The Categorical Product Data Model as a Formalism for Object–Relational Databases

B.N. Rossiter & D.A. Nelson
Computing Science
Newcastle University, England NE1 7RU

M.A. Heather
University of Northumbria at Newcastle

November 1994

Abstract

Category theory has been developed over the last 50 years as a multi-level mathematical workspace capable of modelling real-world objects. Categories of objects are manipulated in geometric logic by a single concept represented by the arrow.

The category of products is an important instance of the universal concept of a limit now recognized to exist in many contexts. The product model provides a natural extension from relational structures on sets to a full formal description of features such as classes, objects, association abstraction, inheritance, views and query closure. The benefit for databases is that these can all be integrated formally through the arrow concept.

About the author

Nick Rossiter is lecturer in the Department of Computing Science with particular interests in databases and systems analysis.

David Nelson is a Ph.D. student in the Department of Computing Science with support from ESPRC and interests in databases and category theory.

Michael Heather is senior lecturer in law where he has been responsible for computers and law since 1979.

Suggested Keywords

object–relational database model, object–oriented databases, category theory, products, posets, subcategories.

1 Background

Databases have always had a formal background. This has had important advantages in proving that data operations are carried out rigorously, in universality of applicability and in the agreement of common standards. The basic database models are firmly based on standard mathematics: hierarchies, directed graphs (networks), relations and functions.

The difficulty for the development of database technology has been that the functionality provided by concepts such as relations is not adequate to deal with real–world requirements. This has resulted in the development of a new breed of databases – the object–oriented – with limited mathematical pedigree compared to the existing models but with very much richer structures which offer the potential for users to define and manipulate increasingly sophisticated structures.

An inadequate mathematical basis for object–oriented databases as pointed out by Kim [1990] has limitations: the ability to prove rigorously that a system works universally is difficult and it is not easy to develop common models and standards without an accepted mathematical framework. In addition, a few areas have proved very difficult to implement which are taken for granted in current databases: views, where different users can see the same stored data in different ways; closure, where the result of a database query is a database structure which can be manipulated further by the system; and generalized query languages, where a high–level language can be employed to answer *ad hoc* queries.

Partly because of the relatively informal nature of object–oriented databases, an alternative strand of development has been that of the object–relational model where attempts are made 'to obtain the best of both worlds' by combining the two approaches. In this model, relational concepts such as sets, relations and functions are included as well as object–oriented concepts of abstraction and behaviour. Examples of this approach are found in Postgres [Stonebraker & Rowe 1986], Montage, Matisse and UniSQL [Kim 1994].

Category theory is a relatively new and very powerful form of mathematics which we believe has the capability for providing an effective and natural formalism for object–based databases. Categorical constructions provide a multi–level capability matching the three–level database architecture of ANSI/SPARC [Tsichritzis 1978] and by being based on the arrow as the basic concept, give a powerful representation of the many mappings involved in a database system. One of the attractions of category theory is its ability to combine diagrammatic formalisms as in geometry with symbolic notation as in algebra: in computing science, diagrams are a common way of mastering complexity and symbolic notation is used for proofs and computation.

1.1 Early Database Work with Categories

The theoretical database models developed by Ullman [1988], with their emphasis on morphisms, can be considered as an intuitive form of category theory developed within the customized context of databases. Ullman's concept of F^+ involves the set of functions both prescribed and implied in a relationship. F^+ includes given functional dependencies, all dependencies on projection and pseudotransitivities. F^+ involves collections of arrows which are represented cumbersomely in set theory but which are directly handled in category theory.

The Logical Data Model of Kuper & Vardi [1993] also uses categories in an intuitive form with products, power sets and unions as basic mathematical structures and a clear separation between names and values. The scope of their model could be made more general by employing formal categories so that the model is naturally extensible to handle further types of structure. Multi–level facilities in category theory would also assist in formalizing mappings between the various structures.

When investigating the relationship between the functional, relational, E–R and DBTG models, Sibley & Kerschberg [1977] used a categorical representation in binary product form of relationships based on the work of Mac Lane [1971].

Early work on the representation of the network and hierarchical database models in category theory [Cartmell 1985] addresses the construction of networks and trees in categorical terms but does not deal with many important aspects of databases such as object structures and manipulation. This work also pre-dates recent text books and papers [Barr & Wells 1990; Freyd & Scedrov 1990; Dennis–Jones & Rhydeheard 1993] which have made the subject more accessible and which have emphasised categorical concepts which are very relevant for database construction: the basic properties of categories and functors; the treatment of *posets* (partially–ordered sets) as categories; and *products* and *limit*.

More recently, Lellahi & Spyratos [1991; 1992] have applied category theory to complex object structures, the relational model, functional dependencies and a limited number of features of the object–oriented paradigm. Their work on functional dependencies with the categorical concept of *limit* shows the potential for category theory in ensuring consistency. However, they have not realised to any great extent the full potential of category theory in database work as they have tended to produce their own formalisms based more on graph theory than on categorical abstractions and have neglected a number of areas paramount to an object–relational model: normal forms, the association abstraction and querying and views in a conceptual manner. As we shall see, the concepts of association and queries can be directly and simply modelled in a rigorous formal manner by *pullbacks* and *subcategories* respectively.

The work presented here is a continuation of our earlier studies on comparing the use for database theory (including access methods) of category theory, Z and set theory [Rossiter & Heather 1992], in expressing database architecture and functional dependencies in category theory [Rossiter & Heather 1993] and in prototyping an example for a student administration database in category theory [Nelson, Rossiter & Heather 1994]. The work also forms a companion to current studies of legal norms, rules and laws expressed in category theory [Heather & Rossiter 1994a] where interoperability between heterogeneous systems is one of the long–term aims, and to studies on representing natural language in category theory [Heather & Rossiter 1994b].

1.2 Relationship to Functional Models

The functional model has been proposed as a suitable formal and practical basis for object–oriented databases [Gray, Kulkarni & Paton 1992]. As the fundamental construction in category theory is the arrow, we should expect our constructions to resemble the functional model more closely than any of the other semantic models. While this turns out to be true, important differences emerge such as the much stronger framework in the categorical approach for multi–level constraints as in the intension–extension mapping and in typing; for inter–object relationships; and for keys and functional dependencies. The query language that we are developing is based on the functional model of DAPLEX [Shipman 1981] but mappings may be between categories as well as between objects giving higher–order operations with closure as will be described later. The need for higher–order logic in databases has already been noted, for example see Beeri [1992].

1.3 Appropriateness of Formalisms

In developing the object models presented here, our motivation has been that the ideal computing formalism is natural, clarifies thought and resolves controversy in the application world, indicates new areas or facilities for extending an approach, employs the minimal number of constructions in an orthogonal manner and is based upon standard mathematics. The extent to which we meet this ideal is reviewed at the end of the paper.

2 Categorical Concepts

Category theory can represent all standard mathematical structures and manipulations as predefined categories. There is therefore no limit placed on category theory in its ability to cope with detail. Further, with the facility to specify formally transformations between different types of mathematics, category theory provides a powerful way of modelling complex systems with heterogeneous structures as is found in database architecture.

2.1 Categories

Category theory is based not on the set as a fundamental but on the concept of a morphism, generally thought of as an arrow and represented by $\longrightarrow$ [Mac Lane 1971]. Manes & Arbib [1986] consider that the morphism can be regarded as an imperative arrow for the purposes of computing science. The arrow represents any dynamic operation or static condition and can cope therefore with descriptive/ prescriptive equivalent views. For example, the arrow is a generalization of mathematical symbols like $=, \in, \subset, \leq, f(x), \ldots$ with the usual respective meaning of equality, membership, partition, comparison, functional image, etc.

The arrow can never be free–standing: it must have some source and target, often conveniently named domain (dom) and codomain (cod) respectively. A category is a collection of arrows.

The basic constructs of category theory are quite simple:

1. The identity arrow 1_A identifies an object A. That is,

$$1_A : A \longrightarrow A$$

2. Arrows are composable if the codomain of the one forms the domain of the other.

3. Identity arrows can be distinguished by unitary composition with some arrow f.

$$f : 1_A \longrightarrow 1_B \qquad \text{or simply} \qquad A \xrightarrow{f} B$$

4. Composition of arrows is associative. Arrows may be composed so that the codomain of one arrow may become the domain of another. Standard category theory requires composition to be associative. For the arrows:

$$A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D \xrightarrow{i} E$$

$$i \circ (hgf) = (ih) \circ (gf) = (ihg) \circ f$$

Conventionally then a category in this context is a collection of arrows between objects which may be named. Below we show a category **C** with two arrows f and g. Where categories are given names, we use the convention throughout the paper of denoting them in bold upper–case letters.

$$f : A \longrightarrow B \qquad g : C \longrightarrow D$$

An object in a category $\mathbf{C}$ where there is one and only one arrow from every other object to it is known as the final or terminal object of $\mathbf{C}$. This may be denoted by $\mathbf{1}$ for the whole category, more precisely with the subscript $\mathbf{1_C}$ where $\mathbf{C}$ now represents the whole category $\mathbf{C}$. Dually (or oppositely) to the final object there may exist a corresponding initial object where there is an arrow from it to every other object in the category.

The derivation of a 'subset' of objects is represented by the subobject concept. The object S is a subobject of A if it contains some of the members of A. See the section Typing for further information on the subobject concept.

The *hom–set* of arrows between objects p and q in a category $\mathbf{C}$ is written $\mathrm{Hom}_{\mathbf{C}}(p, q)$ and represents the set of arrows between the two objects.

A number of universal categories may be recognized to represent well–known mathematical structures. These include the category of sets (**SET** – where the objects are sets and the arrows are total functions) and a poset category (where the objects are compared by arrows representing partial orderings). The category **POS** deals with the universe of posets.

A number of types of arrow are defined in category theory which generalize the concepts in set theory of injection (1:1 mapping), surjection (onto) and bijection (1:1 and onto) to apply to any category [Manes & Arbib 1986]. The categorical terms are monic, epic and isomorphic respectively. Arrows that are monic, epic or isomorphic are typed in the same way as objects [see Typing later].

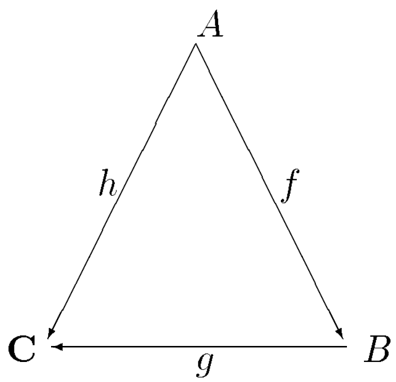

Figure 1: Simple Diagram Chasing

A diagram for a category can be represented as a series of connected triangles. Each triangle may comprise two paths between two objects as shown in Figure 1 – one a composition of two arrows g and f, the other a single arrow h. Then we take commutativity to mean that a comparison of the two paths can be represented as an equality $h = g \circ f$.

