



FRAME INFORM SYSTEMS

LDBD Research Report ◊ June 1996

Databases as diagram algebras:
specifying queries and
views via the graph-based logic of sketches

Zinovy Diskin

Laboratory for Database Design
FIS/LDBD-96-02, June 1996

Databases as diagram algebras: specifying queries and views via the graph-based logic of sketches

Zinovy Diskin *

Lab. for Database Design
Frame Inform Systems
Elizabetes Str. 23, Riga
LV-1234, Latvia
E-mail: diskin@frame.riga.lv
Fax: +371 7 828036

Abstract. The goal of the paper is to develop a graphical formalism for specifying queries and views within the sketch data model (SkeDM) introduced in [17]. *Sketches* are directed multigraphs in which some diagrams are labeled with special markers. These markers denote predicates and operations over diagrams of sets and functions.

Given a signature of operations (query language), any sketch (database schema) can be extended with derived items denoting data that can be retrieved from the database. Views to a schema S are then sketch morphisms $v : S_V \rightarrow S'$ from some sketch (view schema) S_V into an augmentation of S with derived items, S' . In this way one obtains a unifying graph-based formal language for data and metadata definition and manipulation. In particular, a formalized specification framework for heterogeneous multibase systems can be built.

The approach is described with a number of examples and then precisely formalized. The main technical contribution is the development of algebraic machinery for diagram operations including parsing of sketch terms.

* Supported partially by Grant 94.315 from the Latvian Council of Science and partially by Grant from the Chalmers University (Sweden)

1 Introduction and outline of results

1.1 Motivation. The benefits of graph-based specification languages are well-known and a wide variety of semantics-oriented graphical languages for data definition was proposed. In contrast, the issue of graphical data manipulation languages is less developed. One reason is that the main concern of semantic data models is in database schema design while data manipulation is usually moved to the logical (in fact, relational) level. Another reason is that design of graph-based query languages (GQLs) is an intricate problem (note, GQLs should be distinguished from graphical user interfaces). In fact, it requires a special machinery of graphical operations. Nevertheless, GQLs are surely useful and possess many advantages ([19, 20, 30],[10, ?],[31, 32]).

A brief but illuminating discussion of the issue can be found in [19, 20] and we will not repeat it here. The present paper shares the same goal of constructing GQL but essentially differs in the approach. GQLs proposed in [19, 20],[10, ?] are based on manipulations with objects and so are navigational in their nature. In contrast, we propose a declarative GQL based on operations with sets of objects and functions between them, that is, on operations with set-and-function diagrams.

An operation with sets and functions is specified by pointing out its arity diagram in which some input data diagram is designated. These data should possess set-theoretical (denotational) semantics: a formal description of what are output sets and functions in terms of the input ones. The very procedure body of transforming an input set-and-function diagram into the output one (operational semantics) is not specified: it is a matter of implementation and here are relevant navigational object manipulations like that of GraphLog ([10, ?]), GOOD ([19, 20]), structural recursion on sets ([5, 6, 7]) and others.

Of course, declarative object-oriented query languages based on operations with sets are well known (see, *eg*, [22] and references therein). It seems, however, that usually they lack formal semantics foundations and their presentation is overloaded with syntactic sugar. The goal of the present paper is to build such a foundation within the framework of diagram operations. The focus is in the specification machinery which should explain how to specify, compose and formalize graph-based object-oriented queries. This is not a particular query language but a framework for GQL design: by choosing a signature of diagram operations one gets a query language. Different query languages can be related by searching for how basic operations of one language can be derived in another one, and vice versa.

1.2 Problem area. There are two central questions with query languages:

- (i) syntax (with intended semantics) – what are syntactical expressions specifying query operations in a unambiguous way;
- (ii) expressive power – what is the class of operations expressible in the given syntax.

Until now, the theoretical research in the area of QL was focused mainly on the second issue. In contrast, the first one seems not to have deserved a serious elaboration since in the relational data model (RDM) – the corner-stone of the

modern database theory – the question is very easy. Indeed, a relational query is an operation on relations, its specification is nothing but an algebraic term composed from basic relational operations. Note, however, that actually relations are not merely sets but sets with attribute schemas. The presence of the latter complicates the composition: in the well-formed terms certain preconditions of schema matching should be respected. However, as these conditions are very simple, the complication is managed easily and no problem arise.

The situation changes in the case of semantic object based data models. Here operands of relational-like query expressions are classes of objects, they can be connected with complex relationships: IsA, PartOf, references from one class to another and so on. As a result, the hygiene of schema preconditions becomes much more complicated than for RDM. Another important point is the question of explicit positioning a query answer in the class/type hierarchy or, in another terminology, the question of *type derivation*. These problems, especially that of positioning, were intensively discussed in the literature ([39, 37, 22], and in [3] the problem of type derivation was stated explicitly). Nevertheless, no sufficiently general solution was found and every author uses its own *ad hoc* specification apparatus usually syntactically oversugared.

This is not surprising. Actually, the problem of specifying queries and views in the OODB context is that of specifying graphical operations on diagrams and needs special machinery. Indeed, an OODB schema can be thought of as a graph of classes and references (functions) between them. In addition, some diagrams in the graph are subjected to certain constraints. For example, an arrow can be declared as an IsA-arrow, or a family of arrows (functions) with a common target can be declared to be covering (any object in the target set is in the image of at least one function from the family) and so on. Thus, what one actually needs is a machinery of specifying operations with set-and-function diagrams subjected to constraints.

1.3 Machinery and problems to be solved. A powerful data definition language suitable to this end was developed in the mathematical category theory under the name of *sketches* (a standard reference is [4]). However, while category theorists prefer to deal with a few fixed diagram properties (commutativity, limits and colimits), we need the possibility to set arbitrary signatures of diagram properties (see [17, 15] where the sketch framework for semantic modeling is developed). In addition, the query language context suggests to treat diagram properties algebraically (whenever is possible) via introducing diagram operations as explained above. In this respect the sketch framework we are going to develop is similar to FOL whose theories can be written down for arbitrary vocabularies of predicate and operation symbols.

The main difficulty with diagram operations is to find out the proper construction of their composition. The point is that in contrast to ordinary string-based terms, diagram operations can be composed in a diversity of ways because their inputs and outputs are complex data structures (specified by sketches). Developing a precise formal construction of composition and the corresponding procedure of parsing sketch terms turned out to be rather intricate, it is the

main technical contribution of the paper.

1.4 Contents and results. In section 2 the sketch approach to data modeling is outlined through a series of examples. Their aim is to demonstrate the flexibility of the sketch language in specifying database schemas, queries and their composition, views. The material of the section is overlapped with that of [17, 15] but some new examples and considerations are added. The actual goal of the present paper is to provide the semantic modeling framework introduced in [17, 15] with formal mathematical foundations, while section 2 makes the paper self-contained.

In section 3 the formalization is presented. It constitutes the sketch data model (SkeDM) as a mathematical structure. The main results are

- (i) an accurate definition of diagram terms composed from basic diagram operations and the procedure of parsing sketches with diagram operations (Appendix A),
- (ii) an accurate formalization of distinguishing between object-creating and object-preserving queries,
- (iii) an accurate formalization of distinguishing between queries and views,
- (iv) a construction of building a finitary monad from a signature of diagram operations.²

One may think that the last result is a kind of "categorical sugar": a very abstract description of well understandable construct. Though it indeed often happens with powerful specification languages like the categorical one, our case is different. The monad formulation of query languages allows one to build an abstract data-model-independent framework for specifying architecture of complex database systems. The framework is graphically evident like usual software architecture schemas but, in contrast to them, is formalized so that graphical schemas can be considered as precise specifications suitable for implementation (cf. the declaration in [14]). The principal idea is outlined in Appendix B.

A special important part of the paper is the set of tables it contains. Table 2 presents a collection of diagram predicates (a constraint language), and tables 3,4 present a query language of diagram operations. It can be inferred through familiar category theory arguments that the language is complete in a certain very strong sense (a demonstration of its expressive power is presented in tables 5,6). Though the issue of query language completeness is outside the scope of the paper, some very brief remarks in this direction are made in Conclusion.

² The notion of monad is familiar in category theory and can be found in any textbook, *eg*, in [4] where sketches (different from ours) are also considered. Last time the notion of monad have began to be utilized in the theoretical computer science, *eg*, in semantics of programming languages ([28]), functional programming ([43]) and in the database theory as well ([7]). To understand the corresponding part of the paper it is *not* necessary to know the precise definition: an informal explanation will be given in Appendix B.

2 Semantic modeling via sketches

2.1 Internal structure of objects via arrow diagrams

The chief idea of the sketch approach is to consider all classes of objects homogeneous sets while all the type information about objects is moved into arrow diagrams adjoint to classes. In this respect sketches are close to the well-known functional data model, FDM ([26, 38], see also [2, 23]).

In the sketch framework the idea has been made precise: specifying type information amounts to specifying semantic constraints imposed on these diagrams.

For example, instead of saying that a set X consists of n -tuples over a list of domains D_1, \dots, D_n , one can equivalently say that there is a *separating* family of functions, $f_i: X \longrightarrow D_i$ ($i = 1, \dots, n$), that is, a family satisfying the following condition:

(SEP) for any $x, x' \in X$, $x \neq x'$ implies $f_i(x) \neq f_i(x')$ for some $i = 1, \dots, n$

Indeed, in such a case the function $f = [f_1 \dots f_n]$ into the Cartesian product of domains,

$$f = [f_1 \dots f_n]: X \longrightarrow D_1 \times \dots \times D_n, \quad fx \stackrel{\text{def}}{=} [f_1x, \dots, f_nx]$$

is injective so that elements of X can be considered as unique names for tuples from a certain subset of D_1, \dots, D_n , namely, the image of f . In fact, elements of X can be identified with these tuples so that X is a relation up to isomorphism.

Since, conversely, for any relation the family of its projection functions (columns) is separating, the very notions of a relationship set and a *separating source* (of functions) are equivalent. Correspondingly, on the syntax level, to specify a node as a relation one can safely leave the node without any marking but label instead the corresponding source of outgoing arrows by some marker (say, an arc) denoting the (SEP)-constraint. Actually, this is nothing but a well known idea of designating a key of a relation.

Thus, the leftmost sketch on Fig. 1 describes the node X as a ternary relation. Indeed, the intended semantics of this sketch is an extension mapping, say, $\llbracket - \rrbracket$, which assigns sets $\llbracket X \rrbracket$, $\llbracket Y_j \rrbracket$ and functions $\llbracket f_j \rrbracket$ to the corresponding nodes and arrows ($j = 1, 2, 3$) in such a way that the constraint expressed by the arc-marker is satisfied, that is, the family of functions ($\llbracket f_j \rrbracket$, $j = 1, 2, 3$) is separating. Then, as it was shown above, $\llbracket X \rrbracket$ is a relation up to isomorphism.

Another example. Instead of saying that a set X consists of subsets of another set Y , one can equivalently say that there is a binary relation $M \subset X \times Y$ which satisfies the following *epsilon* condition:

(EPS) $(\forall x, x' \in X) x \neq x'$ implies $\{y \in Y : M(x, y)\} \neq \{y \in Y : M(x', y)\}$

Indeed, in such a case the function f_M into the powerset of Y ,

$$f_M: X \longrightarrow \mathbf{Pow}(Y), \quad f_M(x) \stackrel{\text{def}}{=} \{y \in Y : M(x, y)\}$$

is injective so that elements of X can be considered as unique names for some subsets of Y , namely, those that belong to the image of f_M . In fact, elements of X can be identified with these subsets of Y so that X is a *subpowerset* of Y up to isomorphism. Here and afterwards we call a subset of a powerset a *subpowerset*. As for syntax, the considerations above show that to specify a node as a subpowerset one can safely leave the node without any marking but label instead the corresponding span diagram by some marker (say, a directed arc) denoting the (EPS)-constraint.

Similar arrow treatment of other conventional semantic constructs are presented in Table 2. Then, for example, the following five pictures (sketches) on Fig. 1 specify the node X as a (a) relation, (b) disjoint sum, (c) union, (d) the image of a given function and (e) a subpowerset. We mean that any extension mapping $\llbracket - \rrbracket$ satisfying the condition expressed by a marker *will necessarily provide the corresponding structure of the set X and its elements as well*.

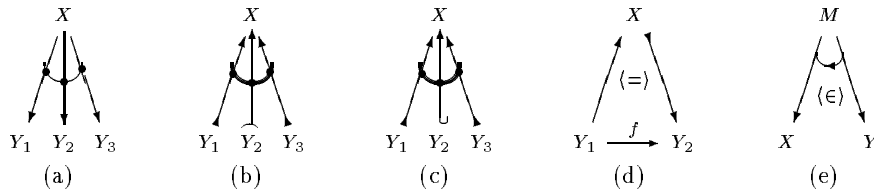


Fig. 1. Several simple sketches

Warning. *Since commutative diagrams occur very often, as a rule we will consider diagrams commutative by default, whereas non-commutative diagrams will be denoted by a special label ∇ .*

Note, ∇ is not a marker but a syntactic-sugar-label.

Example 1. A simple example of using sketches for semantic modeling is presented on Fig. 2.

The semantic situation we wish to model is described by the ER-diagram on the top figure in a conventional ER-notation. Semantics of nodes and attributes is hopefully clear from its names: $MDate$, $DDate$ are dates of marriage and divorce (the latter is optional). $BDate$ is a tuple type determined by its own attributes $Day:Int$, $Month:Int$, $Year:Int$. The domain of the attribute $Character$ is a set consisting of, say, three values a, c, q . The class $Woman$ is of weak entity type since users are assumed to be interested only in women which were/are married.

