query calculus for Complex Object models proposed by Abiteboul and Beeri called **CALC** [AB88] the restriction forcing variables to be of atomic data sorts (or tuple expressions built over these sorts) is dropped. In other words this means a (at least syntactical) move from first-order to higher-order predicate logic.

Example. The following CALC query returns the set of all groups of names which can be built from names found in the PERSON relation (powerset).

$$\varphi(\mathrm{t}) \equiv \forall \mathrm{n}\ (\mathrm{n} \in \mathrm{t} \Rightarrow \exists \mathrm{a}, \mathrm{i}\ (\mathrm{PERSON}(\mathrm{n}, \mathrm{a}, \mathrm{i})))$$

The target variable t is of sort set(string). Hence the answer to this query is of sort set(set(string)).

Range-restricted variants of the presented calculi. One problem associated with all the presented calculi is that they allow to define *unsafe* queries, i.e., queries showing infinite answers.

Example. The query $\varphi(\mathrm{n}, \mathrm{a}, \mathrm{i}) \equiv \neg\mathrm{PERSON}(\mathrm{n}, \mathrm{a}, \mathrm{i})$ returns all combinations of values for Name, Age, and Income not found in the database.

The problem gets even worse when data operations or predicates are allowed to be used in calculus expressions. As shown by the following example this may result in the formulation of *uncomputable* queries.

$\cup$	$\mathrm{set}(s) \times \mathrm{set}(s) \to \mathrm{set}(s)$	$\{\!\!\{$ x | x:y; y:$\{\tau_1, \tau_2\}$ $\}\!\!\}$
$-$	$\mathrm{set}(s) \times \mathrm{set}(s) \to \mathrm{set}(s)$	$\{\!\!\{$ x | x:τ_1, $\neg$IN(τ_2, x) $\}\!\!\}$
$\times$	$\mathrm{set}(s_1) \times \mathrm{set}(s_2) \to \mathrm{set}(\mathrm{tuple}(s_1, s_2))$	$\{\!\!\{$ (x,y) | x:τ_1; y:τ_2 $\}\!\!\}$
π_i	$\mathrm{set}(\mathrm{tuple}(s_1, \ldots, s_n)) \to \mathrm{set}(s_i)$	$\{\!\!\{$ x.i | x:τ $\}\!\!\}$
σ_φ	$\mathrm{set}(s) \to \mathrm{set}(s)$	$\{\!\!\{$ x | x:τ, φ $\}\!\!\}$
ν	$\mathrm{set}(\mathrm{tuple}(s_1, s_2)) \to \mathrm{set}(\mathrm{tuple}(s_1, \mathrm{set}(s_2)))$	$\{\!\!\{$ (x.1, $\{\!\!\{$ y.2 | y:τ, y.1 = x.1$\}\!\!\}$) | x:τ $\}\!\!\}$
μ	$\mathrm{set}(\mathrm{tuple}(s_1, \mathrm{set}(s_2))) \to \mathrm{set}(\mathrm{tuple}(s_1, s_2))$	$\{\!\!\{$ (x.1,y) | y:x.2; x:τ $\}\!\!\}$
ν'	$\mathrm{set}(s) \to \mathrm{set}(\mathrm{set}(s))$	$\{\!\!\{$ $\{\!\!\{$ y | y:τ, y = x $\}\!\!\}$ | x:τ $\}\!\!\}$
μ'	$\mathrm{set}(\mathrm{set}(s)) \to \mathrm{set}(s)$	$\{\!\!\{$ y | y:x; x:τ $\}\!\!\}$

Table 1: Standard operations on structured values[3]

Example. The query $\varphi(\mathrm{x}, \mathrm{y}, \mathrm{z}) \equiv \exists \mathrm{n}\ (\mathrm{n} > 2 \wedge \mathrm{x}^{\mathrm{n}} + \mathrm{y}^{\mathrm{n}} = \mathrm{z}^{\mathrm{n}})$ shows as answer the set of all tuples disproving Fermat's conjecture.

There are two ways in the direction of solving the problem of unsafe and uncomputable queries. One obvious solution is to assume all atomic data sorts (domains) to be of finite range, e.g. $I(\mathrm{int}) = \{\mathrm{minint}, \ldots, \mathrm{maxint}\}$. The other, perhaps more accepted approach to make formulas safe is to require that each variable is attached to a *constant, stored,* or *computed* finite range of values.

It is clear that the range-restricted variants of the presented calculi are generally of less expressive power than the unrestricted versions. For example, the range-restricted calculus $\mathbf{CALC}^-$ lacks the power to compute the powerset of a finite set while CALC or a weaker restricted variant admits to express the powerset operation (see below).

3.4 Expressive Power of QSV

Comparing QSV with query algebras. QSV allows to express all standard operations on structured values. This is shown in Table 1. The first group of operations subsumes the classical operations of relational algebra. In the second group ν denotes the *nest* and μ the *unnest* operator of [SS86]. Specific operations for CO models can be expressed as well. For example, *set construction* ν' and *set collapse* μ' can be derived from the corresponding ν and μ representations by removing s_1 from the corresponding tuple expressions and renaming s_2 to s, thereby changing $\mathrm{set}(\mathrm{tuple}(s_1, s_2))$ to $\mathrm{set}(s)$ and $\mathrm{set}(\mathrm{tuple}(s_1, \mathrm{set}(s_2)))$ to $\mathrm{set}(\mathrm{set}(s))$ respectively.

Since the result of each query defines a finite set which can be made part of further queries every algebra term can be equivalently expressed by a term in QSV. The expressive limit is reached when powerset is to be described [GG92] (see below).

