Prototyping a Categorical Database in P/FDM

D. A. Nelson and B. N. Rossiter
Dept. of Computing Science, University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK
e-mail: (d.a.nelson, b.n.rossiter)@newcastle.ac.uk

Abstract

The relational data model uses set theory to provide a formal background, thus ensuring a rigorous mathematical data model with support for manipulation. The newer generation database models are based on the object–oriented programming paradigm, and so fall short of having a formal background, especially in some of the more complex data manipulation areas. We use category theory to provide a formalism for object databases, known as the product model. This paper will describe our formal model for the key aspects of object databases. In particular, we will examine how this model deals with three of the most important problems inherent in object databases, those of queries, closure and views. As well as this, we investigate the more common database concepts, such as keys, relationships, aggregation, etc. We will implement a prototype of this model using P/FDM, a semantic data model database system based on the functional model of Shipman, with object–oriented extensions.

1 Introduction

Relational data models are supported by a strong theoretical formalism based on set theory, which ensures a rigorous mathematical data model as well as support for manipulation, with both the relational algebra and calculus being strongly defined. Newer generation data models are based on object–oriented systems, which so far are strongly lacking in any kind of formal definition, especially for data manipulation concerns.

This paper is concerned with a formal model for object databases[1]. Category theory [3] is used to define the product model, a formal notation for representing features of an object based database. In particular, we will examine how this model deals with three of the most important problems inherent in object databases, those of queries, closure and views, as well as how our model deals with more common database concepts, such as keys, relationships, aggregation, etc.

A prototype of this model is currently being produced, using P/FDM [10, 12], a database system based on the functional data model database of Shipman [21], but which has incorporated some object–oriented extensions. We will discuss our reasons for using P/FDM, and show some of the problems that occur in developing a categorical database. Our implementation will look at

[1] not necessarily object–oriented, but one which contains most of the concepts from the object–oriented paradigm

both the standard abstractions of data models, and the more important details of object databases as mentioned above.

The aims of our work on this theoretical database model are to demonstrate:

- that category theory provides a feasible formal model for object–relational databases;
- that a practical categorical database can be implemented, and that it can suitably model real world data storage problems;
- that the implementation problems of closure, queries and views inherent in most of the current object–based databases can be resolved through a categorical formalism.

The object–relational model (e.g. Postgres) [25, 26] is similar to our formalism for object-databases, while our relationships are similar in functionality and appearance to those in the entity-relationship model [6]. We also use Boyce–Codd Normal Form (BCNF) [28] as a normalisation constraint when determining the keys in a particular database object, ensuring a high level of consistency in the database.

One important question must be 'why category theory?' Although any theory could be used for modelling object databases, the multi–level architecture of category theory, compared to the flatness of most other theories such as set theory, makes the model less complex when we need different levels for schema, queries, etc. in the database. Category theory is also based on the arrow as its primitive concept, which gives natural modelling of dynamic as well as static aspects, where the arrow can provide either a relationship between two properties, or can act as a function mapping from one property to another. As well as this, the diagrammatical tools of category theory, i.e. diagram chasing giving algebraic equations, and the consistency tests, are useful additions to any model of a database in achieving a database system with a high level of consistency and correctness.

The categorical data modelling manifesto by Cadish and Diskin [4], suggests that category theory has an unexpectedly high relevance for semantic modelling, database design and database theory. Their manifesto supports the reasons we have outlined for using category theory for formalising databases, in particular they believe that using the arrow for defining internal structure of objects, as we do, is just the specification methodology the database area needs for universal models.

1.1 Object Database Abstractions

Because many of the current object–oriented databases are based heavily on C++ (or some other object–oriented programming language), they are usually little more than just persistent object–stores. This means that views and closure are difficult to implement because they do not migrate easily into object–oriented programming languages, due to the fact that run–time schema changes are required, and new objects require creating on the fly.