Note that actually the ER-specification is imprecise: married couples are not identified by pairs $(wife, husband)$ since it is possible that a married couple divorced and after then married again. Thus, a correct identifier is the triple $(wife, husband, mdate)$. Moreover, in the notational ER-standard there are no graphical means to express such semantics. In contrast, this can be easily done via the sketch machinery: the sketch specifying the situation is depicted on the lower

Semantic situation:
 users are interested
 in information
 (say, for the last 50 years)
 about men, married
 couples and married
 women.

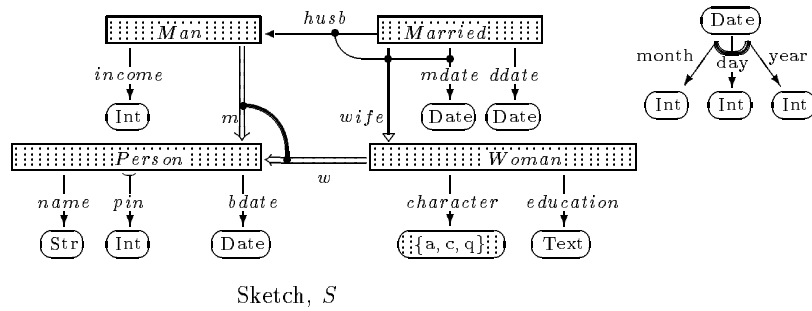
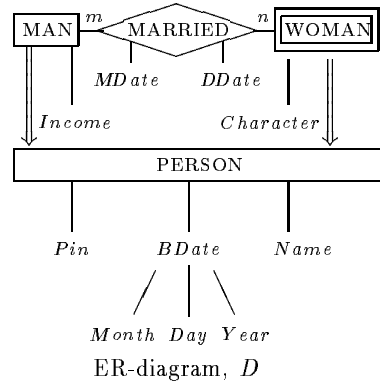


Fig.2. Semantic modeling via sketches

figure. Note also monic markers on the key attribute *pin* and the cover marker on the arrow *wife*.

Rectangle nodes are *abstract classes* whose extensions should be stored in the DB: this is additionally pointed by small dots filling-in rectangles.

Oval nodes are predefined *value domains* whose semantics is *a priori* known to the DBMS. For the sketch approach, **Int** and **{a, c, q}** are *markers* (in our precise sense) hung on corresponding nodes, that is, constraints imposed on their intended semantic interpretations. For example, if a node is marked by **Int** its intended semantics is the predefined set of integers.

At the same time, '*Person*' is a *name* labeling nodes without imposing any constraints. Generally speaking, we should also give names to the nodes labeled by **Int** and **{a, c, q}**, say, '*Number*' and '*Label*'. However, since the intended semantics of so marked nodes is fixed and remains the same for all schemas (independently on names of these nodes : '*Label*', or '*Tag*', or '*Attribute*' etc.), we adopt the convention of using such markers also as names: they will be printed in the typewriter font. So, abstract nodes are labeled by names, and printable nodes (datatypes) are labeled by markers expressing their intended semantics (while their names become redundant and can be omitted)

It seems that distinguishing node names and node markers in semantic schemas

is a precise formal description of a well known differentiation between abstract and printable classes.

2.2 Derived information via diagram operations

We begin the discussion with an observation that formulation of some natural constraints to the DB to be designed involves data which are derived w.r.t. the schema of the DB. Consider, again, the simple DB specified on Fig. 2.

An important constraint that should be added to the schema is the condition of unique identification of any currently married couple by either its husband, or its wife as well. In other words, the subset of the relation ‘Married’ for which the attribute ‘*ddate*’ is undefined, is of the one-one relationship type. To express such a constraint, one should extend the original sketch with derived items as shown on Fig. 3 (a): in order to distinguish basic items from the derived ones, the former are filled-in with small dots intended to remind about the extension to be stored. Of course, derived arrows should be also specially distinguished, and we agree to denote derived items by hanging various superscripts on their names (like ‘*’*, ‘***’, ‘*o*’ etc).

Certainly, the derived items of the sketch we consider can be obtained by an evident **Select-From-Where** SQL-query. We prefer, however, to specify this query as a composition of elementary diagram operations with sets and functions: *Null*, *CoImage*, *Composition*, presented in the top of Table 1 (as for the *Null* operation, we assume that each domain contains a single distinguished null value). Denotational semantics of operations is described in the corresponding column of the table.

Each such an operation F is specified by a sketch denoting its input data, $\mathcal{D}_F^{\text{in}}$, and a sketch denoting the output data, $\mathcal{D}_F^{\text{out}}$. It is convenient, however, to join $\mathcal{D}_F^{\text{in}}$ and $\mathcal{D}_F^{\text{out}}$ into one sketch \mathcal{D}_F so that an operation F is specified by an inclusion $\iota_F: \mathcal{D}_F^{\text{in}} \hookrightarrow \mathcal{D}_F$.

Warning *In the Table 1 sketches \mathcal{D}_F ’s are called output sketches.*

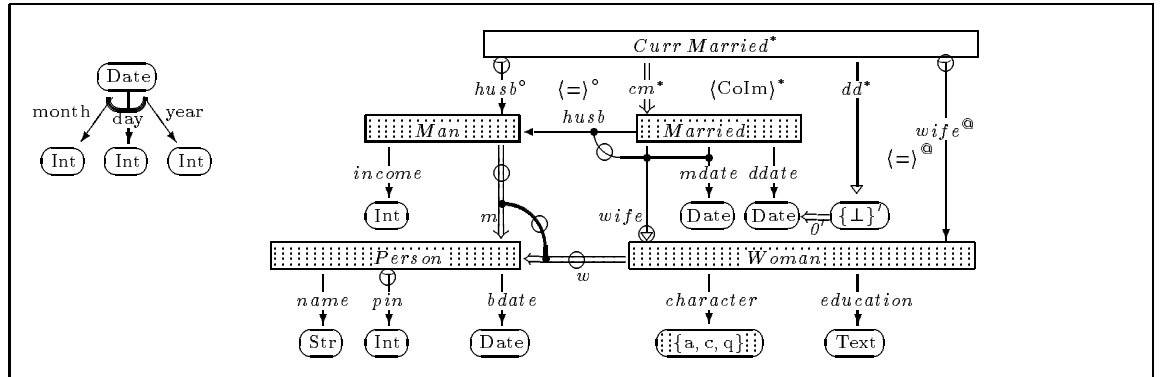
The body of operation is then a procedure $[F]$ which produce an extension of \mathcal{D}_F from a given extension of $\mathcal{D}_F^{\text{in}}$. Thus, the derived items of the sketch \overline{S} on Fig. 3(a) can be presented as follows:

```

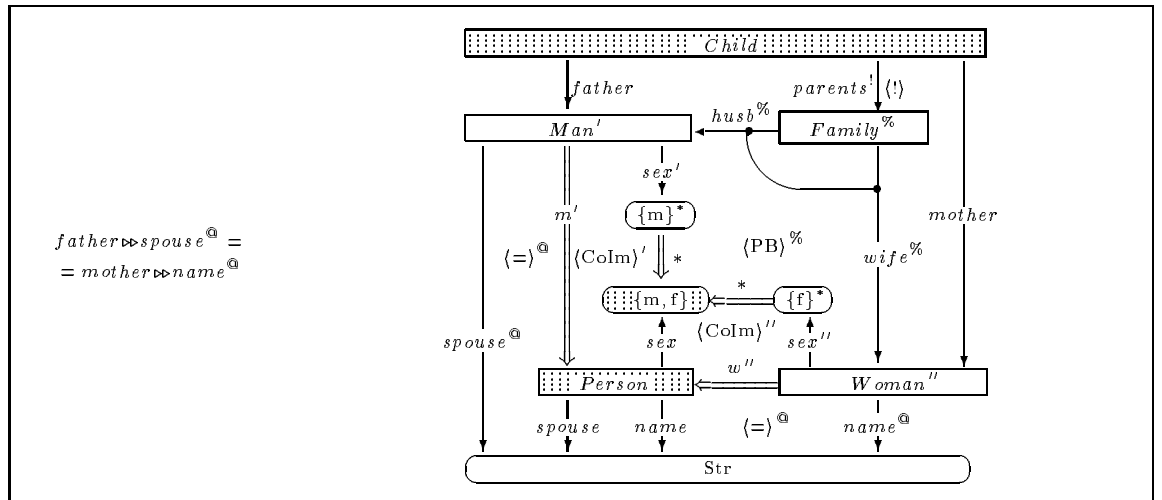
define ( $0'$ ,  $\{\perp'\}$ ) as Null (Date)
define (CurrMarried*,  $cm^*$ ,  $dd^*$ ) as CoImage ( $\{\perp'\}$ , ddate,  $0'$ )
define ( $husb^{\textcircled{a}}$ ) as Composition( $cm^*$ , husb)
define ( $wife^{\textcircled{a}}$ ) as Composition( $cm^*$ , wife)

```

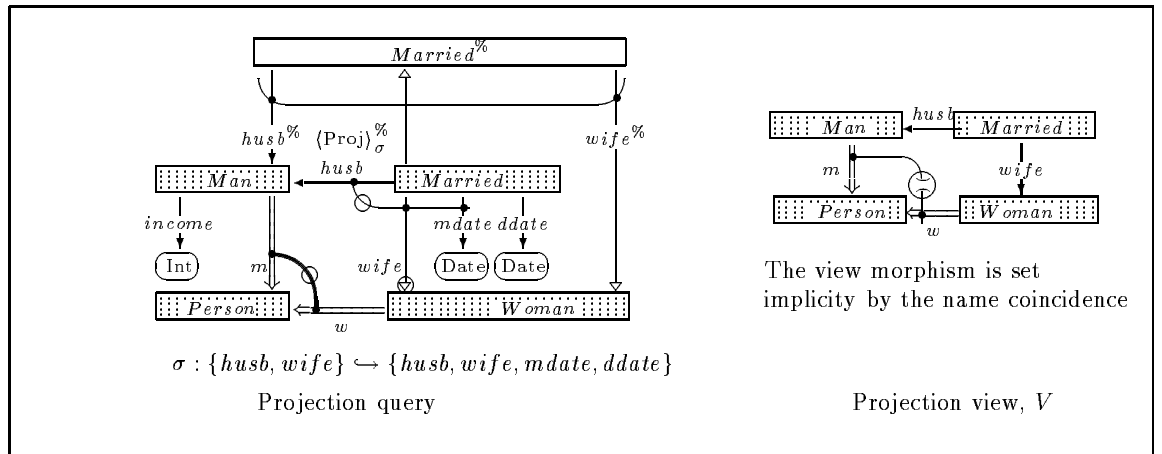
It is important to distinguish the origin of the ISA-marker that labels the arrow cm^* from that of the monic markers labeling arrows $husb^{\textcircled{a}}$ and $wife^{\textcircled{a}}$. The ISA-marker on cm^* is a postcondition of the operation *CoImage*: the corresponding constraint should be satisfied automatically provided the body of the procedure satisfies the specification. There are similar automatically satisfied postcondition markers in specifications of other operations: such a marker is present in the output sketch of operation (see Table 1). In contrast, monic markers (denoting one-one functions) hung on the arrows $wife^{\textcircled{a}}$, $husb^{\textcircled{a}}$ express



a) Involving operations to express constraints



b) Object-creating query



c) Distinguishing between projection queries and projection views.

Fig. 3. Sketches with operation markers: \S specifying a query to a sketch = Extending it with derived items

Table 1. A collection of diagram operations

Name (marker)	Arity Shape		Denotational Semantics	Linear notation
	Input sketch	Output sketch		
Null	$\bullet A$	$A \xleftarrow{0} \{\perp\}$	$0(\perp) = \perp$	
Coimage (CoIm)	$\begin{array}{ccc} & B & \\ & \downarrow b & \\ X & \xrightarrow{f} & Y \end{array}$	$\begin{array}{ccc} B' & \xrightarrow{f'} & B \\ b' \downarrow & & \downarrow b \\ X & \xrightarrow{f} & Y \end{array}$	$B' = \{x \in X : fx \in B\}$ $f' = \text{restriction of } f \text{ on } B'$	$B' = f^{-1}(Y)$ or else $B' = \text{CoIm}_f(B)$
Image (Img)	$X \xrightarrow{f} Y$	$\begin{array}{ccc} I & \xrightarrow{i} & Y \\ f' \uparrow & \nearrow f & \\ X & & \end{array}$	$I = \{f(x) : x \in X\}$ $(\forall x \in X) f'(x) = f(x)$	$I = f(X)$, or else $I = \text{Im}_f(X)$
Composition (=)	$\begin{array}{ccc} Z & \xrightarrow{g_2} & Y \\ g_1 \uparrow & & \\ X & & \end{array}$	$\begin{array}{ccc} Z & \xrightarrow{g_2} & Y \\ g_1 \uparrow & \nearrow f & \\ X & & \end{array}$	$(\forall x \in X) f(x) = g_2(g_1(x))$	$f = g_1 \circ g_2$
Projection (Proj) $_{\sigma}$ $\sigma: \{1 \dots k\} \subset \{1 \dots n\}$	$\begin{array}{ccc} X & & \\ f_1 \swarrow & & \searrow f_n \\ Y_1 & \dots & Y_n \end{array}$	$\begin{array}{ccc} X & \xrightarrow{p'} & X' \\ f_1 \swarrow & & \searrow f'_n \\ Y_1 & \dots & Y_n \\ & & Y'_1 \dots Y'_k \\ & & Y'_j \equiv Y_{\sigma(j)} \end{array}$	$X' = \text{Im}_{p'}(X)$ where $p' : X \rightarrow Y_{\sigma(1)} \times \dots \times Y_{\sigma(k)}$ is set by $p'(x) = [f_{\sigma(1)}x \dots f_{\sigma(k)}x,]$ f'_j are projection arrows	$(X', f'_1, \dots, f'_k) = \text{Proj}_{\sigma}(X, f_1, \dots, f_n)$
Pull-back (PB)	$\begin{array}{ccc} & A & \\ & \downarrow f & \\ B & \xrightarrow{g} & X \end{array}$	$\begin{array}{ccc} R & \xrightarrow{g'} & A \\ f' \downarrow & & \downarrow f \\ B & \xrightarrow{g} & X \end{array}$	$R = \{(a, b) \in A \times B : fa = gb\}$ p, q are projections	$R = \text{PB}(f, g)$
Universal Arrow of PB (!)	$\begin{array}{ccc} Y & & \\ \downarrow h & \searrow e & \\ & R & \xrightarrow{g'} A \\ & \downarrow f' & \downarrow f \\ B & \xrightarrow{g} & C \end{array}$	$\begin{array}{ccc} Y & & \\ \downarrow h & \searrow u & \searrow e \\ & R & \xrightarrow{g'} A \\ & \downarrow f' & \downarrow f \\ B & \xrightarrow{g} & X \end{array}$	$(\forall y \in Y) uy = [ey, hy]$	$u = \text{UAPB}(R, e, h)$
Difference (Dif)	$\begin{array}{ccc} A & & \\ \downarrow & & \\ X & & \end{array}$	$\begin{array}{ccc} A & & \\ \downarrow & \searrow & \\ X & \xleftarrow{=} & A' \end{array}$	$A' = \{x \in X : x \notin A\}$	$A' = X \setminus A$

the actual constraint arising from semantics of the schema, *a priori* these arrows should not be monic.