Comparing QSV with query calculi. QSV stands in the tradition of query calculi proposed for structured values [KV84, BK86, RKS88, ABGG89] from which the calculus of Abiteboul and Beeri (denoted CALC), which was studied in more detail in [AB88], can be considered as an archetype. With regard to CALC we find the following distinguishing features of QSV:

1. The consideration of *non-logical components* (data functions, aggregate functions) as important ingredients of concrete query languages.
2. The treatment of *duplicates* in the evaluation of select terms.
3. The *non-hierarchical structure* of the query formalism guided by the idea of supporting fully orthogonal SQL.

[3] In every calculus expression τ_i stands for the ith argument of the corresponding operation. The dot notation is used for projection. To allow comparison, here all select terms are assumed to evaluate to proper sets by implicit duplicate elimination.

We briefly discuss the three points in more detail.

ad 1. The incorporation of externally defined functions in the query formalism clearly does not contribute to its inherent computational power. This was the main reason why such functions were not considered in [AB88]. On the other hand the pure calculus, even augmented with a fixpoint operator for expressing recursive queries (see below on expressiveness), still fails in solving such a trivial problem like testing a given finite set for even cardinality (*even problem* [AV92]). It has been argued in [AU79] that the role of a query language should be primarily the selection and combination of data from a database, rather than arithmetic or other general computation on this data. So it should not be considered harmful when a query language does not show full computational power. Lack of expressive power can always be compensated by a careful combination of a query language with a general purpose programming language. Of course, the main problem still consists in overcoming the well-known *impedance mismatch* between the set-oriented type system of a query language and the record-oriented style of conventional programming languages.

ad 2. Analogously to relational calculi a query in CALC is simply given by a formula φ with a free variable, for instance $\exists \mathrm{x}\ (\mathrm{x} \in \{-2,-1,0,1,2\} \wedge \mathrm{x}^2 = \mathrm{t})$ (compare with QSV query 2 in Section 3.2). The answer to this query is the set of all possible assignments of values to t making the formula φ true (here $\{\mathit{0}, \mathit{1}, \mathit{4}\}$). Since the definition of query result does not take into account different assignments of values to x, making the formula true for a given value t, one does not obtain duplicates in query results.

ad 3. Since queries in CALC are given as usual formulas it is nearly impossible to structure complex queries. The decreased readability of CALC queries in comparison with SQL-like queries has been criticized in [Bee90]. It follows from the fact that the only way to deal with subqueries in CALC consists in using *set variables*. Let a query $\varphi(t)$ be given. Then the result of this query may be used in another query by referring to a set variable s determined by $\forall t\ (t \in s \Leftrightarrow \varphi(t))$. On the other side, since in QSV every select term is nothing else than a special term, select terms may appear in any place as subterms of other select terms achieving full orthogonality for composing queries from subqueries. The effect on the style how queries are formulated in both formalisms is highlighted by an example given below.

Although CALC and QSV differ considerably in the style how queries are formulated against databases, leaving data functions and bag handling (duplicates in query results) out of consideration, the following equivalence result can be stated:

Proposition (equivalence of QSV and CALC$^-$). Assume that queries in QSV and CALC$^-$ make use of the same (externally defined) data operations and predicates like set construction $\{\ \}$, tuple construction $(\ )$, the membership predicate $\in$, and projection (dot operator). Let duplicate handling in QSV be neglected by implicit duplicate elimination. Then every query in QSV can be expressed by a query in CALC$^-$ and vice versa. $\square$

Sketch of proof. Translation from QSV expressions to CALC expressions can be done in three steps.

Step 1: The first transformation is described by a function $\langle . \rangle$ which takes a QSV expression as input and maps it two an intermediate presentation. This presentation is already very similar to CALC expressions with the exception that it may still contain some set expressions of the kind $\{t \mid \varphi(t)\}$.

1. Terms (see Def. 5):

   i. $\langle v \rangle := v$

ii. $\langle \omega(\tau_1, \ldots, \tau_n) \rangle := \omega(\langle \tau_1 \rangle, \ldots, \langle \tau_n \rangle)$

iii. $\langle \{\!\!| \, \tau \,|\, \delta,\ \varphi \,|\!\!\} \rangle := \{t \,|\, \langle \delta \rangle \wedge \langle \varphi \rangle \wedge t = \langle \tau \rangle\}$ in which t denotes any new variable.

2. Formulas (see Def. 6):

   i. $\langle \pi(\tau_1, \ldots, \tau_n) \rangle := \pi(\langle \tau_1 \rangle, \ldots, \langle \tau_n \rangle)$

   ii. $\langle \tau_1 = \tau_2 \rangle := \langle \tau_1 \rangle = \langle \tau_2 \rangle$

   iii. $\langle \neg(\varphi) \rangle := \neg(\langle \varphi \rangle)$

   iv. $\langle (\tau_1 \vee \tau_2) \rangle := (\langle \tau_1 \rangle \vee \langle \tau_2 \rangle)$

   v. $\langle \exists \delta(\varphi) \rangle := (\langle \delta \rangle \wedge \langle \varphi \rangle)$

3. Declarations (see Def. 7):

   i. $\langle (v : \tau) \rangle := \exists v \ v \in \langle \tau \rangle$

   ii. $\langle (v : \tau; \delta) \rangle := \langle \delta \rangle \wedge \exists v \ v \in \langle \tau \rangle$

Step 2: Make the scope of the variables declared in (3) explicit. For select terms (1.iii) the scope of the declared variables extends to the right brace. In quantified formulas (2.v) the scope of the declared variables is the formula φ.

Step 3: Every set expression of the kind $\{t \,|\, \varphi(t)\}$ can be replaced by a set variable s being characterized by a preceding formula $\exists s \ \forall t \ (t \in s \Leftrightarrow \varphi(t))$. The scope of s must be extended to the expression where the original set expression occured.