The matter of a query language is the most interesting prospect. Some of the newer object database systems are being released with languages based on SQL, and there is a new SQL3 standard [1] being written, which incorporates

features for handling complex objects. Many of the current systems usually provide no more than facilities for querying through C++ methods though, i.e. the application developer must write most queries as C++ methods rather than using some complete query language.

Our query language will be heavily influenced by Shipman's DAPLEX [21], while supporting the whole of the functionality of an SQL based query language. DAPLEX is a data definition and manipulation language based on the functional data model, with a query language based entirely on functions and function composition.

1.2 Overview of Paper

The rest of this paper will outline the categorical concepts used for the product model, in particular highlighting how we aim to achieve queries, closure and views. Then finally, we will discuss our use of P/FDM for developing a prototype of the product model, highlighting the major implementation problems that we have encountered, and discussing other implementations of categorical data types that exist already.

2 The Product Model

Using standard textbook categorical constructions, 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 and functionality and membership class in object (E–R) models.

5. The standard information system abstractions formulated in the 1970s [24] and which are prime targets of current object–oriented databases [2] and object–relational systems [25, 26]. These abstractions include inheritance (generalisation and specialisation); 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 [27] with definitions of views, global schemata and the internal structures and the mapping between them.

All arrow constructions that we employ, as regards composition and association, must conform to the four axioms of category theory [3].

2.1 Classes

2.1.1 Basic Structures

The class construction is an essential starting point for representing the intension of a database. The collection of classes is represented by an effective topos **CLASS** constructed by the Grothendieck method as the category $\mathbf{G}(\mathbf{CLS}, METC)$ where $METC : \mathbf{CLS} \longrightarrow \mathbf{CLASS}$ is a functor embedding each class definition **CLS** in a metaobject **CLASS**.

Each category **CLS** is a collection of arrows F given by the Hom–set $\mathrm{Hom}_{\mathbf{CLS}}(v, v')$ for all $v, v' \in V$ where V is the collection of objects in the category **CLS** given by $\mathrm{obj}_{\mathbf{CLS}}$. Individual arrows may be denoted by f.

Arrows are typed as either actions (transformations) or dependencies by specifying the category (i.e. some pool of values) from which the item is taken. The actions are typed by the category of methods **M**:

$$m : \mathbf{1}_{\mathbf{M}} \longrightarrow M$$

where m is a method arrow in our collection of methods M in the category **CLS** found in the universe of methods **M**. The dependencies are typed by the category of dependencies **D**:

$$d : \mathbf{1}_{\mathbf{D}} \longrightarrow D$$

where d is a dependency arrow in our collection of dependencies D in the category **CLS** found in the universe of dependencies **D**.

In general, typing is indicated by a collection of mappings $\{h : \mathbf{1}_{\mathbf{TYP}} \longrightarrow H\}$ 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.

Each arrow has a domain and a codomain. Our domains and codomains may be either elemental or composite. In the elemental case, the source or target of the arrow is a single variable v, a member of the object V representing all the elementary variables for the class **CLS**. In the composite case, the source or target of the arrow contains two or more variables x, a member of the powerobject of $V (x \in \wp V)$ for the class **CLS**.

```
of the form                                                    ???
with subobject classifier
$c$ and characteristic function $\chi:                         ???
```

In more detail, each arrow f in the category **CLS** has domain $\mathrm{dom}(f)$ and codomain $\mathrm{cod}(f)$. The domain and codomain names are not necessarily distinct. The union of all $\mathrm{dom}(f)$ and $\mathrm{cod}(f)$ in a class gives the collection of variables in the class which was specified earlier as $\mathrm{obj}_{\mathbf{CLS}}$ or more conveniently as the object V. 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 [11] but can cope well with open covers [14]. Variables may be either *persistent variables* given by the subobject A comprising the persistent components a of the class, or *memory variables* given by the subobject U comprising the transient components u of the class. A and U are both subobjects of the object V.

Later, we describe the identification of one or more domains as candidate keys and the selection of one of these as the primary key.