Remark on notation. It would be convenient to depict automatically satisfied markers, as well as derived items and operation markers by one colour, say, green, whereas markers denoting actual constraints by another colour, say, red. Our practice shows that colored sketches are quite readable despite a large number of conventional signs. Unfortunately, the black-white framework forces us to use less evident means. As it was said, in this paper we designate derived nodes and arrows by various superscripts near their names (eg, $X', X^*, X^{\%}, X^!, X^{\$}, X^{\text{a}}$ etc). In addition, for each application

of an operation, the operation marker and the output items are designated with the same superscript. Unfortunately, this method is not suitable for painting various icon-markers (arcs on arrow sources *etc*) by the green colour but we agree to consider them green by default while red icon-markers are additionally framed by a small circle (not to be confused with the superscript @). Thus, in the sketch \overline{S} on Fig. 2, nodes *CurrMarried* and $\{\perp\}'$ as well as arrows $husb^{\textcircled{a}}$, cm^* , dd^* , $wife^{\textcircled{a}}$, $0'$ are green whereas small tail arcs on $husb^{\textcircled{a}}$, $wife^{\textcircled{a}}$ are red. Two markers $\langle = \rangle$ and the marker *CoIm* are green (since they are operation markers) while the arc spanning $husb, wife, mdate$ and one spanning m, w , the head of *wife* and the tail of *pin* are red. Other items are black. To make the sketch much more understandable the reader is encouraged to paint it!

The query just considered is an example of object preserving queries. This follows from the fact that in the output sketch of the operation *CoImage* the derived node is a subclass of the corresponding basic node as this is explicitly specified by the ISA-arrow. In contrast, the query presented in the left part of Fig. 3(c) (do not consider the right one for a while) is object generating.³

Indeed, for each set of *Married*-tuples for which the attribute pair *husb, wife* has the same value, a single new object identifier will be generated. In the terminology of [1], *Married*[%] is a virtual class populated with imaginary objects.

We will return to this example later when we discuss the delicate difference between projection queries and projection views.

For another example of object generating queries we present the diagram treatment of one imaginary object example considered in [1]: assume that a database has a class *Person* with attributes *name, sex, spouse* and a class *Children* with attributes *cname, mother, father*. In addition, *name* and *cname* are identifiers. To view these data as a collection of families, one should create the corresponding derived classes and references as presented on Fig. 3(b). The operation marker $\langle PB \rangle$ hung on the square diagram of $(spouse^{\textcircled{a}}, name^{\textcircled{a}}, wife^{\%}, husb^{\%})$ means that the relation $(Family^{\%}, wife^{\%}, husb^{\%})$ is produced by the procedure *Pull-Back* applied to the pair m', w'' (Table 1). Class *Family*[%] consists of imaginary objects since the procedure *Pull-Back* populates the result class with newly created objects: one new object *o* for each pair of $o1 \in Man, o2 \in Woman$ for which $spouse^{\textcircled{a}}.o1 = name^{\textcircled{a}}.o2$. Commutativity of the two triangle diagrams marked by $\langle = \rangle$ means that attributes $spouse^{\textcircled{a}}, name^{\textcircled{a}}$ are inherited from that of the class *Person*.

The equation written down near the graph is a component of the sketch. It means commutativity of the corresponding square diagram (and would be abundant if to follow literally the agreement on the commutativity by default). This equation provides the precondition of applying the *Universal Arrow of PB* (Table 1) and so the arrow *parents'* can be derived.

2.3 Object-preserving vs object-generating queries

³ Its equivalent SQL-like specification is:

Select *husb, wife* **From** *Married* **Into** *Married*[%](*husb*[%], *wife*[%])

One can see that in SkeDM there exist both object preserving and object generating query operations, and moreover, an operation can simultaneously create new classes of old objects and new classes of newly generated objects. The kind of each newly derived class is determined explicitly in the specification of an operation as follows.

Let F be an operation and N is a derived node in the sketch \mathcal{D}_F (*ie*, $N \notin \mathcal{D}_F^{\text{in}}$). Then N denotes a new class of *old objects* iff there is an ISA-arrow from N to a node in $\mathcal{D}_F^{\text{in}}$. Otherwise, N denotes a class of newly created objects whose type is determined by the set of arrows going out from N . One may compare tables Table 3 and Table 4. Note the difference between operations of union and disjoint union: the former is object preserving whereas the latter is not.

Object creating operations are non-deterministic because of the possibility to choose different names from some predefined name space \mathcal{N} . In other words, operations are parameterized by a finite set $\alpha = \{\alpha_1, \dots, \alpha_n\}$ chosen from \mathcal{N} (see Table 3). Both powerset operations in the bottom of Table 3 are also parameterized by $\alpha = \{\alpha_1, \alpha_2\} \subset \mathcal{N}$ since their definitions involve binary relations to be created. To simplify notation we use in their description a canonical choice when elements of binary Cartesian product are canonical pairs, *ie*, $\alpha = \{1, 2\}$. The same is for the *Partition* operation where we choose a canonical representation of quotients by equivalence classes though another choice is also possible.

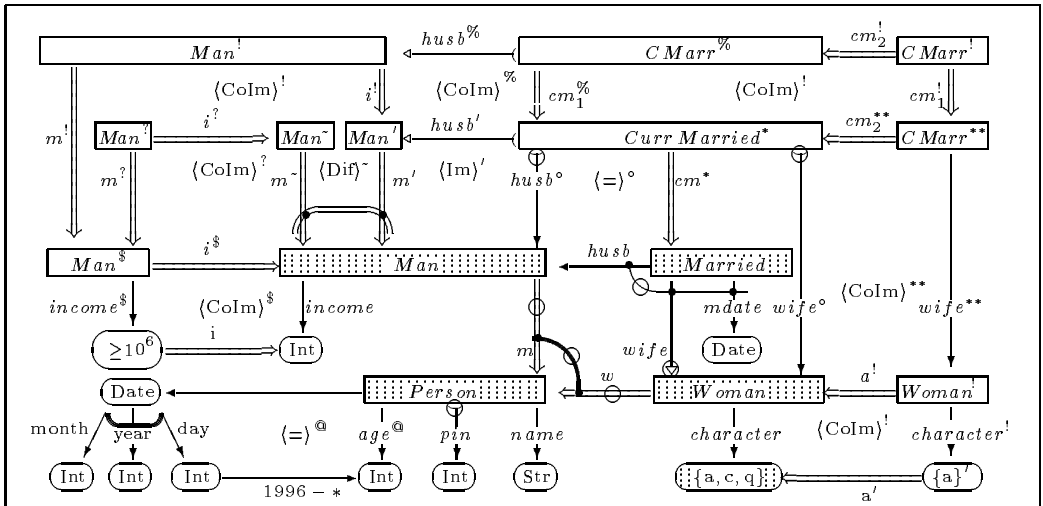
It is easy to see that two results of the same operation with differently chosen parameter α are different but isomorphic. That is, object creating operations are defined up to isomorphism and this point is well known in the DB theory. In contrast, object-preserving operations are uniquely determined: see Table 4.

The issue of up-to-isomorphism-definedness is well known in category theory, moreover, it is one of the categorical faith dogmas. Categorically, object-creating operations are defined predicatively by pointing out the corresponding universal property of the diagram \mathcal{D}_F (see the lower part of the Table 4). After that it is proven that the universal property determines the diagram of output set and functions up to isomorphism.

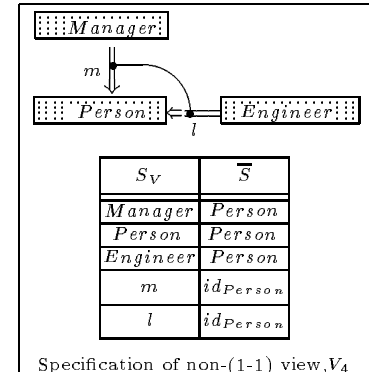
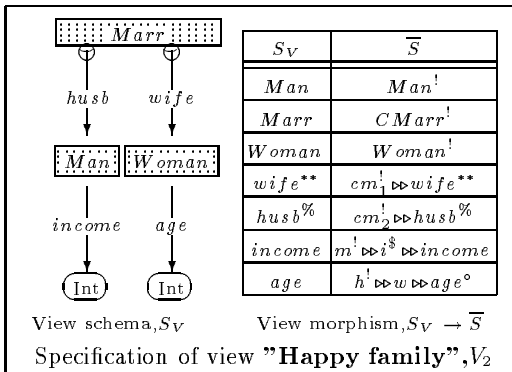
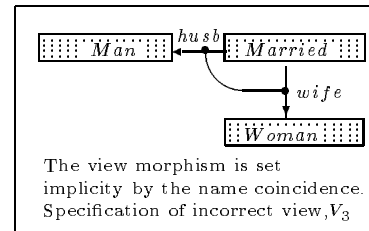
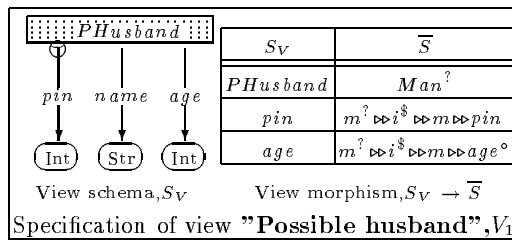
An essential advantage of definitions by universal properties is that they are polymorphic and work for any category, not only the category of sets. Here a *category* is a collection of objects together with mappings between them. In particular, all operations from all tables of operations in the paper are applicable to graphs and sketches in any signature of markers. We will often use this property below and consider, *eg*, pull-backs and push-outs of sketch mappings.

2.4 Composition of query operations

Queries can be composed by passing a part of the output data of a set of queries to the input of another query operation. The way of passing is specified by the corresponding mappings of sketches. On the other hand, the composition can be presented as a stepwise augmentation of the initial schema with derived items. In fact, we have already used this procedure in the process of building the sketches on Fig. 3(a,b). A bit more involved example is presented below.



a) Compositive queries



b) Several simple views

Fig. 4. Specifying views over a sketch = Setting sketch morphisms into its augmentations with derived items

Example 2. Consider query composition on Fig. 4(a) where we omit the fragment specifying derivation of items $CurrMarried$, $husb^{\circledast}$, $wife^{\circledast}$ described in Fig. 3(a).⁴

Given Table 1 (note, the *CoImage* operation applied to two ISA-arrows turns into the ordinary intersection of sets), with conventions adopted above, the graphical image on Fig. 4(a) specifies a system of queries against the basic sketch in a unambiguous way. For example, the marker $\langle \mathbf{CoIm} \rangle^{\$}$ labeling the the square diagram around it specifies that the extent of the node $Man^{\$}$ consists of those objects of the class *Man* for which the value of the attribute *income* is greater than 10^6 . Similarly, the extension of the node Man' is the intersection of extensions of $Man^{\$}$ and Man' where the latter is produced by the operation *Image* applied to the arrow $husb^{\circledast}$ derived, in its turn, by the composed query presented on Fig. 3(a). It follows from the semantics of these procedures that, *eg*, the extension of the class Man' consists of those happy *Man*-objects which are married (at the present time) and rich (with *income* no less than 10^6). Certainly, such a query can be expressed by standard SQL-means but the discussion of which of the languages is better is not relevant for our present considerations. All that we wish to demonstrate in this respect is how standard SQL-queries can be expressed by diagram operations over sketches (brief remark on completeness will be made below).

In fact, the sketch on Fig. 4(a) (or any other sketch with operation markers) can be converted into a parsing tree showing the process of stepwise augmenting the initial sketch via applying diagram operations from the Table 1 (or from another signature). A methodology of parsing sketch with operation markers was proposed in [16] and developed in [13] (see Appendix A below). The sketch parsing is much more intricate than the standard algebraic term parsing since inputs and outputs of diagram operations are complex structures which can be connected in a variety of ways. Such a sketch is correct if, roughly speaking, for each derived node/arrow there is one and only one operation marker F in the sketch such that the item belongs to the output sketch of the operation, \mathcal{D}_F . In addition, any item from the input sketch, $\mathcal{D}_F^{\text{in}}$, is either basic or derived with some preceding operation. These conditions provide that as soon as extension of the basic subsketch is given, the corresponding extension of any of the derived items can be computed. If a derived item belongs to outputs of two operation markers f' , f'' , it means that an equation between corresponding terms holds.