The way that the equation is written is conventional. The other order would be the dual. The nature of proof in category theory should be emphasised. The diagram is a formal diagram. It is a geometric representation equivalent to an expression in algebra. We are in constructive mathematics and the only proof needed is the proof of existence. Therefore so long as it can be shown that the entities belong to formal categories [Freyd 1964], proof up to natural isomorphism is by composition. A formal diagram is in effect a *QED (Quod Erat Demonstrandum).*

2.2 Subcategories

In a subcategory **E** of a category **D**, all of the objects and arrows of **E** are to be found in **D**, the source and targets of arrows in **E** are the same as those in **D**, the identity arrows are the same for objects in **E** as in **D** and composition rules for arrows in **E** are the same as in **D**. **E** is a subcategory of **D** (with collection of objects $\mathrm{obj}_{\mathbf{D}}$) if for objects p, q in **E** (collectively termed $\mathrm{obj}_{\mathbf{E}}$) we have

$$\mathrm{obj}_{\mathbf{E}} \subseteq \mathrm{obj}_{\mathbf{D}} \text{ and } \mathrm{Hom}_{\mathbf{E}}(p,q) \subseteq \mathrm{Hom}_{\mathbf{D}}(p,q) \quad (\forall p, q \in \mathrm{obj}_{\mathbf{E}})$$

Clearly, subcategories in general only contain some of the objects and arrows of their parent categories. However, there are two examples of special interest. If **E** has the same arrows for each pair of objects as in **D**, **E** is termed a *full* subcategory of **D**. If **E** has the same objects as **D**, it is termed a *wide* subcategory of **D**. Any category is a full wide subcategory of itself.

The terms category and subcategory are relative so that a family of categories, with inclusion dependencies between them, can be placed in a partial order with the arrows representing ordering by inclusion. ¿From a functional perspective, the arrows are in fact functors mapping one category to another as described below.

2.3 Functors

An arrow between categories is termed a functor if it satisfies some structure–preserving requirements: each arrow and object in the source category must be assigned (as in homomorphisms); identity morphisms in the source category must be preserved and for each pair of arrows in the source category, $f : A \longrightarrow B$ and $g : B \longrightarrow C$, then $F(gf) = F(g) \circ F(f)$ in the target category where F is the functor. This type of arrow provides the facility for transforming from one category type to another category type.

Functors are therefore basically structure–composing and –preserving morphisms from a source category to a target category. An obvious case is when the shape of the target category is determined by the functor, that is it accomodates all assignments from the source category and has no other structure of its own. However, functors can also be inclusive (or injections) so that the target category contains more structure than the source category. The functor from a subcategory onto the category on which it is founded is an example of such a morphism which we find is very pertinent for database modelling. Such morphisms are free functors.

It is also possible to construct what are known as underlying functors, in carefully controlled circumstances, which forget some of the structure of the source in forming the target category, for example a transformation from a graph to its underlying sets. Such functors do provide a total mapping from one category to another but some of the structure is mapped to bottom $\perp$.

It is possible to construct arrows (functions) from one category to another that are not functors [Freyd & Scedrov 1990, at page 5] but we always use functor constructions as otherwise our formalisms are outside category theory.

2.4 Typing

Category theory has a naturally inherent concept of *type*. Discrete items are identified by the single category $\mathbf{1}$. Therefore an element in a set $a \in A$ is represented categorically by $a : \mathbf{1} \longrightarrow A$. Typing is added by indicating the category (i.e. some pool of values in set theory extensions) from where the item is taken. For example $a : \mathbf{1_C} \longrightarrow A$ (or more simply $\mathbf{C} \xrightarrow{a} A$) makes the element a in set A of type $\mathbf{C}$. However, in general, A need not be a member of an object in the category **SET** but may belong to a more general category.

Typing can be readily based on arrows as well as object values. Monic arrows in a category or object can be considered as of type $\mathbf{M}$ where $\mathbf{M}$ is a category representing the universe of monics. In each of these examples, the arrow is relating categories and is strictly a functor, emphasising the need for multi–level capabilities for typing.

In imperative programming languages, the concept of *range* is used to indicate that a variable may only take a subset of the values specified by a type. The equivalent categorical concept is the subobject, where each subobject, say S, is related through an injective function, say i, to an object O. There is a monic inclusion function i from S to O. Note that although we can regard S as a 'subset' of O, the better categorical view is that the type of S is related to that of O by the type conversion function i. For instance if S are integers and O are reals, the function i, can be thought of either as the inclusion mapping from integers to reals or a type conversion function from integer to real.

2.5 Product and Projection

Two operations common in relational algebra, product and projection, are represented directly in category theory through the construction of a cone. A cone is an open triangle comprising three objects for example, A, C and $A \times C$ where the product $A \times C$ is the vertex of the cone as shown in Figure 2. The projection arrow π operates in either a left (π_l) or a right (π_r) context, depending on which part of the product is being selected.

$$
\begin{tikzcd}
& A \times C \arrow[dl, "\pi_l"'] \arrow[dr, "\pi_r"] & \\
A & & C
\end{tikzcd}
$$

Figure 2: Product Cone for Objects A and C

In strict category terms, the cone as presented in Figure 2 does not appear to commute but it may alternatively be presented as in Figure 3 where for any object V and arrows $q_1 : V \longrightarrow A$ and $q_2 : V \longrightarrow C$, there is a product U with projections A and C such that the diagram commutes, that is the two equations hold:

$$\pi_l \circ q = q_1$$

$$\pi_r \circ q = q_2$$

U is the universal product of $A \times C$.

$$
\begin{tikzcd}
& V \arrow[dl, "q_1"'] \arrow[d, "q"] \arrow[dr, "q_2"] & \\
A & U \arrow[l, "\pi_l"] \arrow[r, "\pi_r"'] & C
\end{tikzcd}
$$

Figure 3: Commuting Product Cone for Objects A and C

2.5.1 Coproduct and Inclusion

The dual concept to the product is the coproduct in which all the arrows in the commuting product cone discussed earlier are reversed in direction. The object S (disjoint union) replaces the object U (universal product). The coproduct of A and C, written $A + C$, is the disjoint union of A and C and is usually represented by the cone in Figure 4 where i_l and i_r are inclusion arrows $i_l : A \longrightarrow A + C$ and $i_r : C \longrightarrow A + C$ respectively:

2.5.2 Finite Products

The preceding examples have been of binary products. The concept of cones is, however, extensible to n–ary products both for multiplication and addition [Rossiter & Heather 1993].

2.5.3 Pullbacks

So far we have considered only the universal product U where the complete (unrestricted) product of two objects is considered. An important product in practice is the pullback or fibred product where a product is restricted over some object. If A and C both have arrows to some common B as $A \xrightarrow{f} B$ and $C \xrightarrow{g} B$, then the subproduct of A and C over B written as $A \times_B C$ may be represented by:

where $f(a) = g(c)$ and $f(a), g(c) \in B, a \in A, c \in C$. This diagram commutes in that $f \circ \pi_l = g \circ \pi_r$. In this paper, we generally say a pullback is of two objects over another. This is actually a kind of synecdoche, as in strict category theory, the pullback is expressed in terms of arrows: π_l is the pullback of f along g in the above example.

2.5.4 Pushouts

Pushouts are an extension of the coproduct concept. They give us the facility to construct amalgamated sums which may be more complex in form than simple disjoint unions. An amalgamated sum S is constructed by a pushout from objects A and C where $A, C \supset B$ and the functions $i_l : B \longrightarrow A$ and $i_r : B \longrightarrow C$ are monic as shown in Figure 6:

The nature of S depends on the choice of functions i_l, i_r, f and g. We can relax the requirement for i_l and i_r to be monic in which case S becomes a sum identifying part of one object with a part of another in a general manner.

2.5.5 Limits

The concept of a limit has only been fully explored theoretically in the last 30 years. Limits play a crucial role in representing database constructions in category theory because they can be used to enforce local and global consistency. In particular they can be used to ensure that extensions comply with the constraints specified in the intension.

There are a number of ways that a limit is defined depending on the perspective. Barr & Wells [1990] consider a limit may be a terminal object of a family of cones. Perhaps the most suited to database work is that of Freyd & Scedrov [1990] where a limit, if it exists, is considered to be the infimum for all product cones of a family where every cone in the family commutes: the limit exists only if every cone in the family commutes. This perspective requires the cones to be placed in a poset where the canonical form of the i^{th} cone D_i is given by the diagram shown in Figure 7.

In this figure, P is a product, π_0 is the projection coordinate of the first element in the product, π_i is the projection of the i^{th} component of the product and K and N_i are projected objects. f_i is some arrow which may or may not cause the cone to commute, that is it is not assumed that $f_i \circ \pi_0 = \pi_i$.

In detail, to determine whether a limit holds, we consider a collection of c cones $(D_i \mid 1 \leq i \leq c)$ as shown in Figure 7. The postulated limit (infimum) is the vertex, P, of the cones as P precedes ($\leq$) all other objects. Therefore, we determine which cones in the family of cones actually commute by determining for each value of i whether $f_i \circ \pi_0 = \pi_i$. If all c cones commute, P is the limit; otherwise we have no limit.

If a source category with a limit is related to a target category by a functor, it is often important to check that the limit also exists in the target category. If it does, the functor preserves limits.

Corresponding to the notion of limit, there is the concept of colimit where the supremum (least upper bound) is sought rather than the infimum. Products and pullbacks may have limits and coproducts and pushouts colimits.

An important example is given later of limits where in Figure 7 we treat P as the product of persistent attributes in a class, K as the identifier (key), N_i as a non–key attribute and f_i as a postulated functional dependency.

2.6 Natural Transformations

An arrow between functors is termed a natural morphism (or transformation) as shown in Figure 8 where there is a natural transformation α from K to L, written:

$$\alpha : K \longrightarrow L$$

This natural transformation assigns to each source object A a target arrow

$$\alpha_A : K(A) \longrightarrow L(A)$$

such that for each source arrow $A \longrightarrow B$ the target square shown in Figure 9 commutes. This is the covariant form. There also exists the corresponding contravariant.