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, $d : x \longrightarrow y$, $x \in \wp A, y \in A$, that is, x is a member of the powerobject 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 the union of $\mathrm{dom}(d)$ and $\mathrm{cod}(d)$.

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, giving rise to a collection of pseudotransitivity arrows [28] $P = \{p_i : x \longrightarrow y\}$ $(x \in \wp A, y \in A, 0 \leq i \leq r'')$. 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 object in the powerobject 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 [21] or virtual variables which can be manipulated by database operations.

2.2 Normalization

We need to define an identifier to enable individual records to be picked out from a collection of records and we also need to determine whether our class structures suffer from storage anomalies. In relational databases, the concept of normalization is used to provide such constraints within the context of a user–defined key often providing a degree of content addressability. Normalization is usually not an automatic task and its benefits as reagrds robustness in update operations are obatined at some cost in complexity. In object–based systems, the procedures are much simpler as the identifier is assigned by the system but there is no methodical attempt to avoid storage anomalies. We consider

the simple object–based system first, including the notation for an identifier, followed by the more challenging relational concepts.

In object–based systems, the key is a system–assigned object identifier defined as the identity functor on a category, for example, $\mathbf{1_{CLS}} : \mathbf{CLS} \longrightarrow \mathbf{CLS}$. No further checks need be made for dependencies. All our 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. So above, $\mathbf{1_{CLS}}$ is the same as K_0.

In a relational system, the key K_0 is derived as shown below for each class category **CLS** [19] following a lattice approach [7] rather than an algorithmic one [28]. The lattice formalism lends itself more to a categorical approach with its emphasis on partial–order constructions. We employ the identifiers and dependencies to test whether our class structures correspond to BCNF. 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 basically automates the production of normalized classes, taking as input the category **CLS** augmented with trivial dependency arrows, and producing as output collections of normalized classes **NOR** with identifiers meeting our rules. In more detail, we first generate two categories **PRJ** and **PSU** containing respectively trivial projection arrows and non–trivial pseudotransititivity arrows (dependencies inferred from the postulated functional dependencies and their combinations [28]). In the third stage, these two new categories are injected into **CLS** to give the equivalent of F^* in standard relational database theory.

1. Generate the partial–order 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 $L : A \longrightarrow \mathbf{PRJ}$.

2. Generate the partial–order category **PSU** with elements $p, q \in E'$ and arrows $\{p\}$ that is to apply the free functor $L'' : E' \longrightarrow \mathbf{PSU}$.

3. Take **PRJ** and **PSU**. Inject them into **CLS**, that is add the arrows of **PRJ** and **PSU** to those already in **CLS**.

Now take the augmented preliminary formulation for each **CLS**. Consider the adjointness:

$$F \dashv U : \mathbf{CLS} \leftrightarrow \mathbf{NOR}$$

where F is a free functor taking **CLS**, which may be a preorder, to a collection of normalized categories **NOR**. F selects a collection **NOR** which meets the normalization rule of BCNF (or whatever level to which we are working). U is the underlying functor which selects those collections of **NOR** that can be naturally joined together to return **CLS**, thus ensuring that **NOR** is a lossless decomposition of **CLS**.

F is left adjoint to U and U is right adjoint to F. By virtue of the adjoint functor theorem [Freyd & Scedrov 1990], left adjoints preserve colimits and right adjoints preserve limits. We can therefore say, that if adjointness occurs

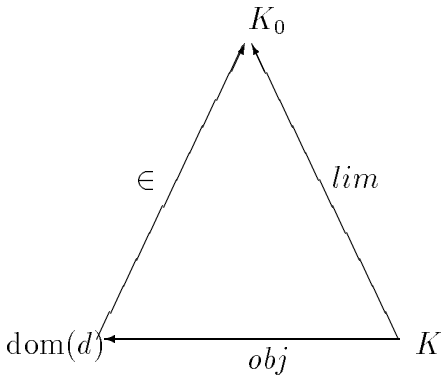


Figure 1: Commuting Diagram for Test for BCNF

between **CLS** and **NOR**, then U preserves limits and gives us readily one of our requirements: that any decomposition of **CLS** into **NOR** shall be lossless with the ability to recover **CLS** by a join operation on the various **NOR**.