2.5 Views via sketch morphisms

An extremely important issue where the presence of derived items in a semantic schema becomes essential is that of *view* to a semantic schema. Despite the large body of work done in the area ([1, 37, 39, 29] if to mention only a few recent publications), the notion of view was not precisely formulated (as we will show)

⁴ In effect, this an instance of the sketch approach to modularization of complex schemas.

and even the terminology is still somewhat confusing. Indeed, in the majority of works (like that just mentioned) the term *view* is used to denote derived items (as a rule, classes or relations), according to the pattern “**define view name as query specification**”. In some works, however ([41, 21]), a *view* to a schema is a subschema of the schema. An evident integrated formulation is to consider a view to a schema \mathcal{S} as a subschema \mathcal{S}_V of some augmentation $\overline{\mathcal{S}}$ of \mathcal{S} with derived items, *ie*, $\mathcal{S}_V \subset \overline{\mathcal{S}} \supset \mathcal{S}$. This is an almost good definition but it forces one to consider the name space of a view schema \mathcal{S}_V as a subspace of that of the schema \mathcal{S} . However, for the view user its schema \mathcal{S}_V should be autonomous with its own name space.

Thus, a more correct consideration is to define a *view* over \mathcal{S} as a pair $V = (\mathcal{S}_V, v)$ with \mathcal{S}_V a schema (similar to \mathcal{S}) and v a mapping (or, schema morphism) $v: \mathcal{S}_V \rightarrow \overline{\mathcal{S}}$ sending items (one may think of names of classes (nodes) and references (arrows)) of \mathcal{S}_V into those of $\overline{\mathcal{S}}$.

An essential advantage of this definition is that the mapping v is not bound to be one-one, that is, it can well happen that two different items N, M in \mathcal{S}_V are glued into one item K in \mathcal{S} , $v(N) = v(M) = K$. The presence of two different names in \mathcal{S}_V could seem a trick that misleads the view user but this is not the case. The database schema \mathcal{S} may evolve and later the item K may diverge into two different items K_1, K_2 such that according to the view semantics the view morphism v has to map N to K_1 and M to K_2 , $v(N) = K_1 \neq K_2 = v(M)$. Or, vice versa, initially there were two different nodes K_1, K_2 with $v(N) = K_1, v(M) = K_2$ but then \mathcal{S} has evolved to a state when K_1 should be merged with K_2 . Thus, by means of a suitable changing the view mapping v but *without changing the view schema \mathcal{S}_V* , the DBA can conform the application built over the view with evolving database environment so that there is no need to rebuild the former.

Example 3. Two simple views V_1, V_2 to the sketch of Fig. 2 are presented on Fig. 4(b). There are however some delicate points associated with our definition of view, which we would like to demonstrate.

A view mapping must be a schema morphisms. That is, for SkeDM, v is not merely a graph morphism but a sketch morphism: if a diagram D in \mathcal{S}_V is labeled by a marker P then its image $v(D)$ in \mathcal{S} must be also labeled by P . For example, in the specification V_3 on Fig. 4(b) the corresponding mapping is not a sketch morphism since the pair $(\mathcal{S}_{V_3}.husb, \mathcal{S}_{V_3}.wife)^5$ is marked by an arc while its image, the pair $(\mathcal{S}.husb, \mathcal{S}.wife)$ in $\overline{\mathcal{S}}$ (Fig.2), is not labeled (well, the triple $[\mathcal{S}.husb, \mathcal{S}.wife, \mathcal{S}.mdate]$ is separating but it does not imply that the pair is). Semantically, this means that according to the view schema \mathcal{S}_{V_3} , any *Married*-object is identified by the pair of values of the attributes *wife, husband*, whereas the data which the view extracts from the \mathcal{S} -extension do not necessarily satisfy this condition. Hence, V_3 is not a view to \mathcal{S} . In the classical ER-notation this means that an entity node in a view schema can be mapped into a relationship node but not vice versa⁶.

⁵ we will qualify sketch items by names of sketches if necessary

⁶ The author observed a few works where this condition was not respected

The view $V4$ is a somewhat artificial example of view where the view mapping is not one-one (as for the arrow id_{Person} , we assume that any node in a sketch has the identity ISA-arrow into itself which is not depicted).

Remark. Projection queries should be carefully distinguished from projection views as it is shown on Fig. 3(c). Indeed, these two constructs have essentially different semantics: extension of the node $\mathcal{S}_V.Married$ coincides with the extension of the node $\mathcal{S}.Married$ by definition whereas extension of the node $\mathcal{S}.Married\%$ is different. Note, the construct called object-preserving projection in [39, 37, 22] is rather a view than a query.

3 An outline of formalizing sketches

Actually, the sketches we considered above are visual presentations of the corresponding formal constructs. We will call the latter *formal sketches* and the former *visual sketches*. The goal of the present section is to outline a precise formal framework for sketches and their semantics. Due to space limitations the presentation is brief and organized into a sequence of *constructions*: description units in which several definitions together with some of their immediate corollaries are compressed. It is hoped that full formal formulations can be easily recovered.

The chief technical notion here is a *diagram* over a given graph G .

We remind that in this paper a *graph* means a directed multi-graph, *ie*, a collection of nodes and arrows together with a mapping which assigns to each arrow f its *source*, $\square f$, and *target*, $f\square$. A *graph morphism* $h: G \rightarrow G'$ is defined to be a mapping which sends nodes to nodes and arrows to arrows such that the incidence relation is preserved: $\square h(f) = h(\square f)$ and $h(f)\square = h(f\square)$. Given two graphs G, G' , the set of all morphisms from G to G' will be denoted by $\mathbf{GraMap}(G, G')$

Intuitively, a diagram in G can be thought of as a collection of nodes and arrows of G . In precise terms, a *diagram* in G is a graph morphism $\delta: D \rightarrow G$ from some graph D called the *shape* of the diagram. The image of δ reflects the intuition underlying the informal notion of diagram. However, because of possible gluing together some of D -items by δ , the latter understanding can lead to ambiguities.

If $h: G \rightarrow G'$ is a graph morphism and $\delta: D \rightarrow G$ is a diagram over G then the composition $\delta' = \delta \triangleright h$ is a diagram in G' with the same shape as δ . If to think of δ informally as a subgraph in G , then δ' can be thought of as the image of δ under the mapping h and we will often designate δ' by $h(\delta)$.

3.1 Signatures of diagram predicates and operations

3.1 Construction.

Definition 1. A *signature of diagram properties (predicates)* is defined to be a pair $\Pi = (Pred, Shp)$ with $Pred = \{P_1, \dots, P_m\}$ a finite set of symbols

called *predicate markers* and Shp a function which assigns to each $P_i \in Pred$ a collection $Shp(P_i)$ of finite graphs called *arity shapes* of P_i .

The set $Shp(P)$ specifies the kind of diagrams on which the marker P can be hung. For example, the arity of the relationship marker (introduced in section 2.1 is the set of graphs which are finite sets of arrows with a common source, the arity of the ISA marker is the set consisting of a single graph which is an arrow *etc* (see Table. 2).

Given a signature Π , a (*formal*) Π -*sketch* is defined to be a tuple $\mathcal{S} = (G, P_1^{\mathcal{S}}, \dots, P_m^{\mathcal{S}})$ where G is a finite graph, *the carrier of* \mathcal{S} , and each $P_i^{\mathcal{S}}$ is a collection (maybe, empty) of diagrams in G with shapes from $Shp(P_i)$. These diagrams are to be thought of as *marked* by P_i . The graph G will be also denoted by $|\mathcal{S}|$ and sets $P_i^{\mathcal{S}}$ by $\mathcal{S}.P_i$.

Example 2. Consider the Fig. 5. The graphical image depicted on figure (b) is a visual sketch in the signature (a): it is a visual presentation of the formal sketch depicted on figure (c). The main idea is that diagrams are visualized by labeling shape items with items (names) of the carrying graph. This is nothing but a familiar way of defining mappings by tables.

Definition 3. A *sketch morphism* or *sketch mapping* is a mapping of underlying graphs compatible with marked diagrams. That is, if a diagram in the source sketch is labeled by a marker P_i then its image in the target sketch must be also labeled by P_i . Thus, with any any pair of sketches, $\mathcal{S}_1, \mathcal{S}_2$, there is associated a set of arrows from \mathcal{S}_1 into \mathcal{S}_2 , $\mathbf{SkeMap}_{\Pi}(\mathcal{S}_1, \mathcal{S}_2)$.

Given a signature of markers, Π , the definition above constitutes a graph (in fact, a category), \mathbf{Ske}_{Π} , consisting of all sketches and mappings between them. Further, we will often call Π -sketches merely sketches and omit the signature subscript in denotations.

We turn to formalizing semantics of sketches.

A predicate signature itself is a purely syntactical notion. It can be of interest if only some semantics of predicate markers is presupposed. That is, each marker P_i should be interpreted as a certain property $\llbracket P_i \rrbracket$ of set-and-function diagrams of shapes from $Shp(P_i)$.

Let $\mathcal{S} = (G, P_1^{\mathcal{S}}, \dots, P_m^{\mathcal{S}})$ be a sketch. Intuitively, an extent of \mathcal{S} is a mapping ε which assigns (finite) sets to nodes of G and functions between them to arrows of G . In addition, if some diagram δ in G is marked by P_i , $\delta \in P_i^{\mathcal{S}}$, then the corresponding diagram of sets and functions, $\varepsilon(\delta)$, must satisfy the condition $\llbracket P_i \rrbracket$. Thus, given a sketch \mathcal{S} as above, each pair (P_i, δ) with $\delta \in P_i^{\mathcal{S}}$ can be thought of as a constraint specification: extents of \mathcal{S} are extents of the underlying graph, which, in addition, satisfy all the constraints embodied into the sketch. To formalize this description we proceed as follows.

3.2 Construction. We assume some universe of proto-objects (urelements) \mathcal{O} given, and let $U = \mathbf{Set}(\mathcal{O})$ denote the graph whose nodes are finite subsets of \mathcal{O} and arrows are functions between them. Then an extent of a given sketch \mathcal{S} (in the context \mathcal{O}) is a graph morphism $\varepsilon: |\mathcal{S}| \rightarrow U$.

Of course, to provide sufficient richness of the graph U one should presuppose that the universe \mathcal{O} possesses some closure properties: closedness under

hereditary finite sets, $\mathcal{O} = \mathbf{HF}\mathcal{O}$, would be sufficient (cf. [11, 42, 36]).

Definition . *Semantics* of a marker P_i is defined to be a mapping

$$\llbracket P_i \rrbracket : \bigcup \{ \mathbf{GraMap}(D, U) : D \in \mathit{Shp}(P_i) \} \rightarrow \{ \mathit{true}, \mathit{false} \}.$$

In addition, this mapping should satisfy a certain *genericity condition* defined in the following way.

Let $\pi : \mathcal{O} \rightarrow \mathcal{O}$ be a permutation of proto-objects, $\pi \in \mathit{Aut}(\mathcal{O})$. Its action on U , π^+ , is defined as follows:

$$\text{for a set } X \subset \mathcal{O}, \pi^+ X = \mathit{Im}_\pi(X) \stackrel{\text{def}}{=} \{ \pi x : x \in X \}$$

$$\text{and for a function } f : X \rightarrow Y, (\pi^+ f)o = \pi f \pi^{-1}(o) \text{ for any } o \in \pi^+ X.$$

It is easy to see that π^+ is a graph isomorphism of U , $\pi^+ \in \mathit{Aut}(U)$.

Permutations also act on diagrams via composition, we write $\pi^+ \delta$ for $\delta \circ \pi^+$.

Now, if one makes a reasonable assumption that urelements are uninterpreted abstract objects, then diagram predicates should not depend on permutations:

$$\text{(GEN)} \quad \llbracket P_i \rrbracket (\pi^+ \delta) = \llbracket P_i \rrbracket (\delta)$$

for any $\pi \in \mathit{Aut}(\mathcal{O})$, $\delta : D \rightarrow U$, $D \in \mathit{Shp}(P_i)$. It is a predicate analog of the well-known genericity condition for queries.

The considerations above can be expressed in a more sketch-oriented manner. Indeed, mappings $\llbracket P_i \rrbracket$, $i = 1, \dots, m$ convert the graph U into a sketch $\mathcal{U} = (U, P_1^{\mathcal{U}}, \dots, P_m^{\mathcal{U}})$ with $P_i^{\mathcal{U}} = \llbracket P_i \rrbracket^{-1}(\mathit{true})$. Genericity means that $\pi^+ \delta \in P_i^{\mathcal{U}}$ for any $\delta \in P_i^{\mathcal{U}}$, that is, any π^+ is an automorphism of the sketch \mathcal{U} , $\pi^+ \in \mathit{Aut}(\mathcal{U})$.

It is easy to see that for a given sketch $\mathcal{S} = (G, P_1^{\mathcal{S}}, \dots, P_m^{\mathcal{S}})$, an extent mapping $\varepsilon : G \rightarrow U$ satisfies all the diagram constraints specified by $P_i^{\mathcal{S}}$, $i = 1, \dots, m$ iff ε is a sketch morphism $\varepsilon : \mathcal{S} \rightarrow \mathcal{U}$. Thus, if to consider a sketch \mathcal{S} as a data schema, the set of data instances satisfying the schema is nothing but the set $\mathbf{SkeMap}_{\Pi}(\mathcal{S}, \mathcal{U})$.