Note the tight inter-relationship between the levels in category theory: morphisms and objects of categories at the lowest level are part of the expressions at the highest level of natural morphisms. A special case of natural transformation is the concept of natural isomorphism where, in the example given, the composites $\alpha \circ \beta$ and $\beta \circ \alpha$ are the identity natural transformations of L and K respectively. This links to another mathematical approach where α is regarded as an isomorphism of a model of categories giving connections to model theory. In effect, a natural transformation is a natural isomorphism when every component of the transformation is an isomorphism.

3 The Product Data Model

Using the categorical constructions introduced so far, we now construct the product data model to capture the semantics of object–relational databases. The minimum objectives for our data model are:

1. A clear separation between intension (class) and extension (object) structures with a rigorous mapping defined between them.
2. Object encapsulation.
3. An orthogonal definition language for functions within a class to include both functional dependencies and methods, the naming and typing of all functions and attributes within each class.
4. Constraints on class structures as represented by the concept of primary and candidate keys, normal forms such as BCNF (Boyce–Codd Normal Form) and functionality and membership class in object (E–R) models.
5. The standard information system abstractions formulated in the 1970s [Smith & Smith 1977] and which are prime targets of current object–oriented databases [Atkinson et al 1992] and object–relational systems [Stonebraker & Rowe 1986]. These abstractions include inheritance (generalization and specialization); composition such as aggregation; classification and association.
6. Message passing facilities between methods located in any part of the system.
7. A query language which can provide results with closure: the output from a query can be held in a class–object structure which ranks equally *pari passu* with other such structures already existing in the database.
8. A multilevel architecture like that in the ANSI/SPARC standard [Tsichritzis 1978] with definitions of views, global schemata and internal structures and the mapping between them.

All symbols declared in the formalism are itemized in Appendix I along with a brief description of their purpose.

3.1 Classes

3.1.1 Basic Structures

The class construction represents the intension for a database. Each class is represented by a category $(\mathbf{CLS_i} \mid 1 \leq i \leq c)$ where c is the number of classes in the database. The class name is the name of the category. Category theory keeps distinct intensional and extensional forms of a data dictionary. For example, $\mathbf{1_{CLS}}$ types a database entity in general, $\mathbf{1_{CLS_1}}$ types the database entity *suppliers* and $\mathbf{1_{CLS_2}}$ types the database entity *parts*. Then $\mathbf{CLS_1}$ is the class of *suppliers* and $\mathbf{CLS_2}$ is the class of *parts*.

Each category $\mathbf{CLS_i}$ is a collection of arrows where an arrow may represent an action (transformation) or an association. The former represent methods and the latter dependencies (functional, inclusive, transitive, etc). Arrow names are the names of methods and dependencies.

Each arrow has a domain and a codomain. Within the universal category **SET**, domains and codomains are sets but in general they can be of arbitrary complexity. Domain and codomain names are the names of variables defined within the class. In the next section, we describe the identification of one or more domains as candidate keys and the selection of one of these as the primary key. The types of arrows, domains and codomains are defined by naming the categories upon which their data types are based. All arrow constructions as regards composition and association must conform to the four axioms of category theory given earlier.

Formally, each category $\mathbf{CLS_i}$ is a collection of k arrows or morphisms $F = \{f_j \mid 1 \leq j \leq k\}$ where f_j has domain $\mathrm{dom}(f_j)$ and codomain $\mathrm{cod}(f_j)$. The domain and codomain names are not necessarily distinct. $\{\mathrm{dom}(f_j) \bigcup \mathrm{cod}(f_j) \mid 1 \leq j \leq k\}$ is the set of variables in the class which we call V with cardinality q. In order to permit complex actions and dependencies, domains may be structured, that is contain more than one variable. For database applications, codomains are normally considered to comprise a single variable although category theory itself need not be restricted to minimal covers [Freyd & Scedrov 1990] but can cope well with open covers [Mac Lane & Moerdijk 1991]. Variables may be either *persistent variables* given by a set $A = \{a_j \mid 1 \leq j \leq n\}$ comprising the persistent component of the class, or *memory variables* given by a set $U = \{u_j \mid 0 \leq j \leq n'\}$ comprising the transient component of the class. Note that A and U are both subobjects of V and $n + n' = q$.

Using our earlier notation, V corresponds to $\mathrm{obj}_{\mathbf{CLS_i}}$ and F to $\mathrm{Hom}_{\mathbf{CLS_i}}(v, v')$ for all $v, v' \in V$.

Arrows are typed, for example the collection of arrows $D = \{d_i \mid 0 \leq i \leq r'\}$ represents arrows occurring in the universe of functional dependencies and $M = \{m_i \mid 0 \leq i \leq s\}$ represents arrows occurring in the universe of methods. Note that D and M are both subobjects of F and $r' + s = k$.

Functional dependencies involve only persistent variables as their domains and codomains. Minimal covers are assumed: domains may be composite involving more than one persistent variable while codomains are restricted to being single persistent variables. Therefore for each functional dependency, we have $d_i : x \longrightarrow y$, $x \in \wp A, y \in A$, that is, x is a member of the powerset of A. Although y is a singleton variable, this does not mean that its structure is simple. y could represent structures such as multivalued sets, lists or arrays. We deduce the set of persistent variables E that participate in functional dependencies, as domain or codomain, by $\{\mathrm{dom}(d_i) \bigcup \mathrm{cod}(d_i) \mid 0 \leq i \leq r'\}$. Note that $E = A$ only in the special case when all domains in D are single attributes and every attribute in A is involved in a dependency.

Functional dependencies can be composed. Thus the composition of $d_1 : \{a\} \longrightarrow \{b\}$ and $d_2 : \{b\} \longrightarrow \{c\}$ gives $d_2 \circ d_1 : \{a\} \longrightarrow \{c\}$. Such compositions are represented without difficulty in the partially–ordered structures that we introduce later as a natural consequence of the transitivity rule (if $\{a\} \leq \{b\}$ and $\{b\} \leq \{c\}$, then $\{a\} \leq \{c\}$). However, in some circumstances, partial composition occurs. For example:

$$d_3 : \{a, e\} \longrightarrow \{c\}$$

$$d_4 : \{b, c\} \longrightarrow \{d\}$$

where $\mathrm{cod}(d_3) \subset \mathrm{dom}(d_4)$. These partial compositions generate a new collection of arrows termed pseudotransitivities obtained by:

1. identifying partial compositions as above where, for two arrows d_i and d_j, $\mathrm{cod}(d_i) \subset \mathrm{dom}(d_j)$;
2. determining variables needed to augment the dependency d_i to achieve total composition as $z = \mathrm{dom}(d_j) - \mathrm{cod}(d_i)$;
3. augmenting both $\mathrm{dom}(d_i)$ and $\mathrm{cod}(d_i)$ with z to give a new function:

$$d'_i : (\mathrm{dom}(d_i) \bigcup z) \longrightarrow (\mathrm{cod}(d_i) \bigcup z)$$

4. composing d_j and d'_i (that is $d_j \circ d'_i$) to give a pseudotransitivity arrow:

$$p_i : (\mathrm{dom}(d_i) \bigcup z) \longrightarrow (\mathrm{cod}(d_j))$$

So in the above example, we determine through pseudotransitivity that $\{a, b, e\} \longrightarrow \{d\}$. After each pseudotransitivity arrow has been identified, it must be determined whether any further ones can be deduced: the process is iterative. The result is a collection of pseudotransitivity arrows $P = \{p_i : x \longrightarrow y\}$ $(x \in \wp A, y \in A, 0 \leq$

$i \leq r''$). The set of variables E' that participate in pseudotransitivities is given by $\{\mathrm{dom}(p_i) \bigcup \mathrm{cod}(p_i) \mid 0 \leq i \leq r''\}$.

For each arrow that is a method, $m_i : x \longrightarrow y (0 \leq i \leq s)$, then $x \in \wp V, y \in V$, that is the domain may be any subobject of the persistent and memory variables and the codomain is a singleton persistent or memory variable. If required, memory variables can be considered as derived [Shipman 1981] or virtual variables which can be manipulated by database operations.

The typing is indicated by a collection of mappings $\{h : \mathbf{1_{TYP}} \longrightarrow H\}$ where H represents the name of either an arrow in F or an object in V, h is an instance of H and **TYP** is the category upon which the type of H is based. The category construction naturally provides an encapsulation of attributes and methods for a class.

3.1.2 Identifiers

As we shall see later, we need a way of deriving identifiers for use in our relationship representations. Identifiers can be natural (primary keys) or system assigned (object identifiers). Both the forms of identifiers are initial objects in categories as there is an arrow from the identifier to every other object in the category. Initial objects are normally denoted by 0 in category theory – hence we adopt K_0 as the notation for the key. The key K_0 is derived as shown below for each class category **CLS** [Rossiter & Heather 1993] following a lattice approach [Demetrovics, Libkin & Muchnik 1992] rather than an algorithmic one [Ullman 1988]. The lattice formalism lends itself more to a categorical approach with its emphasis on poset constructions. We employ the identifiers and dependencies to test whether our class structures correspond to BCNF (Boyce–Codd Normal Form). This normal form is adopted because it is more powerful than 3NF and can easily be deduced from functional dependencies making it ideally suited to a lattice approach. The procedure is as follows:

1. Generate the poset category **PRJ** with elements $p, q \in \wp A$ and projected orderings $(p \times q \leq \pi_l(p \times q); p \times q \leq \pi_r(p \times q))$ as the arrows, that is to take the projections by applying the free functor $G : A \longrightarrow$ **PRJ**.

2. Generate the poset category **DEP** with elements $p, q \in E$ and arrows $\{d_i \mid 0 \leq i \leq r'\}$ as the orderings, that is to apply the free functor $G' : E \longrightarrow$ **DEP**.

3. Generate the poset category **PSU** with elements $p, q \in E'$ and arrows $\{p_i \mid 0 \leq i \leq r''\}$ that is to apply the free functor $G'' : E' \longrightarrow$ **PSU**.

4. Take **DEP** and **PSU** representing respectively the non–trivial functional dependency arrows declared in the previous section and the pseudotransitivity arrows (dependencies inferred from the postulated functional dependencies and their combinations [Ullman 1988]) between $p, q \in \wp A$. Inject these into **PRJ**, that is add the arrows of **DEP** and **PSU** to those already in **PRJ**.

5. Test that **PRJ** is still a poset by checking for anti–symmetry (if $p \leq q$ and $p \geq q$, then $p = q$). Cycles in the ordering would give a *preset* [1] (pre–ordered set) which would need to be partitioned by applying a suitable quotient functor to produce a number of posets which can then be handled collectively. Each **PRJ** as a poset corresponds to an F^+ [Ullman 1988]. Each class (record–type) has its own F^+.

6. The infimum or meet of the elements of A in **PRJ** ($\bigwedge A$) is the primary key PK. If there is no infimum, the set of maximal lower bounds is the set of candidate keys CK.