Example. In the first step the following QSV query which refers to the grouping example (query 5) of the QSV queries in Section 3.2

$$\{\!\!| \, (\mathrm{x.1}, \{\!\!| \, \mathrm{y.2} \,|\, \mathrm{y : R}, \mathrm{y.1 = x.1} \,|\!\!\}) \,|\, \mathrm{x : R} \,|\!\!\}$$

is translated to

$$\{\mathrm{t_1} \,|\, \exists \mathrm{x} \ \mathrm{x} \in \mathrm{R} \wedge \mathrm{t_1} = (\mathrm{x.1}, \{\mathrm{t_2} \,|\, \exists \mathrm{y} \ \mathrm{y} \in \mathrm{R} \wedge \mathrm{y.1} = \mathrm{x.1} \wedge \mathrm{t_2} = \mathrm{y.2}\})\}.$$

Making the scope of variables explicit yields

$$\{\mathrm{t_1} \,|\, \exists \mathrm{x} \ (\mathrm{x} \in \mathrm{R} \wedge \mathrm{t_1} = (\mathrm{x.1}, \{\mathrm{t_2} \,|\, \exists \mathrm{y} \ (\mathrm{y} \in \mathrm{R} \wedge \mathrm{y.1} = \mathrm{x.1} \wedge \mathrm{t_2} = \mathrm{y.2})\}))\}.$$

Replacing the inner set expression by a set variable yields

$$\begin{aligned}\{\mathrm{t_1} \,|\, & \ \exists \mathrm{x} \ (\mathrm{x} \in \mathrm{R} \wedge \\ & \exists \mathrm{s_2} \ (\forall \mathrm{t_2} \ (\mathrm{t_2} \in \mathrm{s_2} \Leftrightarrow \exists \mathrm{y} \ (\mathrm{y} \in \mathrm{R} \wedge \mathrm{y.1} = \mathrm{x.1} \wedge \mathrm{t_2} = \mathrm{y.2})) \wedge \mathrm{t_1} = (\mathrm{x.1}, \mathrm{s_2})))\}.\end{aligned}$$

giving a query φ in CALC.

In CALC$^-$ a formula is called *safe* if all its variables are range-restricted. A variable is called *range-restricted* if there is a corresponding *range formula* $\Phi(t)$. The left column of Table 2 summarizes all the possible syntactic situations, i.e., the cases for a free, existentially bound, and universally bound variable t [ABGG89].

Range formulas in CALC correspond to declarations in QSV although the definition is somehow more involved. Range formulas are defined inductively. The first two rows in the first column of Table 3 give the basis, the next four some inductive steps, and the last row the closure of the construction [ABGG89].

Following the given translation scheme every closed QSV select term or formula is transformed into a safe CALC$^-$ formula. On the other side, Tables 2 and 3 show corresponding representations of safe formulas and range formulas in QSV (here $\langle \Phi(s) \rangle$ always denotes the QSV declaration belonging to a CALC$^-$ formula $\Phi(s)$), showing that every safe formula in CALC$^-$ can be equivalently expressed by a term or formula in QSV.

Towards more expressive power. There exists a variant of CALC$^-$ which in addition to the range formulas of Table 3 also admits range formulas of the kind $\exists s\ (\Phi(s) \wedge t \subseteq s)$. This allows to compute the powerset of a given set. In order to give a corresponding declaration in QSV one had to include a special powerset construction, either declared externally as a generic operation or as a built-in construct of the query formalism [GH91]. The corresponding declaration would be $t : \mathtt{POW}(\{\!\!\{ s \mid \langle\Phi(s)\rangle \}\!\!\})$.

Notably powerset allows to compute the transitive closure of a relation R:set(tuple(s, s)). Let $\mathrm{S} := \pi_1(\mathrm{R}) \cup \pi_2(\mathrm{R})$ and $\mathrm{B} := \mathrm{S} \times \mathrm{S}$. Then the following QSV query

$$\{\!\!\{ \mathrm{y} \mid \mathrm{y} : \mathrm{B}, \forall(\mathrm{x} : \mathtt{POW}(\mathrm{B}))\ (\mathrm{closed}(\mathrm{x}) \wedge \mathrm{x} \supseteq \mathrm{R} \Rightarrow \mathrm{y} \in \mathrm{x}) \}\!\!\}$$

in which closed(x) is defined by $\forall \mathrm{u} : \mathrm{B}\ \forall \mathrm{v} : \mathrm{B}\ \mathrm{u}.2 = \mathrm{v}.1 \Rightarrow (\mathrm{u}.1, \mathrm{v}.2) \in \mathrm{x}$ would compute the transitive closure of R. The idea of this computation is to describe transitive closure at the smallest that is closed under transitivity and includes relation R.

Transitive closure can be more easily described by *recursion* in a rule-based framework. Hence another idea is to include a *fixpoint operator* μ into the calculus (see [KC93] for a concrete language proposal). One can show that first-order calculus with inflationary fixpoint operator is equivalent to algebra with inflationary *while* and Datalog with negation (see [AV88, AV92] for details).

A more pragmatic approach is based on an extensible signature Σ. Whenever a special operator like powerset $\mathtt{POW}_{\mathrm{set}(s)} : \mathrm{set}(s) \rightarrow \mathrm{set}(\mathrm{set}(s))$ or transitive closure $\mathtt{TC}_{\mathrm{set}(\mathrm{tuple}(s,s))} : \mathrm{set}(\mathrm{tuple}(s,s)) \rightarrow \mathrm{set}(\mathrm{tuple}(s,s))$ is needed this can always be defined *externally* by a corresponding generic function. For instance, all operators found in Table 1 could also find their entrance into the query calculus as generic functions on structured values (cf. Section 2.3). In some cases this can considerably assist the formulation of complex queries (take the union of sets in the last row of Table 3 or the QSV realization of transitive closure as an example).