We turn to diagram operations. Sketches we considered in section 2 contain diagrams marked by predicate and operation markers. In addition, it can occur that input data for some operation are specified by a diagram of sets and functions subjected to certain conditions. For example, in the Table 1 input data for the operation *CoImage* is a couple of arrows with a common target and one arrow is marked by the ISA marker. Or, in the Table 3 the input data for the partition operation is a pair of arrows marked as an equivalence relation. Thus, shapes of markers are not graphs but sketches. Moreover, input data for some operation may be constrained by equations involving other, preceding, operations. For example, the shape of the operation of taking the universal arrow of pull-back, *UAPB* are specified by two squares and one of them should be pull-back while the other should be commutative (Table 1). Thence, a signature of diagram operations should be partially ordered into a hierarchy: for a marker F a finite set F^- of its precessors is defined and shapes of F are F^- -sketches (the notion of *Op*-sketch for a signature *Op* of operations is explained in Appendix A

but construction 3.3 should be read before). In the present paper we will be dealing with the simplest case when all F^- are equal to some predefined signature of predicate markers, Π . The general case will be considered in a forthcoming paper.

3.3 Construction.

Definition 1. A *signature of diagram operations* over Π is defined to be a finite set $Op = (F_1, \dots, F_n)$ of *operation markers* together with assignment to each marker F_j a finite collection $Shp(F_j)$ of Π -sketches, where in every shape sketch $\mathcal{D} \in Shp(F_j)$ a subsketch \mathcal{D}^{in} is designated as *the input sketch*. To simplify notation we will further assume that $Shp(F_j)$ consists of a single sketch \mathcal{D}_j with a designated subsketch $\mathcal{D}_j^{\text{in}}$. The changes necessary to conform with the general situation will be always evident.

Thus, we assume that the arity shape of an operation marker F_j is given by a sketch inclusion $\iota_j: \mathcal{D}_j^{\text{in}} \hookrightarrow \mathcal{D}_j$.

The complement sketch $\mathcal{D}_F \setminus \mathcal{D}_F^{\text{in}}$ will be denoted by $\mathcal{D}_F^{\text{out}}$. Note, $\mathcal{D}_F^{\text{in}} \cap \mathcal{D}_F^{\text{out}} \neq \emptyset$ in general (see tables 3,4). The set $|\mathcal{D}_j| \setminus |\mathcal{D}_j^{\text{in}}|$ of output items will be denoted by $Out(F_j)$.

Definition 2. (i) An *Op-algebra* is defined to be a Π -sketch $\mathcal{S} = (G, P_1^{\mathcal{S}}, \dots, P_m^{\mathcal{S}})$ which, in addition, carries the operations from Op , that is, for any $F_j \in Op$ there is defined a mapping

$$f_j: \mathbf{SkeMap}_{\Pi}(\mathcal{D}_j^{\text{in}}, \mathcal{S}) \rightarrow \mathbf{SkeMap}_{\Pi}(\mathcal{D}_j, \mathcal{S})$$

such that

$$f_j(e) \Big|_{\mathcal{D}_j^{\text{in}}} = e \text{ for any } e \in \mathbf{SkeMap}_{\Pi}(\mathcal{D}_j^{\text{in}}, \mathcal{S}),$$

where $f_j(e) \Big|_{\mathcal{D}_j^{\text{in}}}$ denotes the operation of function restriction, actually, $f_j(e) \Big|_{\mathcal{D}_j^{\text{in}}} \stackrel{\text{def}}{=} \iota_j \triangleright f_j(e)$. The condition guarantees projection of the input data.

Thus, *Op-algebra* is a tuple $\mathcal{A} = (\mathcal{S}, F_1^{\mathcal{A}}, \dots, F_n^{\mathcal{A}})$ with \mathcal{S} a Π -sketch denoted also by $|\mathcal{A}|$ and $F_j^{\mathcal{A}}$, $j = 1, \dots, n$ are functions as above. The entire construction can be visualized by the following diagram.

$$\begin{array}{ccc} \mathcal{D}_F^{\text{in}} & \xrightarrow{\iota_F} & \mathcal{D}_F \\ \downarrow q & \xrightarrow{[[F]]^{\mathcal{A}}} & \downarrow q' \\ |\mathcal{A}| & \equiv & |\mathcal{A}| \end{array}$$

If Op is an operation signature over Π , we will call *Op-algebras* also Σ -*algebras* where Σ denotes the merge of predicate and operation signatures into one signature

$$\Sigma = (Pred, Op, Shp, Shp^{\text{in}}) = (P_1, \dots, P_m, F_1, \dots, F_n, Shp, Shp^{\text{in}}),$$

Shp^{in} is defined only on the set Op .

Given two Op -algebras \mathcal{A}' , \mathcal{A}'' , a *morphism* from \mathcal{A}' to \mathcal{A}'' is defined to be a Π -sketch morphism $h: |\mathcal{A}'| \rightarrow |\mathcal{A}''|$ s.t. for any operation marker $F_j \in Op$ the following diagram commutes:

$$\begin{array}{ccc} \mathbf{SkeMap}_{\Pi}(\mathcal{D}_j^{\text{in}}, |\mathcal{A}'|) & \xrightarrow{F_j'} & \mathbf{SkeMap}_{\Pi}(\mathcal{D}_j, |\mathcal{A}'|) \\ \downarrow -; h & & \downarrow -; h \\ \mathbf{SkeMap}_{\Pi}(\mathcal{D}_j^{\text{in}}, |\mathcal{A}''|) & \xrightarrow{F_j''} & \mathbf{SkeMap}_{\Pi}(\mathcal{D}_j, |\mathcal{A}''|) \end{array}$$

This constitutes the category $\mathbf{Alg}_{\Sigma} = \mathbf{Alg}_{Op}(\Pi)$ of Σ -algebras (Op -algebras over Π).

Intended semantics for operation markers is as follows: for a marker F , as soon as data structured according to $\mathcal{D}_F^{\text{in}}$ are given then some procedure $\llbracket F \rrbracket$ assigned to F compute data structured according to $\mathcal{D}_F^{\text{out}}$. In other words, the arity of an operation marker can be considered as a sketch-based specification of the corresponding procedure.

Definition 3. Let \mathcal{U}_{Π} denotes the semantic Π -sketch introduced above in construction 3.2.

Semantics of a marker F with arity shape $\iota_F: \mathcal{D}_F^{\text{in}} \hookrightarrow \mathcal{D}_F$ is defined to be a mapping

$$\llbracket F \rrbracket: \mathbf{SkeMap}_{\Pi}(\mathcal{D}_F^{\text{in}}, \mathcal{U}_{\Pi}) \rightarrow \mathbf{SkeMap}_{\Pi}(\mathcal{D}_F, \mathcal{U}_{\Pi}).$$

This mapping should preserve the input data (*conservativity condition*):

$$\text{(CONS)} \quad \llbracket F \rrbracket(\delta) \Big|_{\mathcal{D}_F^{\text{in}}} = \delta \quad \text{for any } \delta: \mathcal{D}_F^{\text{in}} \rightarrow \mathcal{U}_{\Pi}.$$

where $\llbracket F \rrbracket(\delta) \Big|_{\mathcal{D}_F^{\text{in}}}$ stands for $\iota_{\llbracket F \rrbracket}(\delta)$,

and satisfy the following *genericity condition* (if one makes an assumption that urelements are uninterpreted abstract objects):

$$\text{(GEN)} \quad \llbracket F \rrbracket(\pi^+ \delta) = \pi^+(\llbracket F \rrbracket \delta) \quad \text{for any } \delta: \mathcal{D}_F^{\text{in}} \rightarrow \mathcal{U}_{\Pi}.$$

Mappings $\llbracket F \rrbracket$ convert the Π -sketch \mathcal{U}_{Π} into a Σ -algebra

$$\mathcal{U} = (\mathcal{U}, P_1^{\mathcal{U}}, \dots, P_m^{\mathcal{U}}, F_1^{\mathcal{U}}, \dots, F_n^{\mathcal{U}})$$

with $P_i^{\mathcal{U}} = \llbracket P_i \rrbracket$ as above and $F_j^{\mathcal{U}} = \llbracket F_j \rrbracket$.

Genericity means that any π^+ is an automorphism of the algebra \mathcal{U} , $\pi^+ \in \text{Aut}_{\Sigma}(\mathcal{U})$.

As soon as some operation signature is given, the central question is that of composing derived operations. For diagram operations it is a special issue described in Appendix A.

3.2 Query mechanism

3.4 Construction. An application of an operation F to a sketch \mathcal{S} is set by a sketch morphism $q: \mathcal{D}_F^{\text{in}} \rightarrow \mathcal{S}$. For instance, in the example of Fig. 3(a) the application of the $\langle \text{CoIm} \rangle$ -operation is determined by the mapping presented in the table below on the left of the double line:

$\mathcal{D}_F^{\text{in}}$ $\langle \text{CoIm} \rangle$	X	Y	f	B	b	B'	b'	f'
$\mathcal{D} \langle \text{CoIm} \rangle$	<i>Married</i>	<i>Date</i>	<i>ddate</i>	$\{\perp'\}$	$0'$	<i>CurrMarried*</i>	<i>cm*</i>	<i>dd*</i>

Semantically, q shows which sets and functions from extension of \mathcal{S} should be passed to the input of the procedure F . Syntactically, the mapping q allows to augment \mathcal{S} with derived items of \mathcal{D}_F by laying \mathcal{D}_F on \mathcal{S} in such a way that $\mathcal{D}_F^{\text{in}}$ fits a diagram in \mathcal{S} according to q , while derived items of \mathcal{D}_F extending the diagram are to be added to \mathcal{S} , the result will be denoted by $Q(\mathcal{S})$. After that, q can be extended to a morphism $q': \mathcal{D}_F \rightarrow Q(\mathcal{S})$ (in our example it is presented by the entire table above). Thus, a couple $Q = (F, q)$ with F an operation marker (name) and q a morphism as above is nothing but a query against \mathcal{S} .

Speaking formally, a *query* against \mathcal{S} is defined to be a couple $Q = (F, q)$ as above.

The arrow picture of applying such a query to a sketch is presented by the (left) commutative diagram on Fig. 6. Commutativity of triangles means that the mapping $Q(e)$ coincides with e on \mathcal{S} and coincides with $\llbracket F \rrbracket(q \triangleright e)$ on \mathcal{D}_F , in other words, $Q(e)$ is an extension including the answer to the query, and the input data are not changed.⁷

3.5 Construction. The pair $Q = (F, q)$ can be considered as a new derived operation marker whose arity is $\iota'_F: \mathcal{S} \hookrightarrow Q(\mathcal{S})$. It can be applied to another sketch \mathcal{S}_1 , say, via a query $Q_1 = (Q, h)$ (the right diagram on Fig. 6), that gives another derived marker and so on.

The collection of all derived markers that can be built over Σ in the way described above will be denoted by Σ^* . Of course, some universe of objects from which sketches $\mathcal{S}, \mathcal{S}_1$ etc are built should be *a priori* fixed. Note, if all Σ -operations are tractable then any of derived operations is tractable as well.

Derived operations can be denoted by sketches with operation markers as was demonstrated in section 2.2. On the other hand, any well-formed sketch with operation markers denotes a derived operation (or a conditional derived operation, or an equation between derived operations – see Appendix A for details).

⁷ Some category theory details: Actually, the triple $(Q(\mathcal{S}), \iota'_F, q')$ is the result of the push-out operation (the categorical generalization of union) applied to the triple $(\mathcal{D}_F^{\text{in}}, q, \iota_F)$ (and this is described by the marker $\langle \text{PO} \rangle$ labeling the square diagram, so, meta-arrow constructs are also sketches!). Then the existence and uniqueness of $Q(e)$ are provided by the so called universal property of push-outs since the outer diagram is commutative due to the condition of input data preservation.

3.6 Construction. The augmentation procedure we have described provides the following important property of the sketch morphisms. Let $h: \mathcal{S} \rightarrow \mathcal{S}_1$ be a morphism and $Q = (F, q)$ be a query against \mathcal{S} . Then the pair $Q_1 = (F, q \triangleright h)$ is also a query against \mathcal{S}_1 , which provides a similar augmentation of \mathcal{S}_1 , $Q_1(\mathcal{S}_1)$, as explained above. Then the mapping h can be extended in a unique way to the mapping $h': Q(\mathcal{S}) \rightarrow Q_1(\mathcal{S}_1)$ (the right diagram on Fig. 6).

In fact, the same process was used in the left diagram for passing from e to $Q(e)$. The point is that \mathcal{U} is an *algebra* in which the operation F is defined. Thence, the augmentation of \mathcal{U} by the query $Q_1 = (F, q \triangleright e)$ is nothing else than performing the operation F inside of \mathcal{U} . In a sense, $Q_1(\mathcal{U}) \subset \mathcal{U}$, hence, $Q(e) = e'$ maps $Q(\mathcal{S})$ into \mathcal{U} .

Correspondingly, for any composition of queries over \mathcal{S} producing an augmentation $\overline{\mathcal{S}}$, h can be extended in a unique way to $\overline{h}: \overline{\mathcal{S}} \rightarrow \overline{\mathcal{S}}_1$ where $\overline{\mathcal{S}}_1$ is the similar augmentation of \mathcal{S}_1 . In particular, $e: \mathcal{S} \rightarrow \mathcal{U}_\Pi$ can be extended to $\overline{e}: \overline{\mathcal{S}} \rightarrow \overline{\mathcal{U}} \hookrightarrow \mathcal{U}$ because \mathcal{U} is supposed to carry all the operations.⁸

To formulate these conditions precisely, the notion of monad is required. Namely, it can be shown that the category \mathbf{Ske}_Π of all Π -sketches is closed under countable Push-Outs. The latter is an operation whose input is a set (maybe, countable) of arrows with a common source and output is the corresponding set of arrows with a common target. In fact, countable Push-Outs can be "programmed" as an iteration of ordinary binary push-outs (Table 6). This provides the possibility to apply the standard algebraic routine of free algebra generation and gives the following result.