7. The class is in BCNF if each source of a functional dependency arrow is PK or is a member of CK.

8. The identifier K_0 is either PK or a user–selection from CK. When it is necessary to distinguish the keys for each class, consider K_0^i as the identifier for the i^{th} class $\mathbf{CLS_i}$.

9. Other persistent attributes may be labelled $K_1 \ldots K_r$ where $r = n - c$ with c as the number of attributes in the key. In the simplest situations, $r = r'$, where r' is the cardinality of the set of dependencies D but in many cases such as classes with no dependencies or with multiple candidate keys or with classes that are not in BCNF, this will not be true.

Alternatively, an object identifier can be defined as the identity functor on a category, for example $\mathbf{1_{CLS_i}} : \mathbf{CLS_i} \longrightarrow \mathbf{CLS_i}$.

Our final task is to transfer our results from **PRJ** into the class category **CLS**. This is necessary as, particularly if the key is composite, K_0 is not guaranteed to be a variable in the class **CLS**. We apply an injective functor from a view of the poset **PRJ** into **CLS**. The category that we inject into **C** is the exponential construction $\mathbf{PRJ}^{K_0}$ (the arrows of **PRJ** with K_0 as source). **CLS** now includes the key K_0 and the arrows from K_0 to each of $K_1 \ldots K_r$. If therefore K_0 was not already in **PRJ**, the injection increases the number of persistent variables n in **CLS** by one and the number of arrows k by r, that is $n \longleftarrow n + 1$ and $k \longleftarrow k + r$.

3.2 Relationships

The association abstraction between classes is represented in object models by notation based on the Entity–Relationship approach. In categorical terms, the E–R model is represented by pullbacks. In Figure 5, A and C are entity–types or classes and B is a relationship between them. Instances of the relationship occur when $f(a) = g(c)$.

[1] A radical alternative approach that we are working on, at the moment, is to allow the starting relation to be a preset and to map it automatically into a family of posets satisfying BCNF

Instances for B are of the form $\{< a, c, b >| \ f(a) = g(c), b \in \wp B\}$ where b is any information carried by the link and is an element in the powerset of B (that is a subset of B).

Our pullback is on class identifiers K_0^i as initial objects in categories representing classes. To give an example, consider the pullback of K_0^1 and K_0^2 over O shown in Figure 10, where K_0^1 and K_0^2 are initial objects in the categories for the entity–types *supplier* ($\mathbf{CLS_1}$) and *parts* ($\mathbf{CLS_2}$) respectively and O is a relationship *orders* between suppliers and parts.

The collection of relationships in a database intension is represented by a family of pullback categories ($\mathbf{ASS_i} \mid 0 \leq i \leq p$) where p is the number of relationships. We next include information to cover aspects such as functionality and membership class. First let us consider the nature of each object and arrow in the category:

- K_0^1 is the identifier for the *supplier* class $\mathbf{CLS_1}$.
- K_0^2 is the identifier for the *parts* class $\mathbf{CLS_2}$.
- O is the relationship *orders* representing all instances of this type of association between suppliers and parts. Instances for O are of the form $\{< k_0^1, k_0^2, o >| \ f(k_0^1) = g(k_0^2), k_0^1 \in K_0^1, k_0^2 \in K_0^2, o \in \wp O\}$ where o is information such as quantities and dates of orders and is an element in the powerset of O (or is a subset of O representing that set of orders for a part from a particular supplier). O can be considered as a simple structure including j properties for orders $\{o_i \mid 1 \leq i \leq j\}$.

  Alternatively, where there is considerable complexity in the structure and operations of O, it would be desirable to create a category, say $\mathbf{CLS_3}$, to handle as a class the internal complexity of the orders and to include in the pullback structure the identifier for this class K_0^3 defined as pairs of values $< k_0^1, k_0^2 >$ as a surrogate for the orders category.
- $K_0^1 \times_O K_0^2$ is the subproduct of K_0^1 and K_0^2 over O: it represents the subset of the universal product $K_0^1 \times K_0^2$ that actually occurs for the relationship O.

By considering the nature of the arrows we can now provide more information concerning the relationship O:

- The arrow f maps from identifier K_0^1 to the relationship O. It represents associations between suppliers and orders.
- The arrow g maps from identifier K_0^2 to the relationship O. It represents associations between parts and orders.

- When $f(k_0^1) = g(k_0^2)$, we have an intersection between the two associations, that is a supplier and a part both point at the same order: a set of such orders is associated with a particular supplier-part pair.

- The arrow π_l is a projection of the subproduct $K_0^1 \times_O K_0^2$ over K_0^1 representing all suppliers.

  - If this projection arrow is *onto* (epimorphic or epic in categorical terms) then every supplier appears at least once in the subproduct. Thus every supplier participates in the relationship and the membership class of K_0^1 is indicated as *mandatory*. If, however, π_l is not epic, then not every supplier participates in the relationship and the membership class of K_0^1 is indicated as *optional*.
  - If this projection arrow is *one–to–one* (monomorphic or monic in categorical terms) then each supplier appears just once in the subproduct. If, however, π_l is not monic, then a supplier may participate more than once in the relationship.
  - If π_l is both monic and epic, the projection is said to be isomorphic with each supplier appearing once in the subproduct and K_0^1 having *mandatory* participation in the relationship.

- The arrow π_r is a projection of the subproduct $K_0^1 \times_O K_0^2$ over K_0^2 representing all parts.

  - If this projection arrow is epic, then every part appears at least once in the subproduct. Thus every part participates in the relationship and the membership class of K_0^2 is indicated as *mandatory*. If, however, π_r is not epic, then not every part participates in the relationship and the membership class of K_0^2 is indicated as *optional*.
  - If this projection arrow is monic, then each part appears just once in the subproduct. If, however, π_r is not monic, then a part may participate more than once in the relationship.
  - If π_r is both monic and epic, it is said to be isomorphic with each part appearing once in the subproduct and K_0^2 having *mandatory* participation in the relationship.

3.2.1 Significance of Monic Projections

Using the values for O given in the text, the following diagrams illustrate the conclusions for functionality and membership class from testing for monics and epics:

Consider first the situation in Figure 11 where:

$$O = \{< 1, 9, \{< 222, 6 >, < 301, 8 >\} >, < 2, 9, \{< 224, 9 >, < 287, 12 >\} >\}$$

indicating that there are two relationships between $k_0^1 \in K_0^1$ and $k_0^2 \in K_0^2$:

- $k_0^1 = 1$ and $k_0^2 = 9$ are associated with the subset of orders $\{< 222, 6 >, < 301, 8 >\}$ where 222 and 301 are order numbers and 6 and 8 are quantities. The subset of orders o is in the powerset of orders ($o \in \wp O$).
- $k_0^1 = 2$ and $k_0^2 = 9$ are associated with the subset of orders $\{< 224, 9 >, < 287, 12 >\}$ where 224 and 287 are order numbers and 9 and 12 are quantities.

Note that with π_l being monic, this means that each element $k_0^1 \in K_0^1$ appears once in the subproduct $K_0^1 \times {}_O K_0^2$. As π_l is epic, this means that every element $k_0^1 \in K_0^1$ appears in the subproduct $K_0^1 \times {}_O K_0^2$ giving K_0^1 mandatory membership in the subproduct. Because π_r is not monic, some elements $k_0^2 \in K_0^2$ appear more than once in the subproduct $K_0^1 \times {}_O K_0^2$. As π_r is not epic, this means that not every element $k_0^2 \in K_0^2$ appears in the subproduct $K_0^1 \times {}_O K_0^2$ giving K_0^2 optional membership in the subproduct.

In conventional E-R model terminology, the types of arrows indicate an N:1 relationship $K_0^1 : K_0^2$, that is each supplier is associated with one part, each part is associated with many suppliers. In our view, a better approach as it is readily extendible to n–ary products is to say that each supplier participates N times in the relationship O and each part once. This technique of measuring the cardinality of participation in the relationship is of increasing popularity in some object models [Elmasri & Navathe 1994]. All parts must participate in the relationship but not all suppliers need do so.

In Figure 12

$$O = \{< 1, 9, \{< 222, 6 >, < 301, 8 >\} >, < 1, 10, \{< 225, 5 >\} >\}$$

π_r is monic so that each part participates once in the relationship and π_l is not monic so that each supplier occurs N times in the relationship (a 1:N relationship $K_0^1 : K_0^2$). The non–epic mappings indicate that it is optional for parts and suppliers to participate in the relationship.

3.2.2 Further Examples

Our normal understanding of supplier/parts data would lead us to expect π_l and π_r to be neither monic nor epic: the relationship is $N : M$ and the membership class of both entity–types is optional. In the table below, further examples with different semantics are given for the relationship of A and C over B as shown in Figure 5:

A	C	B	π_l		π_r		relationship			
			epic	mon	epic	mon	partic A:C	mapping A:C	memb.cl. A	C
Suppliers	Parts	Orders	n	n	n	n	N:M	N:M	o	o
Students	Courses	Take	y	n	n	n	N:M	N:M	m	o
County Councils	District Councils	Within	y	n	y	y	N:1	1:N	m	m
National Ins. No.	Name	Ident.	y	y	n	n	1:N	N:1	m	o
Car	Licence	Possess	y	y	y	y	1:1	1:1	m	m

These show that by examining the type of the projection arrows π_l and π_l, we can determine the following:

- the functionality for participation of entities of a particular type in a relationship given by *partic* – how many times an entity appears in the subproduct;
- the functionality as a mapping ratio between two entity–types given by *mapping* – the normal E–R perspective;
- the membership class of entity–types in a relationship as mandatory m or optional o.

It should be emphasised that the handling of the entity–relationship modelling here is very much stronger than in conventional data processing where the functionality and membership classes are represented by labels. In the categorical model, the functionality and membership class are achieved through typing of the arrows so that the constraints cannot be violated. Categorical structures are universal rather than conventional. There is an underlying functor from a categorical E–R model to a conventional one with structure loss through typing constraints being represented as labels.

3.2.3 Enhancements

So far we have considered binary relations (relationships between two entity–types) and have neglected n–ary and involuted relationships, multiple relationships between the same classes and the abstractions of inheritance and composition. These are readily handled by standard categorical constructions. n–ary relationships are represented by finite products [Rossiter & Heather 1992]. Involuted relationships are handled directly: for example $K_0^1 \times_B K_0^1$ is the subproduct of K_0^1 with itself over the relationship with the object B. Multiple relationships between the same classes are handled by a series of pullbacks over the same two initial objects, for example $K_0^1 \times_B K_0^2$ and $K_0^1 \times_D K_0^2$ represent pullbacks of K_0^1 and K_0^2 over B and D respectively. Inheritance and composition are described below.