4 Application to Database States

The query and constraint formalism of Section 3 was developed independently of any concrete database model. Queries were formulated over data constants instead. Clearly in real applications we want to apply the formalism to stored values. Therefore in this section we first discuss the application to existing database models. Secondly we give some insight into an approach to object-oriented specification where the query and constraint formalism is used in the formulation of axioms expressing object properties.

4.1 Queries in value- and object-based data models

A database schema generally includes a number of *containers* for storing objects that share common structure and common behavior. In value-based data models these containers are given by *relations* (or *relational schemas*); in object-based data models they are given by *classes*. The fundamental difference between value-based and object-based data models lies in their particular approach to the representation

safe formula in CALC$^-$	corresponding construct in QSV
$\Phi(t) \wedge \Psi$	$\{\!\!\{ t \mid \langle\Phi(t)\rangle, \langle\Psi\rangle \}\!\!\}$
$\exists t\ (\Phi(t) \wedge \Psi)$	$\exists\ (\langle\Phi(t)\rangle)\ (\langle\Psi\rangle)$
$\forall t\ (\Phi(t) \Rightarrow \Psi)$	$\neg\exists\ (\langle\Phi(t)\rangle)\ (\neg\langle\Psi\rangle)$

Table 2: Safe formulas in CALC$^-$ and their representation in QSV

range formula for t in CALC^-	corresponding declaration in QSV
$t \in c$ (c constant/stored set value)	$t : c$
$t = c$ (c constant/stored value)	$t : \{c\}$
$t = \{s \mid \Phi(s) \wedge \Psi\}$	$t : \{\{\!\mid s \mid \langle\Phi(s)\rangle, \langle\Psi\rangle \mid\!\}\}$
$\exists s\ (\Phi(s) \wedge t = s.A)$	$t : \{\!\mid s.A \mid \langle\Phi(s)\rangle \mid\!\}$
$\exists s\ (\Phi(s) \wedge t \in s)$	$t : \{\!\mid x \mid x : s;\ \langle\Phi(s)\rangle \mid\!\}$
$\exists s_1 \ldots \exists s_n\ (\Phi_1(s_1) \wedge \ldots \wedge \Phi_k(s_k) \wedge$ $t.1 = s_1 \wedge \ldots \wedge t.k = s_k)$	$t : \{\!\mid (s_1, \ldots, s_k) \mid \langle\Phi(s_1)\rangle; \ldots; \langle\Phi(s_k)\rangle \mid\!\}$
$\Phi_1(t) \vee \Phi_2(t)$	$t : \{\!\mid t \mid \langle\Phi_1(t)\rangle \mid\!\} \cup \{\!\mid t \mid \langle\Phi_2(t)\rangle \mid\!\}$

Table 3: Range formulas in CALC^- and their representation in QSV

value-based	relation $r(a_1{:}s_1, \ldots, a_n{:}s_n)$	$r \in \mathit{Term}_{\mathrm{set(tuple}(s_1,\ldots,s_n))}$
object-based	class c with $\mathrm{osort}(c) = o$	$c \in \mathit{Term}_{\mathrm{set}(o)}$
	attribute $a : o \longrightarrow s$	$a \in \Omega$

Table 4: Including containers into the query formalism

of real-world entities [Hul89]. While in value-based models a real-world entity is represented directly as a tuple of its attribute values (*internal structure*), object-based data models spend a special object identifier (surrogate) for this purpose and attributes are functions on object identifiers (*attribute structure*). The latter approach enables updates on objects.

Referring to stored values in queries is based on references to containers. Hence a simple way of extending the query and constraint formalism to database states consists in including containers as set-valued terms. In addition, in object-based data models attributes must be admitted as functions on object surrogates. This is summarized in Table 4.

The first line of the table also holds for the extensions of the relational model mentioned in Section 2.2. In the object-based approach we strictly distinguish between a class c and a sort of object identifiers $\mathrm{osort}(c)$ belonging to a class c although in many data models same names are used to denote both classes and corresponding object identifier sorts.

Queries in the relational data model. In order to formulate queries in the relational data model, the definition of terms (Def. 5) is extended by:

- If $r(a_1{:}s_1, \ldots, a_n{:}s_n)$ is a relational schema, then $r \in \mathit{Term}_{\mathrm{set(tuple}(s_1,\ldots,s_n))}$.

In addition to the interpretation structure I and the current variable assignment α now the evaluation of terms further depends on the current state σ of a relational database, i.e., a relational schema r is evaluated to a finite set $\sigma(r)$ of current tuple values.

- $(I, \alpha, \sigma)[\![\, r \,]\!] = \sigma(r)$

Example. Let a relational schema `PERSON(Name:string, Age:nat)` be given. Then the following query returns the names of all stored persons:

$\{\!\mid$ `Name(p) | p:PERSON` $\mid\!\}$

or in concrete syntax:

```
SELECT Name(p) FROM p:PERSON
```

In this query p is a variable of sort tuple(Name:string, Age:nat) so that Name acts as an projection operator.

One often cited requirement of a query language is that it has to be *closed* with respect to the underlying data model. Since QSV was developed for a more general model of structured values in the relational data model we generally have to restrict the sort of the target term τ in select-terms to expressions of the kind tuple$(d_1, \ldots, d_n)$ where d_i denote atomic data sorts[4].

Queries in object-based data models. Object-based data models should support objects with complex attributes [Ban88]. For those models QSV can be adopted without any restriction.

To formulate queries in object-based data models, the definition of terms (Def. 5) is extended by:

- If c is a class with $\mathrm{osort}(c) = o$, then $c \in \mathit{Term}_{\mathrm{set}(o)}$.
- If $a : o \rightarrow s$ and $\tau \in \mathit{Term}_o$, then $a(\tau) \in \mathit{Term}_s$.