F needs to do more. In our first series of operations, we produce an identifier for each normalized class. We next determine whether the class meets our normalization rules through a commuting test. First, the identifier K_0 is determined:

1. The limit of the objects of A in **NOR** ($\bigwedge A$) is the primary key PK. If there is more than one limit, the set of maximal lower bounds is the set of candidate keys CK.

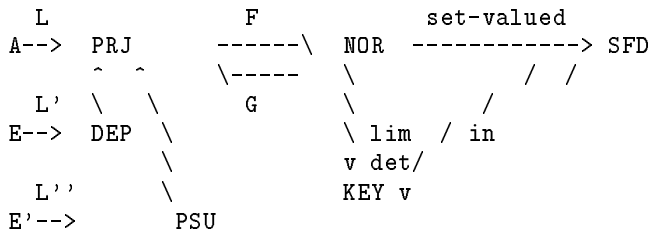
2. Each class **NOR** is in BCNF if each source of a functional dependency arrow is PK or is a member of CK.

3. 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 $\mathbf{CLS_i}$.

4. 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.

Then each **NOR** is tested for BCNF by performing the following test shown in the diagram 1. Every menber of a collection of **NOR** must commute according to the equation ??? for it to be a valid selection by the free functor G.

```
K ---------->dom(d)
i          /
i         /
V        V
```



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 $k \longleftarrow k+r$.

?? 3NF may be more fun - need to use adjointness to give free and underlying ?? functors

2.3 Relationships

The association abstraction between classes is represented in object models by notation based on the Entity–Relationship [6] (E–R) approach. In categorical terms, the E–R model is represented by pullbacks.

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 2, 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 $\{< k_0^1, k_0^2, o > \mid f(k_0^1) = g(k_0^2), k_0^1 \in K_0^1, k_0^2 \in K_0^2, o \in \wp O\}$ 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

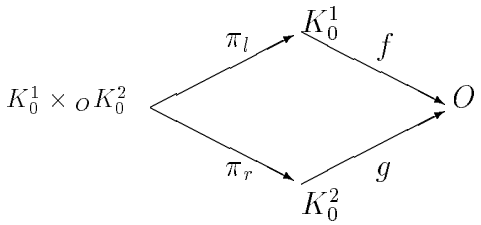


Figure 2: Diagram of Pullback of K_0^1 and K_0^2 over O

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.

Analogous reasoning can be applied to the arrow π_r.

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.

2.3.1 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 [18]. 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.

2.3.2 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 1 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.

2.3.3 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 3 which yields a disjoint union of two or more objects. Consider:

- a category $\mathbf{CLS_3}$ (employers) with the 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

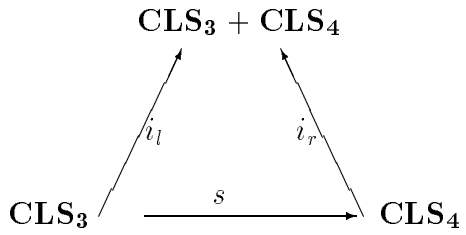


Figure 3: Coproduct Cone for Objects $\mathbf{CLS_3}$ and $\mathbf{CLS_4}$

- a category $\mathbf{CLS_4}$ (managers) with the set of arrows $\mathrm{Hom}_{\mathbf{CLS_4}}p, q$ and the 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 specialisation of $\mathbf{CLS_3}$ over $\mathbf{CLS_4}$. The arrow s (meaning subclass) shows the direction of the specialisation: $s : \mathbf{CLS_3} \longrightarrow \mathbf{CLS_4}$ (employee has subclass manager). In general, the superclass category will be identified by one or more properties in the data and the subclass 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 superclass category $\mathbf{CLS_3}$ and the identity functor $\mathbf{1}_{\mathbf{CLS_4}}$ of the subclass category:

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

Since a coproduct can, in turn, be the base of another cone, it is a simple matter to construct inheritance hierarchies [15]. 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, although we are currently investigating the use of pushouts [3] for multiple inheritance. 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 [5].