3.2.4 Pullback Identifiers

The values for a subproduct in a pullback will always be unique so generally this component of the diagram can be used as an identifier. Therefore in Figure 12 the identifier is $K_0^1 \times_O K_0^2$. Note that, as in the class diagram, the identifier is the infimum of the diagram.

3.2.5 Inheritance

Inheritance in object–oriented terms is the assumption by classes of properties and methods defined in other classes. It is an intensional concept affecting the manner in which classes are created. In categorical terms, this is achieved by the coproduct construction shown in Figure 13 which yields a disjoint union of two or more objects. Consider:

- a category $\mathbf{CLS_3}$ (employers) with set of arrows $\mathrm{Hom}_{\mathbf{CLS_3}} p, q$ between objects p, q and set of domains and codomains $\mathrm{obj}_{\mathbf{CLS_3}}$; and
- a category $\mathbf{CLS_4}$ (managers) with set of arrows $\mathrm{Hom}_{\mathbf{CLS_4}} p, q$ and set of domains and codomains $\mathrm{obj}_{\mathbf{CLS_4}}$.

The coproduct $\mathbf{CLS_3} + \mathbf{CLS_4}$ is the disjoint union of the arrows $(\mathrm{Hom}_{\mathbf{CLS_3}} p, q + \mathrm{Hom}_{\mathbf{CLS_4}} p, q)$ and the domains and codomains $(\mathrm{obj}_{\mathbf{CLS_3}} + \mathrm{obj}_{\mathbf{CLS_4}})$.

In this example, $\mathbf{CLS_3}$ and $\mathbf{CLS_4}$ contain the specific properties and methods for employers and managers respectively and $\mathbf{CLS_3} + \mathbf{CLS_4}$ is the amalgamation of these objects and arrows into a new category which is in effect the specialization of $\mathbf{CLS_3}$ over $\mathbf{CLS_4}$. The arrow s (for subtype) shows the direction of the specialization: $s : \mathbf{CLS_3} \longrightarrow \mathbf{CLS_4}$ (employee has subtype manager). In general, the supertype category will be identified by one or more properties in the data and the subtype category (being a weak entity) by an identity functor to give an object identifier. In more concrete terms, s can therefore be considered as the mapping between the key of the supertype category $\mathbf{CLS_3}$ and the identity functor $\mathbf{1}_{\mathbf{CLS_4}}$ of the subtype category:

$$s : K_0^3 \longrightarrow \mathbf{1_{CLS_4}}$$

Since a coproduct can, in turn, be the base of another cone, it is a simple matter to construct inheritance hierarchies [Nelson, Rossiter & Heather 1994]. The ancestry of each class in the hierarchy is preserved in the construction of pushouts. Note though that, with our scheme at present, multiple inheritance is not permitted as the disjoint union would not include properties or arrows that appeared in both categories at the base of the cone. At present therefore, our model provides inheritance through the arrangement of categories in a partial order restricted to hierarchical constructions rather than the more general poset of Cardelli [1984].

For convenience, we consider the additional g class categories ($\mathbf{CLS_i} : c+1 \leq i \leq c+g$), such as $\mathbf{CLS_3} + \mathbf{CLS_4}$ above, created as coproducts to comprise the family of categories **UNI**.

Polymorphism at its simplest level is achieved by the coproduct construction. Methods defined for $\mathbf{CLS_3}$ as arrows in the set $(\mathrm{Hom}_{\mathbf{CLS_3}} p, q)$ are also available automatically in the set $(\mathrm{Hom}_{\mathbf{CLS_3}} p, q + \mathrm{Hom}_{\mathbf{CLS_4}} p, q)$.

3.2.6 Composition

Composition including aggregation is the creation of new classes from a collection of other classes. The method of composition is flexible varying from standard mathematical operations such as products or unions on classes [Kuper & Vardi 1993] to qualified operations such as relational joins. The basic ways of representing these compositions have already been introduced such as universal product, disjoint union, qualified product and amalgamated sum.

3.3 Typing

Arrows and attributes are typed, as described earlier, by specifying the categories from which their values will be drawn. These categories may be other classes, basic pools of values such as integer and string, or domains of arbitrary complexity such as complex objects, arrows, lists, graphs and sets.

3.4 Message Passing

We consider message passing to be a function from one arrow to another arrow, where the arrows may be within the same category (intra–class) or in different categories (inter–class). This function is best viewed in category theory as a morphism in the arrow category [Barr & Wells 1990] which is written $\mathbf{C}^{\rightarrow}$ to view the arrows of $\mathbf{C}$ as objects in $\mathbf{C}^{\rightarrow}$. For example, suppose the arrow η_j takes a value from an arrow for the method m_k in the class $\mathbf{CLS_i}$ to an arrow for the method m_n in the class $\mathbf{CLS_j}$ where $\mathbf{CLS_i}$ and $\mathbf{CLS_j}$ are not necessarily distinct. This is viewed in the arrow category as a morphism between objects in $\mathbf{CLS_i^{\rightarrow}}$ and $\mathbf{CLS_j^{\rightarrow}}$ as shown below:

$$\eta_j : m_k \longrightarrow m_n \quad (m_k \in \mathbf{CLS_i^{\rightarrow}}, m_n \in \mathbf{CLS_j^{\rightarrow}})$$

We can show that message passing is performed in a consistent manner if the diagram in Figure 14 commutes, that is $m_n \circ \eta_{j_a} = \eta_{j_b} \circ m_k$.

The form of Figure 14 is the same as that for the natural transformation target square shown earlier in Figure 9 as the message passing function is a natural transformation between objects in the category of arrows [Simmonds 1990]. A simple way to realise that inter-arrow morphisms are natural transformations is to consider that the mapping between $\mathbf{CLS}$ and $\mathbf{CLS^{\rightarrow}}$ is a functor; hence a mapping between $\mathbf{CLS}$ – $\mathbf{CLS^{\rightarrow}}$ pairs is a natural transformation.

The constructions above provide a sound framework for investigating aspects of message passing such as control of types of initiators/ receivers and a formal basis for reflective systems.

3.5 Objects

Objects represent the extensional database holding values which must be consistent with the intension (the class structures).

There is a mapping V_i from each class $\mathbf{CLS_i}$ to the instances for each object–type $\mathbf{OBJ_i}$ which ensures that the constraints specified in the intension hold in the extension. The mapping is a functor as it is between categories. The functor V_i takes each arrow f in $\mathbf{CLS_i}$ to a set of arrow instances $V_i(f)$ in $\mathbf{OBJ_i}$, each domain $\mathrm{dom}(f)$ in $\mathbf{CLS_i}$ to a set of instances $V_i(\mathrm{dom}(f))$ in $\mathbf{OBJ_i}$, each codomain $\mathrm{cod}(f)$ in $\mathbf{CLS_i}$ to a set of instances $V_i(\mathrm{cod}(f))$ in $\mathbf{OBJ_i}$, the key K_0 to a set of instances $V_i(K_0)$, each non–key attribute $(K_i \mid 1 \leq i \leq r)$ to a set of instances $V_i(K_i)$ and each functional dependence $(d_i \mid 1 \leq i \leq r)$ to a set of arrow instances $V_i(d_i)$. All assignments by the functor V_i are of values for arrows, domains and codomains.

For each class $\mathbf{CLS_i}$, the functor V_i should preserve limits with respect to the functional dependencies, that is the diagram in Figure 15 should commute for every cone where $\prod A$ is the product of $(V_i(K_0) \times V_i(K_1) \ldots \times V_i(K_r))$, $(\pi_j \mid 0 \leq j \leq r)$ is a projection coordinate from $\prod A$ and $\{V_i(d_i) : V_i(K_0) \longrightarrow V_i(K_i) \mid 1 \leq i \leq r\}$ are the postulated functional dependencies. The commuting requirement is for all $V_i(K_i)$ where $(1 \leq i \leq r)$ it is true that $V_i(d_i) \circ \pi_0 = \pi_i$.

Referring back to our original discussion of limits, we are checking that the limit is preserved when real-world data is examined: that is, all cones in our family of cones commute and therefore an infimum can be constructed for the family of cones, in this case $\prod A$.

In object–oriented terms, objects contain values consistent with their class definitions (including typing) and perform operations according to the methods defined in their classes. The classes are the intension, the objects the extension. This can be represented generically by the diagram in Figure 16 where **CLS** represents a family of class categories, **OBJ** a family of object categories and **TYP** a family of type categories.

E, P and I are functors representing the mappings from object to class, from class to type and from object to type respectively. E (the dual of D) maps extension to intension. I is an inclusion functor so that **OBJ** is a subcategory of **TYP**. P indicates the typing constraints applied to classes and is a collection of arrows as indicated earlier in Categorical Concepts comprising:

- $\{v_i : \mathbf{1_{TYP_i}} \longrightarrow V_i\}$, representing the constraint that each instance v_i of an object $V_i (1 \leq i \leq q)$ is found in the category $\mathbf{TYP_i}$.
- $\{f_i : \mathbf{1_{TYP_i}} \longrightarrow F_i\}$, representing the constraint that each instance f_i of an arrow $F_i (1 \leq i \leq k)$ is found in the category $\mathbf{TYP_i}$.

In relational database terminology, each category **TYP** is a domain and each V is an attribute name. The database is consistent when the diagram commutes, that is $P \circ E = I$, representing the situation that our objects in the extension conform both to the class definition in the intension and to the typing constraints.

In a similar way, another functor R takes each pullback category **ASS** at the intension level to its extension **LNK**. This functor also preserves limits so that the constraints, such as for monic, epic and multiple relationships must apply in every case to the arrows between the actual data values. Diagram chasing ensures that type declarations are obeyed. Note how the model is not simply labelling constraints in the intension, it is enforcing them as *limit* or commuting requirements in the actual data values held in the extension.

3.6 Physical Storage Structures

In a similar way to the mapping between classes and objects, it is straight–forward to define mappings as functors between categories for objects and categories representing disk structures, say, hash tables or indexes. In earlier work [Rossiter & Heather 1992], we considered the various approaches to hashing in categorical terms.

3.7 Families of Categories

Shortly, we turn our attention to manipulation of our categories. For this purpose, it is convenient to introduce the concept of families of categories. In effect, we make the following groups:

- The category **INT** representing the intension as a family of c classes **CLS**, p association definitions **ASS** and g coproducts **UNI** representing inheritance.
- The category **EXT** representing the extension as a family of c objects **OBJ** and p association instances **LNK**.
- The functor D mapping from category **INT** to category **EXT**. This functor is called D (for database) because this is effectively the purpose of a database management system.