The evaluation of these terms depends again on the current database state σ, which associates a finite set $\sigma(c)$ of current object identifiers with a class c and an attribute value $\sigma(a)(\underline{o})$ with each object identifier $\underline{o}$.

- $(I, \alpha, \sigma)[\![\, c \,]\!] = \sigma(c)$.
- $(I, \alpha, \sigma)[\![\, a(\tau) \,]\!] = \sigma(a)((I, \alpha, \sigma)[\![\, \tau \,]\!])$.

Examples. Let us replace the relational schema by a class `PERSON` with attributes `Name:string` and `Age:nat`.

1. Compared with the relational formulation a query returning the names of all stored persons is again given by

   ```
   -{ Name(p) | p:PERSON }-
   ```

   But in contrast to the relational query here p is a variable of object identifier sort person = osort(`PERSON`) and Name is used as an attribute function. Note that both select terms are of sort bag(string) reflecting the fact that a given name may appear more than once in a query result. Note also that in the object-based database it is even possible that two persons with the same name and age exist while this is excluded in a corresponding relational database without duplicates.

2. The following query gives an example of a select term with a nested select in the select-clause (grouping).

   ```
   BTS-{ (p1, BTS-{ p2 | p2:PERSON, Age(p2)>Age(p1) }-) | p1:PERSON }-
   ```

   or in concrete syntax:

   ```
   BTS(SELECT (p1, BTS(SELECT p2
                       FROM   p2:PERSON
                       WHERE  Age(p2)>Age(p1)))
       FROM   p1:PERSON)
   ```

[4] One direct consequence of this restriction is that grouping can no longer be expressed in our formalism. Indeed the `GROUP BY` clause of relational SQL is nothing else than a trick to count with intermediate NF^2 relations in a pure 1NF setting.

The query results in a value of sort set(tuple(person, set(person))). It contains all persons together with the older ones.

3. The next query gives an example of a nested select in the where-clause of a select term.

```
BTS⦃ x | x:PERSON, Age(x)=MAX⦃ Age(y) | y:PERSON ⦄ ⦄
```

   or in concrete syntax:

```
BTS(SELECT x
    FROM   x:PERSON
    WHERE  Age(x)=MAX(SELECT Age(y)
                      FROM   y:PERSON))
```

   The query makes use of an aggregate function `MAX` : bag(nat) $\rightarrow$ nat. It returns the set of the oldest persons. Note that in concrete syntax we did not write `SELECT MAX(Age(y))` as it would have been done in standard SQL showing some peculiar inconsistency of this standard [Dat84].

   The same query could also be formulated without making use of an aggregate function by:

```
BTS⦃ x | x:PERSON, ∀ y:PERSON Age(y)<=Age(x) ⦄
```

4. Let <Q3> denote the query given in item 3. Then the following query returns the names of the oldest persons.

```
BTS⦃ Name(p) | p:<Q3> ⦄
```

   This also gives an example for a query with nested select in the from-clause of a select term.

5. Let `Hobbies:SET(string)` be a further attribute of `PERSON`. Then the following query which makes use of a declaration sequence returns the set of hobbies of persons with age less than 18.

```
BTS⦃ h | h:Hobbies(p); p:PERSON, Age(p)<18 ⦄
```

   or in concrete syntax:

```
BTS(SELECT h
    FROM   h:Hobbies(p); p:PERSON
    WHERE  Age(p)<18)
```

Queries in object-based data models with taxonomic structure on classes. Up to now we have not paid attention to inheritance relations between classes. In order to capture inheritance we have to deal with polymorphic functions.

Example. Let `EMPLOYEE` be a subclass of `PERSON` and `EmpNo:string` be an attribute of `EMPLOYEE`. Then we have to reflect the fact that in each state (1) every employee is a person and (2) all attributes of persons are applicable to employees.

The first point may be mathematically described by a subset relationship $\sigma(\texttt{EMPLOYEE}) \subseteq \sigma(\texttt{PERSON})$. The second point can be described by a polymorphic function with Name : person $\rightarrow$ string and Name : employee $\rightarrow$ string (analogously for Age). Hence the following query

```
SELECT (Name(e), EmpNo(e))
FROM   e:EMPLOYEE
WHERE  Age(e)>50
```

returns the name and number of each employee being older than 50.

Remark (attribute propagation against inheritance direction). Normally the attribute EmpNo : employee $\rightarrow$ string is only applicable to employees. In order to simplify navigation in object-oriented databases [BV93] attribute propagation in the opposite direction may be considered meaningful, too.

```
SELECT (Name(p), EmpNo(p))
FROM   p:PERSON
WHERE  Age(p)>50
```

The query returns the name and employee number of each person being older than 50. Of course, when in a given database state a person does not play the role of an employee then `EmpNo(p)` must be understood to be evaluated to the bottom value $\perp$. Such a mechanism may help to avoid explicit type conversions as proposed in [GH91].

4.2 Axioms in object-oriented specifications

The query formalism of structured values was motivated by a concrete language proposal for describing static and dynamic properties of objects called TROLL *light* [CGH92, GCH93, HCG94, Her95, GH95]. For TROLL *light* a sublanguage was needed for the axiomatic part of object descriptions called *templates*. This includes static integrity constraints, rules to describe the effect of events on attributes, derivation rules for attributes, rules for the synchronization of events in different objects, event preconditions, etc.