Proposition . Any signature Op of (finitary) operation markers gives rise to a (finitary) monad \mathbf{der}_{Op} over \mathbf{Ske}_Π . In addition, Op -algebras are nothing but (Eilenberg-Moore) algebras of this monad (see Appendix B for an informal explanation).

3.3 View mechanism

3.7 Construction. A view over \mathcal{S} is an arrow $v: \mathcal{S}_V \rightarrow \overline{\mathcal{S}}$ so that, given an extension e of \mathcal{S} , the corresponding extension of the view sketch is the composition of two arrows, $v \triangleright \overline{e}$. Moreover, if $v1: \mathcal{S}_{V1} \rightarrow \overline{\mathcal{S}}_V$ is a view over \mathcal{S}_V , then the extension of \mathcal{S}_{V1} is the composition $v1 \triangleright \overline{v} \triangleright \overline{e}$ where $\overline{v}: \overline{\mathcal{S}}_V \rightarrow \overline{\mathcal{S}}$, $\overline{e}: \overline{\mathcal{S}} \rightarrow \mathcal{U}$ are the corresponding extended mappings (construction 3.6). Thus, a system of views (views over views *etc*) over a sketch is nothing but a system of arrows adjoint to the node representing the sketch.

On the whole, the approach we suggest constitutes a graph-based meta-framework for semantic modeling: nodes are sketches, arrows are mappings between them, there is one distinguished node, *the universe-sketch* \mathcal{U} , such that arrows from a node (sketch \mathcal{S}) into \mathcal{U} are considered as *extensions* of \mathcal{S} . The database theory can be developed in this abstract framework to a great extent

⁸ Following the ordinary algebraic terminology, such augmentations of schemas and morphisms might be called *free*.

in a way independent of any special data model! A fragment of this development is presented in Appendix B

4 Concluding remarks

The issue of comparing expressive power and completeness of different sets of diagram operations was not addressed in the present paper, it is a subject of future research. The goal of the present one was to explain and formalize the very notion of specifying graph-based set-oriented query languages. Nevertheless, several brief remarks can be made already now.

4.1 Completeness. First of all, the language of diagram operations provides a proper mathematical framework for studies of query language completeness. Indeed, the relational algebra operations are operations on set-and-function diagrams rather than sets themselves since relations have attribute schemas. It can be shown that all classical relational operations can be derived from the diagram operations of *Cartesian Product*, *Equalizer*, *Image*, *CoImage*, *Union/Intersection*, *Difference*, *UAPr,UAEq*. To get the expressive power of nested relational algebra the *Relative Powerset* operation should be added. If one wishes complex objects with variant types, *Disjoint Union* with *UAUn* should be added as well.

The *Partition* operation is the diagram formalization of the **abstraction** operation studied in [12]. Thus, their results can be immediately transferred into the sketch framework.

On the other hand, expressiveness of different collections of diagram operations has been studied intensively in the mathematical category theory in the framework of toposes. Briefly, a *topos* is a category possessing all set-theoretical constructs. It is known that actually it is sufficient to have the operations of *Cartesian Product*, *Equalizer* and *Powerset* with their universal arrows (*UA*-operations), all the other can be derived. However, the well-established framework of toposes cannot be applied to the issue of completeness in a immediate way due to the following two points.

Firstly, in the query languages context one needs the tractable *Relative Powerset* operation instead of intractable full powerset. Secondly, category theorists do not work with inclusion arrows, they use equivalence classes of monic arrows instead. However, for studying object-preserving queries inclusions are very natural. Thus, what we really need is a version of toposes with relative powersets and inclusions. Let us call it *bounded topos*. In other words, a *bounded topos* is a category with inclusions closed under the diagram operations presented in tables 3 and 4⁹.

⁹ Actually, the relative powerset operation should be described categorically by a corresponding universal property whose formulation is too complicated to be included in the table 4. A year ago the author initiated a discussion on bounded toposes in the category theory mailing list, and a proper definition of the universal property was found by Thomas Streicher

Bounded toposes constitutes a universe for tractable set-and-function diagram operations. In fact, they seem to be a diagram counterpart of *bounded set theory* advocated and studied by Sazonov ([34, 35, 36]). To state precise relations between the three frameworks – query languages in the database theory, bounded set theory, bounded toposes – a further research is needed.

4.2 Objects and views revised. The great flexibility of sketches with diagram operations can be checked in the framework proposed in the known paper by Abiteboul and Bonner [1]. It can be shown that all constructs described in that paper can be readily explained in the sketch framework. For example, their *hide* construct ([1, section 3]) is nothing but a syntactic sugar for specifying sketch morphisms. Or, the virtual attribute construct (section 2) is the operation of taking the universal arrow of Cartesian product, *UAPr*. Specialization and generalization (section 4.1) is given by the *Union and Intersection* operation which is always applicable in the OO context since any class is a subclass of the top class of all objects. The construction of behavioral generalization (section 4.1) is more intricate. To select a certain set of classes one applies the *CoImage* operation to the meta-schema and then applies *Union* to the selected family of classes.

In this way all constructs of [1] can be obtained. Moreover, the formal framework of diagram operations reveals some points which were implicit or missed in their presentation, *eg*, the necessity of commutativity preconditions for the universal arrow operations.

4.3 Summary. What was proposed in the paper is not a single query language but rather a framework for constructing graph-based query languages. Everyone can choose the signature of operations that one prefers, and what we suggest is how to specify graph-based operations in a proper way independently on any syntactic details (a correct framework can be easily sugared afterwards!). Correspondingly, we do not elaborate in any way the deep problems of query language completeness. On the other hand, the framework of diagram operations is really universal: it is proven in category theory that any query language possessing a formal (set-theoretical) semantics can be simulated by an appropriate signature of diagram operations over sketches and, moreover, there is a single universal collection of such operations suitable for emulating all such languages.

¹⁰

An essential advantage of the languages that can be built in the sketch frame-

¹⁰ The proof goes roughly through the following known facts (see, *eg*, [4, 27]).

1. Any formal specification can be expressed in some version of set theory, or higher-order logic, or type theory (actually, this is a definition of formalizability). All these theories are interpretable as toposes.
2. Toposes can be specified by limit-colimit sketches.
3. Any topos can be presented as an algebra over the corresponding graph (since the forgetful functor is monadic).

The conclusion is that the full power of higher-order logic, including its algebraizability, can be simulated by sketches. In other words, everything that can be specified formally can be specified by sketches as well.

work is their *homogeneity*: inputs and outputs of all query procedures are collections of sets and functions satisfying certain preconditions and postconditions. This property provides the important characteristics of *orthogonality* (possibility of composition without reservations) of language constructs. Among other benefits are the evident *descriptiveness* (as opposed to navigation-orientedness) and *adequacy* in the sense of [22] when a query may return any primitive construct admissible in the model: relation, class of old objects, class of newly created objects, reference, and their arbitrary combinations as well.

References

1. S. Abiteboul and A. Bonner. Objects and views. In *Proc.ACM SIGMOD Conf. on Management of Data*, pages 238–247, 1991.
2. S. Abiteboul and R. Hull. IFO: a formal semantic database model. *ACM TODS*, 12(4):525–565, 1987.
3. R. Agrawal and L. G. DeMichiel. Type derivation using the projection operation. In *Int.Conf. EDBT'94*, Springer, LNCS'779, pages 7–14, 1994.
4. M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall International Series in Computer Science, 1990.
5. V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *3rd Int. Workshop DBPL'91*, pages 9–19, 1991.
6. V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *Int.Conf.ICDT'92*, Springer LNCS'646, pages 140–154, 1992.
7. P. Buneman, L. Libkin, D. Suciu, V. Tanen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
8. B. Cadish and Z. Diskin. Algebraic graph-based approach to management of multi-base systems, I: Schema integration via sketches and equations. In *Next Generation of Information Technologies and Systems, NGITS'95*, 2nd Int. Workshop, pages 69–79, Naharia (Israel), 1995.
9. B. Cadish and Z. Diskin. Heterogenous view integration via sketches and equations. To appear in *Proc.Int.Symp.on Methodologies for Information Systems*, Springer LNAP?, 1996.
10. P. Consens and A. Mendelson. Graphlog: a visual formalism for real life recursion. In *1990 ACM Symposium on Principles of database systems*, pages 404–416, 1994.
11. E. Dahlhaus and J. Makowsky. Computable directory queries. *Rend.Sem.Mat.Univ.Pol.Torino*, 1987.
12. J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. In *1991 ACM Symposium on Principles of Database Systems*, pages 291–299, 1991.
13. Z. Diskin. Algebraic graph-based approach to management of multibase systems, II: Mathematical aspects of schema integration. Technical Report 9502, Frame Inform Systems/LDBD, Riga,Latvia, 1995. (On ftp://ftp.cs.chalmers.se/pub/users/diskin/tr9502.*).
14. Z. Diskin. Formalization of graphical schemas: General sketch-based logic vs. heuristic pictures. In *10th Int.Congress of Logic,Methodology and Philosophy of Science*, page 401, 1995.
15. Z. Diskin. A unified algebraic graph-based framework for specifying queries, views and refinements over semantic schemas. Technical Report 9601,

- Frame Inform Systems, Riga, Latvia, 1996. Submitted for *ER'96* (On ftp://ftp.cs.chalmers.se/pub/users/diskin/er96.*).
16. Z. Diskin and I. Beylin. What is an operation over sketches? A talk given at Winter Meeting of Computer Science Department, Chalmers University, Göteborg, January, 1995.
 17. Z. Diskin and B. Cadish. Variable sets and functions framework for conceptual modeling: Integrating ER and OO via sketches with dynamic markers. In *Proc. 14th Int. Conf. on Object-Oriented and Entity-Relationship Modeling, (OO & ER)'95*, Springer LNCS'1021, 1995.
 18. J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of ACM*, 39(1):95–146, 1992.
 19. M. Gyssens, J. Paredaens, and D. Van Gutch. A graph-oriented object database model. In *1990 ACM Symposium on Principles of Database Systems*, pages 417–424, 1990.
 20. M. Gyssens, J. Paredaens, and D. Van Gutch. A graph-oriented object model for database end-user interfaces. In *1990 ACM SIGMOD Int. Conf. Management of Data*, pages 24–33, 1990.
 21. S.J. Hegner. Pairwise-definable subdirect decomposition of general database schemata. In *3rd Symp. MFDBS'91*, Springer LNCS'495, pages 243–257, 1991.
 22. A. Heuer and M.H. Scholl. Principles of object-oriented query languages. In *Database systems for Office, Scientific and Engineering Applications*, pages 178–197, Kaiserslautern, 1991. Springer.
 23. R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.
 24. L. Kalinichenko. *Methods and tools for integration of heterogeneous databases*. Nauka Publ.Co., 1983.
 25. L. Kalinichenko. Methods and tools for equivalent data model mapping construction. In *Advances in Database Technology - EDBT'90*, pages 92–119, 1990.
 26. L. Kerschberg and J.E.S. Pacheko. A functional database model. Technical report, Pontificia Univ. Catolica do Rio-de-Janeiro, 1976.
 27. J. Lambek and P. Scott. *Introduction to higher order categorical logic*. Cambridge University Press, 1986.
 28. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
 29. I.S. Mumick. The rejuvenation of materialized views. In *6th Int. Conf. CISMOT'95*, Springer LNCS'1006, pages 258–264, 1995.
 30. J. Paredaens, P. Peelman, and L. Tanca. G-Log: A declarative graphical query language. In *Int. Conf. DOOD'91*, Springer LNCS'566, pages 108–128, 1991.
 31. C. Parent and S. Spaccapietra. Complex object modeling: an entity relationship approach. In *Nested relations and complex objects in Databases*, LNCS'361, pages 272–296, 1987.
 32. C. Parent and S. Spaccapietra. ERC+: an object-based entity-relationship approach. In *Conceptual modeling, databases and CASE: An integrated view of Information System Development*, 1992.
 33. E. Pitoura, O. Bukhres, and A. Elmagarmid. Object orientation in multidatabase systems. *ACM computing surveys*, 27(2):141–195, 1995.
 34. V. Ju. Sazonov. Bounded set theory, polynomial computability and δ -programming. *Application aspect of mathematical logic. Computing Systems*, 122:110–132, 1987. (in Russian). See also the short English version in LNCS'278(1987)391-397.

35. V. Ju. Sazonov. Bounded set theory and inductive definability. *J.Symbolic Logic*, 56:1141–1142, 1991.
36. V. Yu. Sazonov. Hereditary finite sets, data bases and polynomial-time computability. *Theoretical Computer Science*, 119:187–214, 1993.
37. M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for object bases. In *Int.Workshop on Distributed Object Management, Edmonton, Canada*, 1992.
38. D. Shipman. The functional data model and the data language DAPLEX. *ACM TODS*, 6(1):140–173, 1981.
39. M. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *2nd Int.Conf.DOOD'91*, Springer LNCS'566, pages 189–207, 1991.
40. A. Sneth and C. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 1990.
41. C. Tuijn and M. Gyssens. Views and decompositions from a categorical perspective. In *4th Int.Conf. on Database Theory, ICDT'92*, volume 646 of *Springer LNCS*, pages 99–112, 1992.
42. J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating query languages. In *33rd Symposium on Foundations of Computerr Science*, pages 372–379, 1992.
43. P. Wadler. Comprehending monads. *Mathematical structures in Computer Science*, 2(4), 1992.