Between any two intension categories $\mathbf{INT_i}$ and $\mathbf{INT_j}$ (not necessarily distinct), m message passing routes can be defined using arrows of the form η described earlier between the corresponding arrow categories $\mathbf{INT_i^{\rightarrow}}$ and $\mathbf{INT_j^{\rightarrow}}$ respectively.

In future work, we intend to employ the concept of the categorical topos to represent the families described above.

3.8 Manipulation

A fundamental difficulty in current object–based systems is that of closure. It is not easy to obtain an output from a database that can be held as objects with associated class definitions such that the new structures rank equally *pari passu* with those in the existing database. Another difficulty with some object systems is that the output is a subset of variables in an object without any consideration of the arrows (functions) which are an equally important part of the data. This latter difficulty is readily handled in a formal manner by subcategories which provide a means of selecting some of the objects and arrows in a category and hence give in a natural manner the basis for a query mechanism. We remind ourselves that category $\mathbf{INT_j}$ is a subcategory of category $\mathbf{INT_i}$ if:

$$\mathrm{obj}_{\mathbf{INT_j}} \subseteq \mathrm{obj}_{\mathbf{INT_i}} \wedge \ \mathrm{Hom}_{\mathbf{INT_j}}(p, q) \subseteq \ \mathrm{Hom}_{\mathbf{INT_i}}(p, q) \quad (\forall p, q \in obj_{\mathbf{INT_j}})$$

Query operations can be defined at two levels: intra–object and inter–object. In categorical terms, in the general sense, there is no difference between the two as both are handled by arrows. The query language that we have developed is therefore based on arrows as in a functional data model such as DAPLEX [Shipman 1981], but our arrows are higher-order mappings from one category to another. Our arrows are in fact functors between the input structure and the output structure. The input for each operation is a category and the output is another category or a subcategory.

A functor arrow will return a category. It is therefore the norm that the output of a query on a category will be another category complete with arrows and objects which can be held in the database in the same way as other categories. The output or target category could contain structured values not present in the source category and assigned by another functor. It is therefore possible to create complex categories through manipulating values from a number of database categories. Alternatively, a forgetful functor applied to a category forgets some of the structure and this could be used, if the user desires, to forget the arrows and return simple tables of values as is the normal practice in network and some object–oriented databases.

An example of a query is given in the next section.

3.8.1 Query Example

We take the supplier–parts example given earlier, augmenting it with an inheritance structure where electrical parts are a specialization of parts in general. The following categories are defined:

- $\mathbf{INT_1}$ for the class $\mathbf{CLS_1}$ for suppliers: identifier K_0^1
  arrows:

$f_1 : K_0^1 \longrightarrow$ sname
$f_2 : K_0^1 \longrightarrow$ saddress
$f_3 : K_0^1 \longrightarrow$ no.shares
$f_4 : K_0^1 \longrightarrow$ share.price
$f_5 :$ (no.shares $\times$ share.price) $\longrightarrow$ capitalization

where sname, saddress, no.shares, share.price $\in A$; capitalization $\in U$; $f_1, \ldots, f_4 \in D$; $f_5 \in M$. A, U, F, M are defined in section on Classes. More detailed typing is not shown here.

- **INT$_2$** for the class **CLS$_2$** for parts: identifier K_0^2
  arrows:
  $f_6 : K_0^2 \longrightarrow$ pname
  $f_7 : K_0^2 \longrightarrow$ size
  $f_8 : K_0^2 \longrightarrow$ weight

  where pname, size, weight $\in A$; $f_6, \ldots, f_8 \in D$.

- **INT$_3$** for the pullback **ASS$_1$** of suppliers and parts over orders as in Figure 10: identifier $K_0^1 \times_O K_0^2$
  arrows:
  $\pi_l : K_0^1 \times_O K_0^2 \longrightarrow K_0^1$
  $\pi_r : K_0^1 \times_O K_0^2 \longrightarrow K_0^2$
  $f : K_0^1 \longrightarrow O$
  $g : K_0^2 \longrightarrow O$

  - K_0^1 is the identifier for the *supplier* class **CLS$_1$**.
  - K_0^2 is the identifier for the *parts* class **CLS$_2$**.
  - O is the powerset of *orders*.
  - Instances for O are of the form $\{< k_0^1, k_0^2, o >\mid f(k_0^1) = g(k_0^2), k_0^1 \in K_0^1, k_0^2 \in K_0^2, o \in \wp O\}$.

- **INT$_4$** for the class **CLS$_3$** for electrical parts – a specialization of parts with object identifier $\mathbf{1_{INT_4}}$ as the identity functor on **INT$_4$**
  arrows:
  $f_9 : \mathbf{1_{INT_4}} \longrightarrow$ voltage
  $f_{10} : \mathbf{1_{INT_4}} \longrightarrow$ capacity

  where voltage, capacity $\in A$; $f_9, f_{10} \in D$.

- **INT$_5$** for the union (coproduct) $\mathbf{UNI_1} = \mathbf{INT_2} + \mathbf{INT_4}$: identifier K_0^2
  arrows:
  $f_6, \ldots, f_8$ from **INT$_2$**
  f_9, f_{10} from **INT$_4$**
  $s_1 : K_0^2 \longrightarrow \mathbf{1_{INT_4}}$

The natural language query is "*What are the names and identifiers of suppliers with capitalization greater than one million pounds who supply an electrical part with voltage rating of 90 volts?*".

The series of functorial operations is given below. As is usual in database systems, these operations are defined in intensional terms but later, in order to introduce the closure concept, we look in more depth at what is actually involved in a query in terms of deriving an intension-extension mapping.

1. $X_1 : \mathbf{INT_6} \longrightarrow \mathbf{INT_5}$
   (Hom-set in $\mathbf{INT_6} = f_9, s_1$; subobjects in $\mathbf{INT_6} = (K_0^2, \mathbf{1_{INT_4}}, voltage \mid voltage = 90)$);

2. $X_2 : \mathbf{INT_7} \longrightarrow \mathbf{INT_3}$
   (Hom-set in $\mathbf{INT_7} = \pi_l$; subobjects in $\mathbf{INT_7} = (K_0^1 \times_O K_0^2, K_0^1 \mid K_0^2 \in \mathbf{INT_6})$);

3. $X_3 : \mathbf{INT_8} \longrightarrow \mathbf{INT_7}$
   (Hom-set in $\mathbf{INT_8} = \{\}$; subobject in $\mathbf{INT_8} = K_0^1$);

4. $X_4 : \mathbf{INT_9} \longrightarrow \mathbf{INT_1}$
   (Hom-set in $\mathbf{INT_9} = f_1, f_3, f_4, f_5$; subobjects in $\mathbf{INT_9} = (K_0^1$, sname, no.shares, share.price,capitalization | capitalization > 1000000));

5. $X_5 : \mathbf{INT_{10}} \longrightarrow \mathbf{INT_9}$
   (Hom-set in $\mathbf{INT_{10}} = f_1$; subobjects in $\mathbf{INT_{10}} = (K_0^1, \text{sname} \mid K_0^1 \in \text{obj}_{\mathbf{INT_8}})$);

The first functor X_1 derives the subcategory $\mathbf{INT_6}$ from $\mathbf{INT_5}$ by taking the composition of the arrows $s_1 : K_0^2 \longrightarrow \mathbf{1_{INT_4}}$ and $f_9 : \mathbf{1_{INT_4}} \longrightarrow voltage$ to determine which part identifiers K_0^2 are associated with a voltage of 90.

The second functor X_2 derives the subcategory $\mathbf{INT_7}$ from $\mathbf{INT_3}$ by restrictions on $\mathbf{INT_3}$ to the arrow π_l and on the source of π_l to cases where the part is in the subobject K_0^2 derived by X_1.

The third functor X_3 takes the output $\mathbf{INT_7}$ from X_2 and restricts it further to produce the subcategory $\mathbf{INT_8}$ with no arrows and subobject K_0^1. This subobject represents suppliers who supply parts rated at 90 volts.

The fourth functor X_4 produces subcategory $\mathbf{INT_9}$ from $\mathbf{INT_1}$ with the arrows f_1, f_3, f_4, f_5 and subobjects, including (K_0^1, sname), for which the application of f_3, f_4, f_5 to K_0^1 gives a capitalization of more than a million pounds.

The final functor X_5 produces the answer in a new subcategory $\mathbf{INT_{10}}$ which is a subcategory of $\mathbf{INT_9}$ with arrow f_1 and subobjects (K_0^1, sname) such that the values for K_0^1 are found in the category $\mathbf{INT_8}$, effectively giving an intersection between $\mathbf{INT_8}$ and $\mathbf{INT_9}$ over K_0^1.

Note that the strategy involves a selection of both arrows and objects rather than just objects as in the relational approach. The selection of arrows is achieved through defining hom–sets and the selection of objects through defining subobjects. Further, subobject specifications can involve predicates of arbitrary complexity to facilitate sophisticated searching techniques. All operations produce new subcategories. Results can also be injected into other categories so that new categories of arbitrary complexity can be constructed through free functors.

3.8.2 Closure in Queries

So far we have seen how intensional subcategories can be defined as results for searches. But can we store the results obtained in our example queries back in the database in their current form to be used in exactly the same way as existing classes?

The answer is that we have defined a series of subcategories $\mathbf{INT_6} \ldots \mathbf{INT_{10}}$ in intensional terms but have omitted to define the corresponding extensional subcategories. The relationship between each intension $\mathbf{INT_i}$ and extension $\mathbf{EXT_i}$ is given by the mapping $D_i : \mathbf{INT_i} \longrightarrow \mathbf{EXT_i}$. Therefore for a query earlier, say no.4, we can write in more detail:

$$D_1 : \mathbf{INT_1} \longrightarrow \mathbf{EXT_1}$$

$$D_9 : \mathbf{INT_9} \longrightarrow \mathbf{EXT_9}$$

as functors for the query representing intension and extension mapping respectively. Each query therefore involves a mapping between an intension–extension pair as source and an intension-extension pair as target. We can represent this structure as shown in Figure 17 with the query now represented by the natural transformation σ_4.

To be a natural transformation, the square introduced earlier as Figure 9 and shown as Figure 18 for our current query σ_4 should commute for every arrow $f_j : \text{dom}(f_j) \longrightarrow \text{cod}(f_j)$ in the source category $\mathbf{INT_i}$ $(1 \leq j \leq k, 1 \leq i \leq (c+p+g))$.