Example.

```
TEMPLATE Node
  DATA TYPES   String, Nat;
  SUBOBJECTS   Children(Name:string):node;
  ATTRIBUTES   Info:nat;
               DERIVED Total:nat;
  EVENTS       BIRTH create(nat);
                     updateInfo(nat);
                     createChild(Name:string, Info:nat);
               DEATH destroy;
  CONSTRAINTS  CNT(Children) < 5;
  VALUATION    [create(I)]     Info = I;
               [updateInfo(I)] Info = I;
  DERIVATION   Total = Info + SUM(SELECT Total(x)
                                 FROM x:Children);
  INTERACTION  createChild(N,I) >> Children(N).create(I);
  BEHAVIOR     PROCESS Node =
                 ( create -> Nodelife );
               PROCESS Nodelife =
                 ( updateInfo -> Nodelife  |
                   createChild -> Nodelife |
                   {CNT(Children)=0} destroy );
END TEMPLATE;
```

TROLL *light*	template t with $\mathrm{osort}(t) = o$	$\mathrm{self} \in \mathit{Term}(t)_o$
	attribute $a : o \longrightarrow s$	$a \in \Omega$
	subobject slot $u : o \longrightarrow s$	$u \in \Omega$

Table 5: Support for queries formulated in the context of objects

The template describes nodes (of a tree). Every node is assigned a natural number as node information. Each node may have up to four subnodes being identified by a string. For a given node n the derived attribute Total returns the sum of all node information found in the subtree rooted at n (*bill of material* problem). Besides the birth and death of a node other possible events in a node's life cycle concern updates of the node information and adding of subnodes. A node may only die when currently there are not any subnodes.

Templates may describe object communities as trees of arbitrary width and depth. In this special case the object community is a homogeneous one with all objects being of the same sort. In general object communities will be composed of rather different kinds of objects.

Let us now pay closer attention to how the query formalism is used in the construction of templates. Here we focus on the description of *derived attributes* which may be considered as predefined queries and the formulation of (static) *integrity constraints* which may be regarded as invariants.

It is a distinctive feature of TROLL *light* object descriptions that they do not presume a certain global schema view as this is usually done in most value- or object-based data models. This follows from the fact that with TROLL *light* descriptions of complex object systems can be iteratively composed from descriptions of subsystems using the subobject concept of templates [HCG94]. This has a certain impact on the manner how queries are formulated against object communities. To summarize the difference: Queries are formulated from the perspective of local objects rather than starting from a fixed schema level.

Table 5, being a continuation of Table 4, shows the technical details to deal with queries in context of TROLL *light*. Instead of referring to (global) containers, every query to be formulated in context of a certain object $\underline{o}$ starts with a self-reference to this object. Depending on the template t of which this object is an instance, the self-term to be evaluated is of object identifier sort $\mathrm{osort}(t)$[5] and, of course, in context of $\underline{o}$ self is evaluated to $\underline{o}$.

In concrete syntax the self-term is often omitted. To show the difference we give the abstract counterpart of the derivation rule.

```
Info(self)+SUM{| Total(PRJ²(x)) | x:Children(self) |}
```

Having the reference to an object one may firstly observe its attribute or subobject slots. Both are considered as functions on object identifier sorts. Attributes are evaluated to the current attribute values. Subobject symbols can be parameterized but they need not. A non-parameterized subobject symbol is evaluated to the object identifier of the corresponding subobject if this does currently exist. Otherwise it is evaluated to the bottom value $\perp$. Parameterized subobject symbols are treated as map-valued functions. For instance, in the above example **Children** is treated as a function $\mathrm{Children} : \mathrm{node} \rightarrow \mathrm{map}(\mathrm{string}, \mathrm{node})$. Such a function yields the set of

[5] In TROLL *light* there is the following convention: Names of templates start with an upper case letter while same names but with a lower case letter are used for the corresponding object identifier sorts.

object identifiers of all existing subobjects together with their logical names in the superobject.

Now to explain the above query in full detail: `Children(self)` is a term of map(string,node) being evaluated to the set of current subnodes together with their names. In turn map(string,node) can be regarded as a subsort of set(tuple(string,node)) with a functional dependence of the second component onto the first. Hence `x` is a variable of sort tuple(string,node) so that $\texttt{Total(PRJ}^2\texttt{(x))}$ returns the total value of a subnode (in concrete syntax `Total(x)` has to be considered as an abbreviation of the full path expression). All totals of subnodes are summed up and the local `Info` value is added to give the desired result.

With this information in mind the constraint expressed in the node template should be self-explaining. In the cited form the query and constraint formalism can only be used to express static integrity conditions (invariants). Apart from other means for behavior specification transitional integrity conditions could be expressed by adding special *old* and *new* predicates referring to the state before or after the current state. For arbitrary temporal constraints temporal logic may be employed.

5 Conclusions

In this paper we gave a formal definition of a query formalism in the context of structured values. We showed that this formalism can be easily used as a formal basis of concrete SQL-like query languages in both value- and object-based database models. Apart from that it can be used for stating axioms in object-oriented specifications.

Regarding general issues of object-oriented query languages [BNPS92] we claim that our query formalism fulfills basic requirements. For instance, *navigation* is supported by having surrogate-valued attributes as operations, and *methods* are integrated by user-defined operations and by derived attributes. The calculus is *extensible* by changes to the basic signature. We stress the fact that in contrast to many other query proposals our query calculus is completely *orthogonal*. This can smooth out some need to explicitly store interim query results.

Object-oriented query languages have been classified along the lines of *value-generating*, *object-generating* and *object-preserving*. With respect to this classification the presented formalism may be called value-generating. However, it is a characteristic feature of values in contrast to objects that their existence is time-independent. Only objects can be created or deleted [Mac82]. So a better denomination would be *value-resulting*. When restricting select queries to select terms of sort set(o) where o represents an object identifier sort the corresponding query formalism could also be called object-preserving. Queries based on the invention of new object identifiers were not considered in this paper (cf. [AK89]).