A Appendix. Composing diagram operations and parsing of sketches with operation markers

As soon as some diagram operation signature is given, the central question is that of composing derived operations. Since inputs and outputs of diagram operations are complex data structures specified by arbitrary sketches, they can be connected in a diversity of ways. Thus, it is highly non-trivial to find a proper construction of composing diagram operations, in other words, to find a proper graph-based counterpart of the notion of term. Of course, we mean a construction which would specify diagram terms in some abstract way regardless to their syntactical representation.

The sample we have in mind is the tree-based description of terms in the ordinary algebra: a term is a tree which can be visualized by various notational mechanisms: with or without brackets, in prefix, suffix or infix form *etc.* So, the key point in handling diagram operations is to design a proper abstract notion of a composed diagram operation (term) independently of its possible syntactical representations.

An appropriate solution was described in [16] where it was suggested to treat diagram terms as trees labeled in a special way (in a close analogy to the tree presentation of composed operations in universal algebra).

As usual, composed operations (terms) are built with operation symbols from constants or variables. The diagram version is as follows.

3.8 Construction.

Definition 1. A marker $c \in \Sigma$ is called a *constant* if $\mathcal{D}_c^{\text{in}} = \emptyset$. Each finite Π -sketch X determines a constant c_X with $\mathcal{D}_{c_X} = X$ and $\mathcal{D}_{c_X}^{\text{in}} = \emptyset$.

In the ordinary universal algebra an operation, say, f or arity 3, produces a single element which can be denoted by the expression $f(c_1, c_2, c_3)$. A diagram operation produces a sketch, that is, a collection of nodes and arrows, and one needs special names to call these items. To this end, we will fix a pool V of node names and arrow names. In addition, labeling items resulted from diagram operations must be compatible with the incidence relation. For example, if some arrow a is labeled by a name $A \in V$ and $\square a$ and $a\square$ are labeled, correspondingly by names N and M , then, if another arrow b is also labeled by the name A , $\square b$ and $b\square$ must be labeled by N and M . This requirement is satisfied if to consider the pool V as a graph of names so that labeling amounts to a graph morphism into V .

Thus, we fix a graph V containing countable many items (node and arrow names). With any finite Π -sketch X whose carrier graph is a subgraph of V we connect the constant operation marker v_X with $\mathcal{D}_{v_X} = X$, $\mathcal{D}_{v_X}^{\text{in}} = \emptyset$.

Now, let $\Sigma = (\text{Pred}, \text{Op}, \text{Shp}, \text{Shp}^{\text{in}})$ be a sketch signature (Shp^{in} is defined only on the set Σ), V a pool graph (of names) and

$$\text{Op}^+ \stackrel{\text{def}}{=} \text{Op} \cup \{v_X \mid X \text{ is a finite subgraph of } V\}$$

Definition 2. A (Σ, V) -tree is defined to be a finite tree T whose nodes are labeled by pairs of the form (F, δ) with $F \in \text{Op}^+$ and $\delta: |\mathcal{D}_F| \rightarrow V$ a graph morphism. The pair (F, δ) labeling a node k will be denoted by $\lambda(k)$.

We remind that given a sketch \mathcal{S} , its underlying graph is denoted by $|\mathcal{S}|$, and given a graph G , the set of its items (nodes and arrows) is denoted by $|G|$. Thus, $||\mathcal{S}||$ is the set of nodes and arrows of \mathcal{S} .

So, at every tree node k labeled by $\lambda(k) = (F, \delta)$, one has three sets of shape items:

$$||\mathcal{D}_F||, ||\mathcal{D}_F^{\text{in}}|| \quad \text{and} \quad \text{Out}(F) \stackrel{\text{def}}{=} ||\mathcal{D}_F|| \setminus ||\mathcal{D}_F^{\text{in}}||,$$

their images in the name pool under the mapping $|\delta|: ||\mathcal{D}_F|| \rightarrow |V|$ will be denoted by respectively,

$$V_k, V_k^{\text{in}}, V_k^{\text{out}}.$$

Note, V_k^{out} is not necessarily equal to $V_k \setminus V_k^{\text{in}}$ since δ can assign the same name to an item from $||\mathcal{D}_F^{\text{in}}||$ and that from $\text{Out}(F)$.

Actually, each node of a tree denotes a computation (query) whose input should be taken from outputs of computations denoted by preceding nodes. The very data transferring is provided by common names in the ranges of the corresponding diagrams.

Definition 3. A (Σ, V) -tree is called *well-formed* if for each node k the following condition is fulfilled:

$$(\text{wft}) \quad V_k^{\text{in}} \subset \bigcup \{V_j \mid j \in k^-\}$$

where k^- denotes the set of nodes preceding k .

Now we turn to convenient syntactical presentations of (Σ, V) -trees similar, in a sense, bracket-based notation in algebra.

3.9 Construction. As it was demonstrated in section 2 (Fig. 4a), it is possible to specify composed operations in the sketch language too. Indeed, since

any operation marker is evidently a predicate marker as well, one can combine $Pred$ and Op into a signature

$$\Sigma_{pr} = (Pred \cup Op, Shp, Shp^{in}) = (P_1, \dots, P_m, F_1, \dots, F_n, Shp, Shp^{in})$$

where Shp^{in} is defined only for the second part of the list of markers. The same data without the mapping Shp^{in} is a predicate signature which is denoted by Σ_{pr} .

Thus, we should carefully distinguish between Σ_{pr} -sketches and Σ -sketches. The former are defined as usual. In contrast, Σ -sketches are Σ_{pr} -sketches in which, in addition,

- (i) some items are designated *basic*, and they have to form a subsketch,
- (ii) each non-basic item is *properly derived*, that is, occurs in the output part of a diagram labeled by an operation marker. Then the input part of the diagram must be derived with some preceding operation applications.

These conditions provide that as soon as extension of the basic subsketch is given, the corresponding extension of any of the derived items can be computed. In other words, Σ -sketches is a graph-based counterpart of terms in the ordinary logic.

Of course, these considerations are informal. To make them precise we need to develop a construction of Σ -sketch parsing. We mean a procedure which will convert any Σ_{pr} -sketch \mathcal{S} into a (Σ, V) -term tree or end in failure pointing out that \mathcal{S} is an ill-formed Σ -sketch (though it is a well-formed Σ_{pr} -sketch).

Evidently, any (Σ, V) -tree T determines a Σ -sketch $Ske(T)$ whose carrier graph $V(T) \stackrel{\text{def}}{=} |Ske(T)|$ is the union of all V_k when k ranges over T -nodes (see definition 3.8.2).

The diagrams of this sketch are as follows.

For each $P_i \in Pred$, $P_i.Ske(T) = \bigcup \{P_i.D_F \mid k \in T \text{ and } \lambda(k) = (F, \delta)\}$,

for each $F_j \in Op$, $F_j.Ske(T) = \{\delta \mid k \in T \text{ and } \lambda(k) = (F_j, \delta)\}$.

In addition, the carrier graph contains a distinguished subgraph

$V^{in}(T) \stackrel{\text{def}}{=} \bigcup \{V_k \mid \lambda(k) = (F, \delta) \text{ with } F \in Op^+ \setminus Op \text{ i.e., } k \text{ is a variable leaf}\}$.

Actually it is a Π -sketch $Ske^{in}(T)$ with the following marked diagrams:

$$P_i.Ske^{in}(T) = \bigcup \{P_i.D_F \mid k \in V^{in}(T) \text{ and } \lambda(k) = (F, \delta)\}.$$

This sketch specifies the input data of the procedure denoted by the entire tree.

The construct extracted above from (Σ, V) -trees can be described as follows.

Definition 1. Given a signature of markers Π , a Σ -specification is a pair $\mathbf{S} = (\mathcal{S}, \mathcal{B})$ where \mathcal{S} a finitary Σ_{pr} -sketch and \mathcal{B} is some its distinguished Π -subsketch, $\mathcal{B} \subset \mathcal{S}$. \mathcal{B} is called *the input sketch* of \mathbf{S} .

If to speak about visualization, then a Σ -specification is a visual Σ_{pr} -sketch in which some *basic* subsketch is designated (by, say, painting items not occurring into it, *derived items*, with the green colour).

Definition 2. \mathbf{S} is called *well-formed* (wf) if there is a (Σ, V) -tree T s.t. $\mathcal{S} = Ske(T)$ and $\mathcal{B} = Ske^{in}(T)$. In this case we will call \mathbf{S} a Σ -sketch.

Proposition 3. (*Sketch parsing*).

There is an algorithm either extracting from any given specification $\mathbf{S} = (\mathcal{S}, \mathcal{B})$ a tree $T = Tree(\mathbf{S})$ s.t. $\mathcal{S} = Ske(T)$ and $\mathcal{B} = Ske^{in}(T)$, or terminated with failure proving that \mathbf{S} is not well-formed.

In other words, it is decidable whether a given Σ -specification (visual Σ_{pr} -sketch) is well-formed.

A simple example of sketch parsing is presented on Fig. 7(a) (we remind that items with superscript are assumed to be green).

3.10 Construction. The notational mechanism of Σ -sketches possesses a great flexibility: it allows to specify in a uniform way not only diagram terms but also equations between terms and conditional terms as well (the latter are terms which denote an operation only if some equations are satisfied).

Definition 1. A pair of *different* nodes (k, l) is called *independent* if $V_k^{out} \cap V_l^{out} = \emptyset$. Otherwise nodes are called *dependent* and we write $Dep(k, l)$.

Informally, dependency means that some output items of the procedure assigned to k should be always equal to certain output items of the l -procedure and, hence, one has an equational constraint.

A node k labeled by (F, δ) is called *free* if

- (a) it is independent on other nodes,
- (b) $V_k^{out} \cap V_k^{in} = \emptyset$ and
- (c) the restriction of the mapping $|\delta|$ on the set $Out(F)$ is one-one.

The intuition is that a node is free if items produced at this node are named by their own names so that there are no any equations between neither themselves nor with any other node output items. In contrast, a non-free node denotes a procedure whose output is constrained by some equations.

Definition 2.

A tree is called *free* if all its nodes are free.

A tree is called *operational* if its root is a free node.

A tree is called *assertional* if its root is not free.

A free tree specifies a composed operation without any reservation on the output items of the components, and hence can be applied for any input data (extent instantiation of its leaves). In contrast, non-free operational tree specifies a procedure which succeed only if some equations hold, that is, for some input data the operation may be undefined. An assertional tree specifies certain equations between composed operations, *ie*, a constraint assertion.

Now proposition 3.9.3 can be developed in the following way.

Proposition 3. (*Sketch parsing completed*)

There is an algorithm either extracting from any given sketch specification $\mathbf{S} = (\mathcal{S}, \mathcal{B})$ a (Σ, V) -tree T s.t. $\mathcal{S} = Ske(T)$ and $\mathcal{B} = Ske^{in}(T)$, or terminated with failure proving that \mathbf{S} is not well-formed.

If \mathbf{S} is well-formed then the extracted tree T is

- either a free operational,
- or a non-free operational,
- or an assertional.

So, it is decidable whether a given Σ -specification denotes a free term, or a conditional terms, or a system of equations, or is ill-formed.

Simple examples are presented on Fig. 7.

Proposition 4. Let $\mathbf{S} = (\mathcal{S}, \mathcal{B})$ be a Σ -sketch (wf sketch specification) and $e: \mathcal{B} \rightarrow \mathcal{A}$ be a Π -sketch-morphism for some Σ -algebra \mathcal{A} .

(i) There is no more than one expansion of e to a Σ -sketch-morphism $\bar{e}: \mathcal{S} \rightarrow \mathcal{A}$ for which the restriction $\bar{e}|_{\mathcal{B}}$ equals e .

(ii) If $Tree(\mathbf{S})$ is free the expansion do exist.

If $\mathcal{A} = \mathcal{U}$, an expandable morphism $e: \mathcal{B} \rightarrow \mathcal{A}$ is called an (*extent*) instance of \mathbf{S} .

B Appendix. Databases as diagram algebras: formalizing schemas for multibase environment architecture

Graphical pictures describing software architecture are widely used as an informal means facilitating presentation and discussion. In contrast, we present a framework where software schemas are formal graph-based specifications (sketches!) whose nodes are DB schemas and arrows are schema morphisms. The framework is based on three fundamental observations:

- query languages are monads over categories of DB schemas,
- views are Kleisly morphisms of these monads,
- databases are algebras of these monads.

Given a signature Π of diagram predicates, Π -sketches can be considered as database schemas. The collection of all such schemas is a graph, **Schema**, whose nodes are Π -sketches and arrows are mappings between them. In fact, this graph is a category since compositions of sketch mappings is defined and there are identity sketch mappings.

As was said, any signature Σ of diagram operations over Π gives rise to a finitary monad over the category **Schema**. A monad is an arrow generalization of closure operators frequently used in logic and algebra. Given a sketch \mathcal{S} , by repeatedly applying the procedure of augmenting a sketch \mathcal{S} with derived items (constructions 3.5,3.6), one gets the monstrous closure

$$\mathbf{der}_{\Sigma}\mathcal{S} = \bigcup \{Q(\mathcal{S}) \mid Q \in \Sigma^*\} \supset \mathcal{S}$$

which contains all possible derived items produced by Σ -queries.