This means that for all f_j in $\mathbf{INT_i}$ then $\sigma_{4_b} \circ D_1(f_j) = D_9(f_j) \circ \sigma_{4_a}$ that is our two paths from the values for domains of arrows in the source category $D_1(\text{dom}(f_j))$ to the values for codomains of arrows in the target category $D_9(\text{cod}(f_j))$ should be equal. One path A involving σ_{4_a} navigates from domain values in the source category via domain values in the target category to codomain values in the target category; the other B involving σ_{4_b} has the same starting and finishing points but navigates via codomain values in the source category.

In path A, the arrow σ_{4_a} creates a subobject of the domains for arrows f_j in $\mathbf{EXT_1}$ to be assigned to the extension category $\mathbf{EXT_9}$. In path B, the arrow σ_{4_b} creates a subobject of the codomains for arrows f_j in $\mathbf{EXT_1}$ to be assigned to the extension category $\mathbf{EXT_9}$. Referring back to the syntax used in our query examples, the hom-set of the target category is defined as the set of f_j assigned by D_9 and the subobjects in the target category are defined as the union of $\text{dom}(f_j)$ and $\text{cod}(f_j)$ for arrows f_j assigned by D_9.

The output from σ_4 is clearly a structure which can be held in our database, ranking equally with other classes and objects in the system. Typing constraints will continue to be enforced in the output structure. So the typing for objects and arrows in $\mathbf{INT_9}$ will be based on that in $\mathbf{INT_1}$ with the additional constraint that capitalizations must be greater than one million pounds. In computing terms, we are expressing the constraint that no object can exist in our database which is not fully described by a class definition.

In categorical terms, we are expressing a query as a natural transformation. Each functor can be considered as a continuous function (infimum preserving) between two posets with limits : each structure $D_i : \mathbf{INT_i} \longrightarrow \mathbf{EXT_i}$ is then viewed as a closed cartesian category where D_i is a continuous function preserving the infimum (as key) within the poset $\mathbf{INT_i}$ in $\mathbf{EXT_i}$. Closed cartesian categories have been used in other areas of computing science, in formalisms such as Scott domains, as they are equivalent in theoretical power to the typed lambda calculus [Barr & Wells 1990].

3.9 Views on Classes

The mechanism required for views is similar to that for queries. In fact a snapshot view will be identical to a query. However, there are two other aspects of views that need further consideration:

- The need to retain the definition within the database and produce views of the current data on demand by the user.

- The problems of updating the database by users who have limited views of the data structures.

The first involves creating a mapping in intensional terms only as we did with the queries originally defined as $X_1 \ldots X_5$. Thus the functors in the family X defined earlier can all be construed as defined views. When a view is realised, the corresponding natural transformation is activated to deduce the extension.

The second involves the definition of another functor, say τ, to relate the result from the query back to the main database values. Thus if we define a view as shown in Figure 19, we can achieve updatable views on a class.

A well–known special case of a view is that taken of the complete database. In this case for every $D_i : \mathbf{INT_i} \longrightarrow \mathbf{EXT_i}$ in the database, the application of σ_i returns an identical $D_i : \mathbf{INT_i} \longrightarrow \mathbf{EXT_i}$ in the view. The application of τ_i to each $D_i : \mathbf{INT_i} \longrightarrow \mathbf{EXT_i}$ in the view should then faithfully return our initial database. If this is so, there is a natural isomorphism between σ and τ and our database is consistent.

4 Conclusions

The conclusions can be stated briefly. Mainstream mathematics with the development of category theory has now attained the same level of formal abstraction as needed for databases. Category theory therefore provides a formal modelling technique that is universal in the sense of mathematics.

Category theory, for example, fills in the gaps in current object models where there is weakness in comparison to relational models in respect of formality, views, query closure, etc. We would claim that category theory actually provides a formal basis for the object–relational model, underpinning work on systems such as Postgres [Stonebraker & Rowe 1986] and on the forthcoming SQL–3 standard.

More specifically, we have provided evidence of the following:

- multi–level theory gives natural handling of intension, extension and views;
- imprecise descriptions of association, inheritance and aggregation can now be rationalized and made formal;
- message passing can be represented by natural transformations between methods;
- queries *with closure* are natural transformations between intension–extension functors;

- views with updating are pairs of dual natural transformations between intension–extension functors;

Orthogonality and consistency are achieved throughout by use of the single concept of an arrow. We have kept carefully within the known theory rigorously established over the last 50 years by a number of pure mathematicians of world class. We have resisted the temptation to customize the main stream mathematics or make up our own definitions on the basis that any concept should be understood fully and tested in pure theory before it becomes applicable in applied mathematics (see comments by Hoare in [de Moor 1992]).

No doubt alternative modelling techniques could be developed to provide the same power and multi–level capability available in category theory. But everything would need to be proved from scratch. Because of the constructive nature of category theory, our diagrams are themselves formal proofs. The results obtained therefore by treating a database as a functor show the advantages available to the database community from category theory.

5 References

Atkinson et al 1990, M.Atkinson et al, The Object–oriented Database System Manifesto, in a number of publication including: The Story of O_2: Implementing an Object–oriented Database System, Morgan Kaufmann 1992.

Barr & Wells 1990, M.Barr & C.Wells, Category Theory for Computing Science, Prentice–Hall.

Beeri 1992, C.Beeri, New Data Models and Languages – the Challenge, Proceedings 11th ACM Symposium on Principles of Database Systems 1–15.

Cardelli 1984, L.Cardelli, A Semantics of Multiple Inheritance, in: Semantics of Data Types, Lecture Notes in Computing Science **173** 51–67, Springer Verlag.

Cartmell 1985, J.Cartmell, Formalising the Network and Hierarchical Data Models – an Application of Categorical Logic, Lecture Notes in Computer Science **240** 466–492.

Demetrovics, Libkin & Muchnik 1992, J.Demetrovics, L.Libkin & I.B.Muchnik, Functional Dependencies in Relational Databases: A Lattice Point of View, Discrete Applied Mathematics **40**(2) 155–185.

Dennis–Jones & Rhydeheard 1993, E.Dennis–Jones & D.E.Rhydeheard, Categorical ML – Category–Theoretic Modular Programming, Formal Aspects in Computing **5**(4) 337–366.

Elmasri & Navathe 1994, R.Elmasri & S.B.Navathe, Fundamentals of Database Systems, Benjamin/Cummings, Redwood City, 2nd edition.

Freyd 1964, P.Freyd, Abelian Categories: An Introduction to the Theory of Functors, Harper and Row, New York.

Freyd & Scedrov 1990, P.J.Freyd & A.Scedrov, Categories, Allegories, North–Holland Mathematical Library **39**.

Gray, Kulkarni & Paton 1992, P.M.D.Gray, K.G.Kulkarni & N.W.Paton, Object-Oriented Databases: A Semantic Data Model Approach, Prentice Hall.

Heather & Rossiter 1994a, M.A.Heather & B.N.Rossiter, Applying Geometric Logic to Law, Proceedings 4th National Conference on Law, Computers and Artificial Intelligence, Exeter 80–95.

Heather & Rossiter 1994b, M.A.Heather & B.N.Rossiter, Category Theory: the Mathematics for the Humanities?, in: International ALLC/ACH Conference, *CONSENSUS EX MACHINA*, Paris (Sorbonne).

Kim 1990, W.Kim, Introduction to Object–oriented Database Systems, MIT Press.

Kim 1994, W.Kim, On Object-Oriented Database Technology, ADB. Inc.

Kuper & Vardi 1993, K.M.Kuper & M.Y.Vardi, The Logical Data Model, ACM TODS **18**(3) 379–413.

Lellahi & Spyratos 1991, S.K.Lellahi & N.Spyratos, Towards a Categorical Model supporting Structured Objects and Inheritance, FIDE Technical Report, University of Glasgow, FIDE/91/8.

Lellahi & Spyratos 1992, S.K.Lellahi & N.Spyratos, Categorical Modelling of Database Concepts, FIDE Technical Report, University of Glasgow, FIDE/92/38.

Mac Lane 1971, Saunders Mac Lane, Categories for the Working Mathematician, Springer–Verlag 1971.

Mac Lane & Moerdijk 1991, Saunders Mac Lane & Ieke Moerdijk, Sheaves in Geometry and Logic, A First Introduction to Topos Theory, Springer–Verlag 1991.

Manes & Arbib 1986, E.Manes & M.Arbib, Algebraic Approaches to Program Semantics, Springer Verlag 1986.

de Moor 1992, O. De Moor, Categories, Relations and Dynamic Programming, Oxford University Computing Laboratory Report PRG–98.

Nelson, Rossiter & Heather 1994, D.A.Nelson, B.N.Rossiter & M.A.Heather, The Functorial Data Model – An extension to Functional Databases, Technical Report no.488, Computing Science, Newcastle University.

Rossiter & Heather 1992, B.N.Rossiter & M.A.Heather, Applying Category Theory to Databases, presented to 8th British Colloquium for Theoretical Computing Science in March 1992, published as Technical Report no.407, Computing Science, Newcastle University.

Rossiter & Heather 1993, B.N.Rossiter & M.A.Heather, Database Architecture and Functional Dependencies expressed with Formal Categories and Functors, published as Technical Report no.432, Computing Science, Newcastle University.

Shipman 1981, D.W.Shipman, The Functional Data Model and the Data Language DAPLEX, ACM TODS **6** 140–173.

Sibley & Kerschberg 1977, E.H.Sibley & L.Kerschberg, Data Architecture and Data Model Considerations, AFIPS Conference Proceedings, Dallas 1977, 85–96.

Simmonds 1990, H.Simmonds, Lecture Notes for SERC School on Logic for Information Technology, University of Leeds.

Smith & Smith 1977, J.Smith & D.Smith, Data Abstraction, Aggregation and Generalization, ACM TODS **2**(2) 105–133.

Stonebraker & Rowe 1986, M.Stonebraker & L.A.Rowe, The Design of Postgres, Proceedings ACM SIGMOD Conference, 340–355.

Tsichritzis 1978, D.Tsichritzis, ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Data Base Management Systems, Information Systems **3**.

Ullman 1988, J.D.Ullman, Principles of Database and Knowledge-base Systems, Computer Science Press **1**.