A prototype implementation of the query formalism using the facilities of the object-oriented database management system ObjectStore [LLOW91] to store structured values has been finished. The term and formula evaluation unit is part of a validation tool for object-oriented specifications [VHG$^+$93, HG94b, Her95]. The aim of this tool is to support the animation of object descriptions according to [CGH92]. At the time being term evaluation is based on the (naive) evaluation scheme of Section 3.2 which does not take into account query optimization. Since query evaluation is close to evaluation of applicative programs, optimization strategies developed in this field may be adopted to improve evaluation performance [JK84].

References

[AB88] S. Abiteboul and C. Beeri, *On the Power of Languages for the Manipulation of Complex Objects*, Research report 846, INRIA France, 1988.

[ABGG89] S. Abiteboul, C. Beeri, M. Gyssens, and D. Van Gucht, *An Introduction to the Completeness of Languages for Complex Objects and Nested Relations*, In Abiteboul et al. [AFS89], pp. 117–138.

[AFS89] S. Abiteboul, P.C. Fischer, and H.J. Schek (eds.), *Nested Relations and Complex Objects in Databases*, Springer, Berlin, LNCS 361, 1989.

[AK89] S. Abiteboul and P. Kanellakis, *Object Identity as a Query Language Primitive*, Proc. ACM Int. Conf. on Management of Data (SIGMOD) (J. Clifford, B. Lindsay, and D. Maier, eds.), ACM SIGMOD Record 18:2, 1989, pp. 159–173.

[AU79] A.V. Aho and J.D. Ullman, *Universality of Data Retrieval Languages*, Proc. 6th ACM Symp. Principles of Programming Languages (POPL), 1979, pp. 110–120.

[AV88] S. Abiteboul and V. Vianu, *Datalog Extensions for Database Updates and Queries*, Research report 715, INRIA France, 1988.

[AV92] S. Abiteboul and V. Vianu, *Expressive Power of Query Languages*, Research report 1587, INRIA France, 1992.

[Ban88] F. Bancilhon, *Object-Oriented Database Systems*, Proc. 7th ACM Symp. Principles of Database Systems (PODS), 1988, pp. 152–162.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel, *A Query Language for the O_2 Object-Oriented Database System*, Proc. 2nd Int. Workshop on Database Programming Languages (R. Hull, R. Morrison, and D. Stemple, eds.), Morgan-Kaufmann, San Mateo (CA), 1989, pp. 122–138.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis (eds.), *Building an Object-Oriented Database System - The Story of O_2*, Morgan-Kaufmann, San Mateo (CA), 1992.

[Bee88] D. Beech, *A Foundation for Evolution from Relational to Object Databases*, Advances in Database Technology, Proc. Int. Conf. on Extending Database Technology (EDBT) (J.W. Schmidt, S. Ceri, and M. Missikoff, eds.), Springer, Berlin, LNCS 303, 1988, pp. 256–270.

[Bee90] C. Beeri, *A Formal Approach to Object-Oriented Databases*, Data & Knowledge Engineering **5** (1990), no. 4, 353–382.

[BH93] J. Van den Bussche and A. Heuer, *Using SQL with Object-Oriented Databases*, Information Systems **18** (1993), no. 7, 461–487.

[BK86] F. Bancilhon and S. Khoshafian, *A Calculus of Complex Objects*, Proc. 5th ACM Symp. Principles of Database Systems (PODS), 1986, pp. 53–60.

[BNPS92] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella, *Object-Oriented Query Languages: The Notion and the Issues*, IEEE Trans. on Knowledge and Data Engineering **4** (1992), no. 3, 223–237.

[BV93] J. Van den Bussche and G. Vossen, *An Extension of Path Expressions to Simplify Navigation in Object-Oriented Queries*, Deductive and Object-Oriented Databases, Proc. DOOD'93 (S. Ceri, K. Tanaka, and S. Tsur, eds.), Springer, Berlin, LNCS 760, 1993.

[Cat94] R. Cattell, *The Object Database Standard: ODMG-93*, Morgan-Kaufmann, San Mateo (CA), 1994.

[CDLR90] S. Cluet, C. Delobel, C. Lécluse, and P. Richard, *RELOOP, an Algebra Based Query Language for an Object-Oriented Database System*, Data & Knowledge Engineering **5** (1990), no. 4, 333–352.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig, *TROLL light: A Core Language for Specifying Objects*, Informatik-Bericht 92–02, Technische Universität Braunschweig, 1992.

[Cod70] E.F. Codd, *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM **13** (1970), no. 6, 377–387.

[Dat84] C. Date, *A Critique of the SQL Database Language*, ACM SIGMOD Record **14** (1984), no. 3, 8–54.

[DGJ92] S. Dar, N.H. Gehani, and H.V. Jagadish, *CQL++: A SQL for the Ode Object-Oriented DBMS*, Advances in Database Technology, Proc. Int. Conf. on Extending Database Technology (EDBT) (A. Pirotte, C. Delobel, and G. Gottlob, eds.), Springer, Berlin, LNCS 580, 1992, pp. 201–216.

[EGH+92] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich, *Conceptual Modelling of Database Applications Using an Extended ER Model*, Data & Knowledge Engineering **9** (1992), no. 2, 157–204.

[GCH93] M. Gogolla, S. Conrad, and R. Herzig, *Sketching Concepts and Computational Model of TROLL light*, Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO) (A. Miola, ed.), Springer, Berlin, LNCS 722, 1993, pp. 17–32.

[GG92] M. Gyssens and D. Van Gucht, *The Powerset Algebra as a Natural Tool to Handle Nested Database Relations*, Journal of Computer and System Sciences **45** (1992), no. 1, 76–103.