Certainly, for particular questions one needs only finite parts of $\mathbf{der}_{\Sigma}\mathcal{S}$, however, the concept of the full closure can be very useful as we will see below.¹¹ In addition, the semantic universe of data (the database), \mathcal{U} , turned out to be a \mathbf{der}_{Σ} -closed schema: it is endowed with a mappings $\mu_{\mathcal{U}}: \mathbf{der}_{\Sigma}\mathcal{U} \rightarrow \mathcal{U}$ corresponding to computing extent of derived items.

Thus, a query mechanism is reduced to a closure operator \mathbf{der}_{Σ} (monad) over the category **Schema** = **Ske** $_{\Pi}$. Different signatures can generate the same

¹¹ This is quite similar to the usefulness of the notion of the set of natural numbers which is nothing but $\mathbf{der}_{\Sigma}\{0\}$ for Σ consisting of a single operation *successor*

monad and then, if to identify query languages with the same expressive power, a monad is a precise formalization of the notion of query language independent of particular signature choices.

To summarize, while a data definition language constitutes a category **Schema** (**Ske Π** in our formalism), a query language Σ gives rise to a monad over this category. Given a database schema (Π -sketch) \mathcal{S} , a view V over \mathcal{S} is a mapping $v: \mathcal{S}_V \rightarrow \mathbf{der}_\Sigma \mathcal{S}$ from the view schema $\mathcal{S}_V \in \mathbf{Schema}$ into the \mathbf{der}_Σ -closure of the database schema (such morphisms are called *Kleisly morphisms* of a monad). As soon as an extent $e: \mathcal{S} \rightarrow \mathcal{U}$ of the database schema is given, the extent $e_V: \mathcal{S}_V \rightarrow \mathcal{U}$ can be computed by composition $v \gg \bar{e}$.

In this framework, a centralized DB-system with a system of views over it can be specified by the diagram presented on Fig. 8. Note, the corresponding graphical image is not merely a picture convenient for heuristic discussion but also a precise formal specification.

The same framework allows to specify formally the general architecture of a federal DB system (substantially described in [40, 33] as shown on Fig. 9), the result is presented by the meta-schema depicted on Fig. 10. There are shown two federal superviews, $\overline{\mathcal{S}}_a^I$ and $\overline{\mathcal{S}}_b^I$, which are obtained by some procedure of sketch integration described in [8, 9]; two sketches $\overline{\mathcal{S}}_{CI}$ specify the correspondence information required for integration (see the references above for details). There is also depicted a system of external views against the a -superview (a similar system for b is not shown).

In the explanation column on the right, 'data model' means a triple

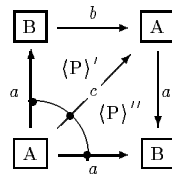
$$(\mathbf{Schema}, \mathbf{der}, \mathcal{U})$$

as above, and ordinary arrows are morphisms in the corresponding category **Schema**. In contrast, "curly" arrows are functors between categories; in effect, these functors endowed with additional monad-related components are *data model morphisms* studied by Kalinichenko in [24, 25] or, in another terminology, *institution morphisms* extensively studied in the institution theory, see [18] for references.

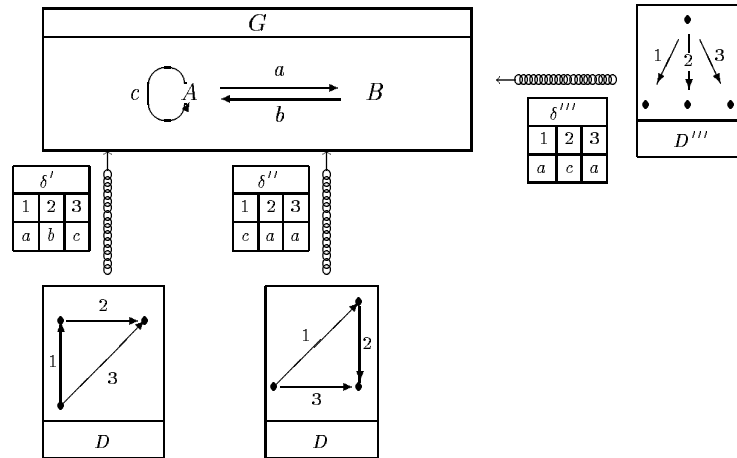
As a justification for *the abstract nonsense* of category theory, one can observe nice similarity of Fig. 9 and Fig. 10: while the former is an informal picture, the latter is a precise specification.

Marker	P	Q	\smile
Shapes			

a) Signature, Π



b) Visual Π -sketch



c) Formal presentation of the visual sketch above,

$$S = (G, P^S, Q^S, \smile^S)$$

$$P^S = \{\delta', \delta''\}, Q^S = \emptyset, \smile^S = \{\delta'''\}$$

Fig. 5. Example of sketch with predicate markers

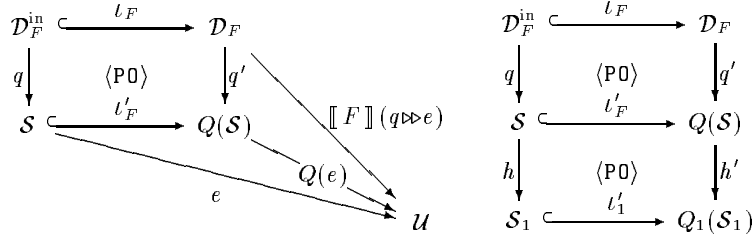
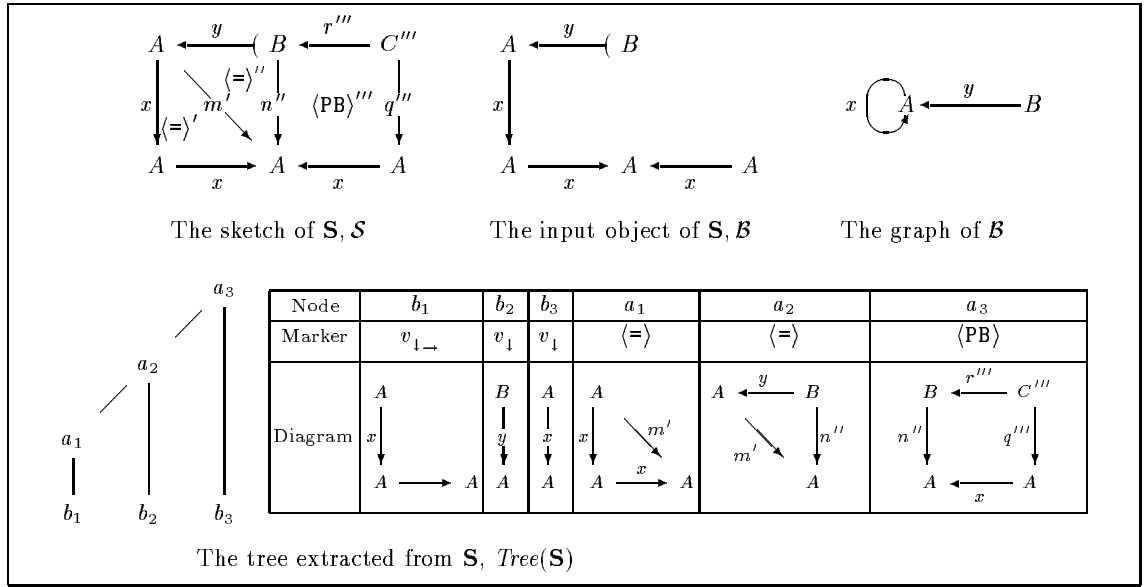
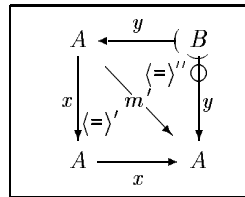


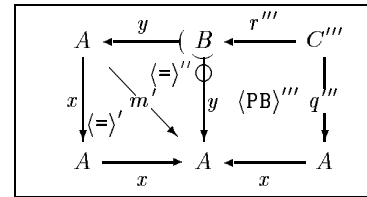
Fig.6. Diagrams of query mechanism



a) Free operational specification $\mathbf{S}=(\mathcal{S}, \mathcal{B})$ and its parsing



b) Assertional specification: $y \bowtie (x \bowtie x) = y$



c) Conditional operational specification:
if $y \bowtie (x \bowtie x) = y$
then $(C''', r''', q''') = PB(y, x)$

Fig.7. Examples of Σ -sketches in the operation signature $\Sigma = \{ \dashv, =, PB \}$ with arity shapes in Table ??

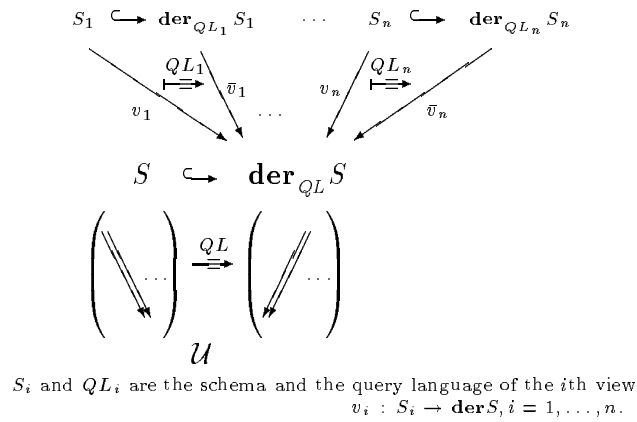


Fig. 8. Meta-schema of a centralized DB with a system of views

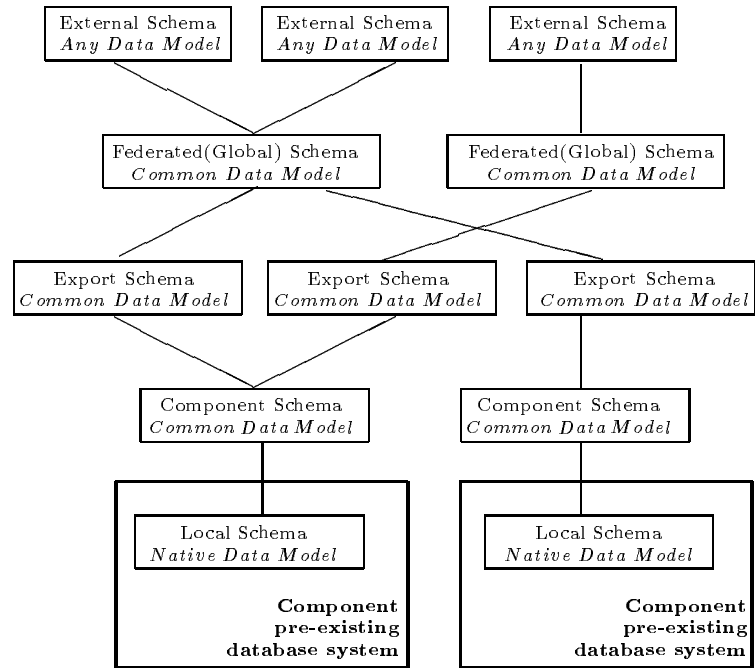


Fig. 9. General architecture of a federal DB system (from [33])

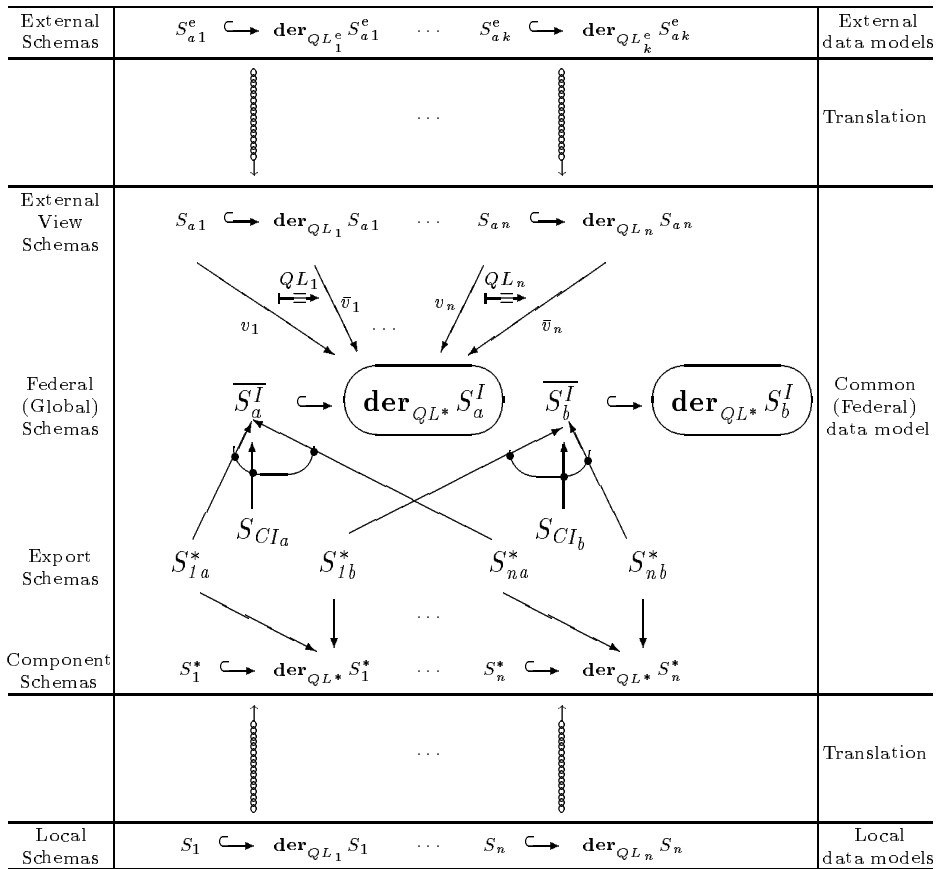


Fig.10. Meta-schema of a federal DB

Table 2. Table 2

Table 3. Table 3

Table 4. Table 4

Table 5. Table 5

Table 6. Table 6