6 Appendix I: Symbols employed for representing database concepts

$$
\begin{tikzcd}
& A + C & \\
A \arrow[ur, "i_l"] & & C \arrow[ul, "i_r"']
\end{tikzcd}
$$

Figure 4: Coproduct Cone for Objects A and C

$$
\begin{tikzcd}
& A \arrow[dr, "f"] & \\
A \times_B C \arrow[ur, "\pi_l"] \arrow[dr, "\pi_r"'] & & B \\
& C \arrow[ur, "g"'] &
\end{tikzcd}
$$

Figure 5: Diagram of Pullback of A and C over B

$$
\begin{tikzcd}
& A \arrow[dr, "f"] & \\
B \arrow[ur, "i_l"] \arrow[dr, "i_r"'] & & S \\
& C \arrow[ur, "g"'] &
\end{tikzcd}
$$

Figure 6: Diagram of Pushout of S from A and C

$$
\begin{tikzcd}
& P \arrow[dl, "\pi_0"'] \arrow[dr, "\pi_i"] & \\
K \arrow[rr, "f_i"'] & & N_i
\end{tikzcd}
$$

Figure 7: A Canonical Example of a Cone D_i

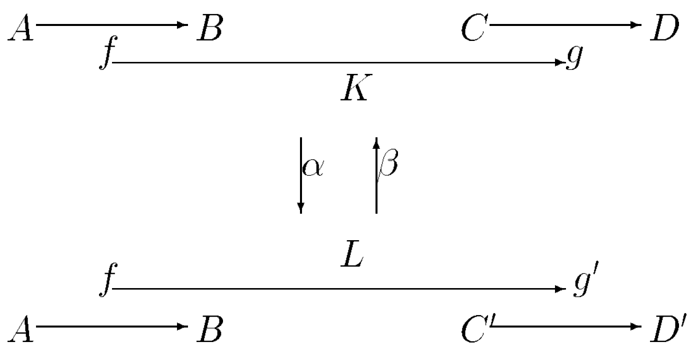

Figure 8: Natural Transformations compare Functors

$$\begin{tikzcd}
K(A) \arrow[r, "\alpha_A"] \arrow[d, "K(f)"'] & L(A) \arrow[d, "L(f)"] \\
K(B) \arrow[r, "\alpha_B"'] & L(B)
\end{tikzcd}$$

Figure 9: Commuting Target Square with Covariant Natural Transformation

$$\begin{tikzcd}
& K_0^1 \arrow[dr, "f"] & \\
K_0^1 \times_O K_0^2 \arrow[ur, "\pi_l"] \arrow[dr, "\pi_r"'] & & O \\
& K_0^2 \arrow[ur, "g"'] &
\end{tikzcd}$$

Figure 10: Diagram of Pullback of K_0^1 and K_0^2 over O

$$
\begin{tikzcd}
 & K_0^1 = \{1,2\} \arrow[dr, "f"] & \\
K_0^1 \times_O K_0^2 = \{<1,9>,<2,9>\} \arrow[ur, "\pi_l"] \arrow[dr, "\pi_r"'] & & O \\
 & K_0^2 = \{9,10\} \arrow[ur, "g"'] &
\end{tikzcd}
$$

π_l is monic and epic
π_r is not monic and not epic

Figure 11: Diagram of Pullback of K_0^1 and K_0^2 over O

$$
\begin{tikzcd}
 & K_0^1 = \{1,2\} \arrow[dr, "f"] & \\
K_0^1 \times_O K_0^2 = \{<1,9>,<1,10>\} \arrow[ur, "\pi_l"] \arrow[dr, "\pi_r"'] & & O \\
 & K_0^2 = \{9,10,11\} \arrow[ur, "g"'] &
\end{tikzcd}
$$

π_l is not monic and not epic
π_r is monic and not epic

Figure 12: Diagram of Pullback of K_0^1 and K_0^2 over O

$$
\begin{tikzcd}
 & \mathbf{CLS_3} + \mathbf{CLS_4} & \\
\mathbf{CLS_3} \arrow[ur, "i_l"] \arrow[rr, "s"] & & \mathbf{CLS_4} \arrow[ul, "i_r"']
\end{tikzcd}
$$

Figure 13: Coproduct Cone for Objects $\mathbf{CLS_3}$ and $\mathbf{CLS_4}$

$$
\begin{tikzcd}
\mathrm{dom}(m_k) \arrow[r, "\eta_{j_a}"] \arrow[d, "m_k"'] & \mathrm{dom}(m_n) \arrow[d, "m_n"] \\
\mathrm{cod}(m_k) \arrow[r, "\eta_{j_b}"'] & \mathrm{cod}(m_n)
\end{tikzcd}
$$

Figure 14: Commuting Square for Message η_j between m_k and m_n in Arrow Categories $\mathbf{CLS_i^{\rightarrow}}$ and $\mathbf{CLS_j^{\rightarrow}}$ respectively

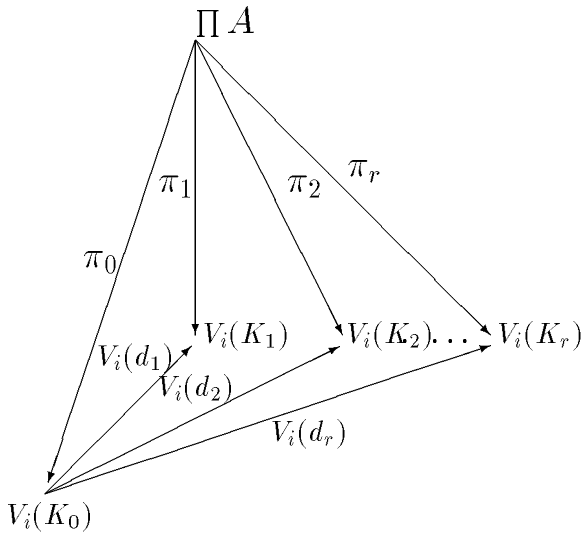

Figure 15: Cone for extension $\prod A$ in the Category **OBJ**

$$\begin{tikzcd}
 & \mathbf{TYP} & \\
\mathbf{CLS} \arrow[ur, "P"] & & \mathbf{OBJ} \arrow[ul, "I"'] \arrow[ll, "E"]
\end{tikzcd}$$

Figure 16: Commuting Diagram for Consistency of Objects with Classes and Types

$$\begin{tikzcd}
\mathbf{INT_1} \arrow[rr, "D_1"'] \arrow[drr, phantom, "\downarrow \sigma_4"] & & \mathbf{EXT_1} \\
\mathbf{INT_9} \arrow[rr, "D_9"] & & \mathbf{EXT_9}
\end{tikzcd}$$

Figure 17: The Query σ_4 as a Natural Transformation with source D_1 and target D_9

$$
\begin{tikzcd}
D_1(\mathrm{dom}(f_j)) \arrow[r, "\sigma_{4_a}"] \arrow[d, "D_1(f_j)"'] & D_9(\mathrm{dom}(f_j)) \arrow[d, "D_9(f_j)"] \\
D_1(\mathrm{cod}(f_j)) \arrow[r, "\sigma_{4_b}"'] & D_9(\mathrm{cod}(f_j))
\end{tikzcd}
$$

Figure 18: The query σ_4 as a Commuting Target Square with Covariant Natural Transformation σ_4 from functor D_1 to functor D_9

$$
\begin{tikzcd}
\mathbf{INT_1} \arrow[rr, "D_1"'] & {} \arrow[dd, shift right=2, "\sigma_4"] & \mathbf{EXT_1} \\
 & & \\
\mathbf{INT_9} \arrow[rr, "D_9"] & {} \arrow[uu, shift right=2, "\tau_4"'] & \mathbf{EXT_9}
\end{tikzcd}
$$

Figure 19: The View σ_4 as a Natural Transformation with Updates through τ_4

Level	symbol	instance	range i	concept
Category	**ASS**	$\mathbf{ASS_i}$	$1 \ldots p$	association intension
	CLS	$\mathbf{CLS_i}$	$1 \ldots c$	class
	$\mathbf{CLS}^{\rightarrow}$	$\mathbf{CLS_i^{\rightarrow}}$	$1 \ldots c$	class with arrows considered as arrow–objects
	DEP	$\mathbf{DEP_i}$	$1 \ldots c$	dependencies (in poset)
	EXT	$\mathbf{EXT_i}$	$1 \ldots c + p$	database extension
	INT	$\mathbf{INT_i}$	$1 \ldots c + p + g$	database intension
	$\mathbf{INT}^{\rightarrow}$	$\mathbf{INT_i^{\rightarrow}}$	$1 \ldots c + p + g$	intension with arrows considered as arrow–objects
	LNK	$\mathbf{LNK_i}$	$1 \ldots p$	association extension
	OBJ	$\mathbf{OBJ_i}$	$1 \ldots c$	database object
	PRJ	$\mathbf{PRJ_i}$	$1 \ldots c$	persistent variables (in powerset ordered by projection)
	PSU	$\mathbf{PSU_i}$	$1 \ldots c$	pseudotransitivities (in poset)
	TYP	$\mathbf{TYP_i}$	≥ 1	types
	UNI	$\mathbf{UNI_i}$	$1 \ldots g$	coproduct (inheritance)
Arrow	D	d_i	$0 \ldots r'$	dependencies
	F	f_i	$0 \ldots k$	all arrows within a class
	M	m_i	$0 \ldots s$	methods
	P	p_i	$0 \ldots r''$	pseudotransitivity
	S	s_i	$0 \ldots g$	supertype–subtype
Object	A	a_i	$1 \ldots n$	persistent variables
	E	e_i	$0 \ldots r'$	persistent variables in arrows D
	E'	e'_i	$0 \ldots r''$	persistent variables in arrows P
	K_0^i	k_0^i	$1 \ldots c$	initial object (key) in $\mathbf{CLS_i}$
	$K_j^i (1 \leq j \leq r)$	k_j^i	$1 \ldots c$	non–key attributes in $\mathbf{CLS_i}$
	U	u_i	$0 \ldots n'$	memory variables
	V	v_i	$1 \ldots q$	all variables
Functor	D	D_i	$1 \ldots c + p$	map intension to extension
	E	E_i	$1 \ldots c$	map object to class
	G	G_i	$1 \ldots c$	map variables A to **PRJ**
	G'	G'_i	$1 \ldots c$	map variables E to **DEP**
	G''	G''_i	$1 \ldots c$	map variables E' to **PSU**
	I	I_i	$1 \ldots c$	map object to type
	P	P_i	$1 \ldots c$	map class to type
	R	R_i	$1 \ldots p$	map association intension to extension
	V	V_i	$1 \ldots c$	map class to object
	X	X_i	≥ 0	query mapping intension to intension
Natural Transform-ation	σ	σ_i	≥ 0	query/view deriving one **INT** : **ENT** pair as a 'subset' of another
	τ	τ_i	≥ 0	dual of query/view σ
	η	η_i	$0 \ldots m$	message from arrow–object in $\mathbf{INT_i^{\rightarrow}}$ to arrow–object in $\mathbf{INT_j^{\rightarrow}}$