[GH91] M. Gogolla and U. Hohenstein, *Towards a Semantic View of an Extended Entity-Relationship Model*, ACM Trans. on Database Systems **16** (1991), no. 3, 369–416.

[GH95] M. Gogolla and R. Herzig, *An Algebraic Semantics for the Object Specification Language TROLL light*, Recent Trends in Data Type Specification (WADT'94) (E. Astesiano, G. Reggio, and A. Tarlecki, eds.), Springer, Berlin, LNCS, 1995.

[GMN84] H. Gallaire, J. Minker, and J.-M. Nicolas, *Logic Databases: A Deductive Approach*, ACM Computing Surveys **16** (1984), no. 2, 153–185.

[Gog94] M. Gogolla, *An Extended Entity-Relationship Model — Fundamentals and Pragmatics*, Springer, Berlin, LNCS 767, 1994.

[GV92] G. Gardarin and P. Valduriez, *ESQL2: An Object-Oriented SQL with F-Logic Semantics*, Proc. 8th Int. Conf. on Data Engineering (ICDE), IEEE Computer Society Press, 1992, pp. 320–327.

[HCG94] R. Herzig, S. Conrad, and M. Gogolla, *Compositional Description of Object Communities with TROLL light*, Proc. Basque Int. Workshop on Information Technology (BIWIT'94): Information Systems Design and Hypermedia (C. Chrisment, ed.), Cépaduès-Éditions, Toulouse, 1994, pp. 183–194.

[HD91] C. Harris and J. Duhl, *Object SQL*, Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD (R. Gupta and E. Horowitz, eds.), Prentice-Hall, 1991, pp. 199–215.

[HE92] U. Hohenstein and G. Engels, *SQL/EER — Syntax and Semantics of an Entity-Relationship-Based Query Language*, Information Systems **17** (1992), no. 3, 209–242.

[Her95] R. Herzig, *Zur Spezifikation von Objektgesellschaften mit TROLL light*, VDI-Verlag, Düsseldorf, Reihe 10 der Fortschritt-Berichte, Nr. 336, 1995, (Dissertation, Naturwissenschaftliche Fakultät, Technische Universität Braunschweig, 1994).

[HG94a] R. Herzig and M. Gogolla, *A SQL-like Query Calculus for Object-Oriented Database Systems*, Proc. Int. Symp. on Object-Oriented Methodologies and Systems (ISOOMS), Palermo (Italy) (E. Bertino and S. Urban, eds.), Springer, Berlin, LNCS 858, 1994, pp. 20–39.

[HG94b] R. Herzig and M. Gogolla, *An Animator for the Object Specification Language TROLL light*, Proc. Colloquium on Object Orientation in Databases and Software Engineering (V.S. Alagar and R. Missaoui, eds.), Université du Quebéc à Montréal, to be published by World Science Publishing, 1994, pp. 4–17.

[HK87] R. Hull and R. King, *Semantic Database Modelling: Survey, Applications, and Research Issues*, ACM Computing Surveys **19** (1987), no. 3, 201–260.

[Hul89] R. Hull, *Four Views of Complex Objects: A Sophisticate's Introduction*, In Abiteboul et al. [AFS89], pp. 87–116.

[JK84] M. Jarke and J. Koch, *Query Optimization in Database Systems*, ACM Computing Surveys **16** (1984), no. 2, 111–152.

[KC93] K. Koymen and Q. Cai, *SQL*: A Recursive SQL*, Information Systems **18** (1993), no. 2, 121–128.

[Kim94] W. Kim, *Observations on the ODMG-93 Proposal for an Object-Oriented Database Language*, ACM SIGMOD Record **23** (1994), no. 1, 4–9.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv, *Querying Object-Oriented Databases*, Proc. ACM Int. Conf. on Management of Data (SIGMOD) (M. Stonebreaker, ed.), ACM SIGMOD Record 21:2, 1992.

[KV84] G.M. Kuper and M.Y. Vardi, *A New Approach to Database Logic*, Proc. 3th ACM Symp. Principles of Database Systems (PODS), 1984, pp. 86–96.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreib, *The ObjectStore Database System*, Communications of the ACM **34** (1991), no. 10, 50–63.

[LR89] C. Lécluse and P. Richard, *Modeling Complex Structures in Object-Oriented Databases*, Proc. 8th ACM Symp. Principles of Database Systems (PODS), 1989, pp. 360–368.

[Mac82] B.J. MacLennan, *Values and Objects in Programming Languages*, ACM SIGPLAN Notices **17** (1982), no. 12, 70–79.

[Mak77] A. Makinouchi, *A Consideration on Normal Form of Not Necessarily Normalized Relation in the Relational Data Model*, Proc. 3rd Int. Conf. on Very Large Data Bases (VLDB), 1977, pp. 447–453.

[NPS91] M. Negri, G. Pelagatti, and L. Sbattella, *Formal Semantics of SQL Queries*, ACM Trans. on Database Systems **16** (1991), no. 3, 513–534.

[RKS88] M.A. Roth, H.F. Korth, and A. Silberschatz, *Extended Algebra and Calculus for Nested Relational Databases*, ACM Trans. on Database Systems **13** (1988), no. 4, 389–417.

[SS86] H.J. Schek and M.H. Scholl, *The Relational Model with Relation-Valued Attributes*, Information Systems **11** (1986), 137–147.

[Ull88] J.D. Ullman, *Principles of Database and Knowledge Base Systems, Vol. I*, Computer Science Press, Rockville (MD), 1988.

[VHG+93] N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad, and H.-D. Ehrich, *Towards Reliable Information Systems: The KORSO Approach*, Advanced Information Systems Engineering, Proc. 5th CAiSE'93 (C. Rolland, F. Bodart, and C. Cauvet, eds.), Springer, Berlin, LNCS 685, 1993, pp. 463–482.