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 $(\mathrm{Hom}_{\mathbf{CLS_3}}p, q + \mathrm{Hom}_{\mathbf{CLS_4}}p, q)$.

2.3.4 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 [13] 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.

2.4 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.

2.5 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 4 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 $V_i(d_i) \circ \pi_0 = \pi_i$.

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 5 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 V) 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 comprising:

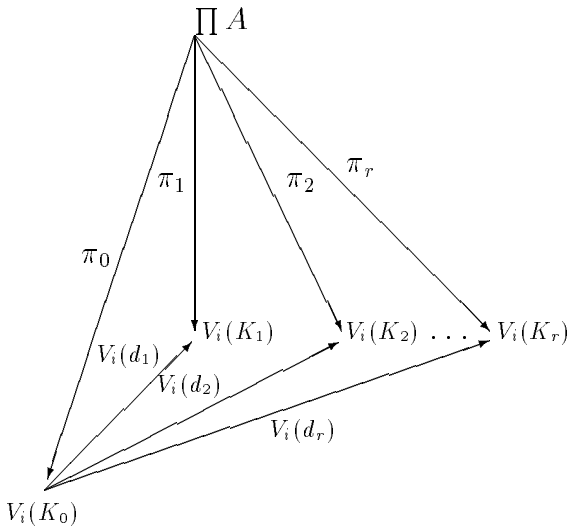


Figure 4: Cone for extension $\prod A$ in the Category **OBJ**

- $\{v_i : \mathbf{1_{TYP_i}} \longrightarrow V_i\}$, 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.

2.6 Encapsulation

The mapping between intension and extension naturally provides an encapsulation of attributes and methods for a class. Operations are only permitted on the extension if they are defined in the intension and are performed so as to enable the functor from intension to extension to preserve consistency.

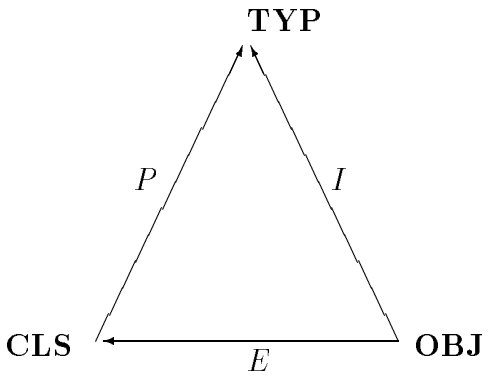


Figure 5: Commuting Diagram for Consistency of Objects with Classes and Types

2.7 Physical Storage Structures

In a similar way to the mapping between classes and objects, it is straightforward to define mappings as functors between categories for objects and categories representing disk structures, say, hash tables or indexes. In earlier work Rossiter and Heather [18] considered the various approaches to hashing in categorical terms.

2.8 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[2]. 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 (see later) 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.

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

2.9 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 [3] 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:

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

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 database such as DAPLEX [21], 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.

2.9.1 Query Example

We take the supplier–parts example given earlier, augmenting it with an inheritance structure where electrical parts are a specialisation 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) $\longrightarrow$ capitalisation

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

- $\mathbf{INT_2}$ for the class $\mathbf{CLS_2}$ 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 $\in A$; $f_6, \ldots, f_8 \in D$.

- $\mathbf{INT_3}$ for the pullback $\mathbf{ASS_1}$ of suppliers and parts over orders as in Figure 2: 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 $\mathbf{CLS_1}$.
  - K_0^2 is the identifier for the *parts* class $\mathbf{CLS_2}$.
  - O is the powerset of *orders*.
  - Instances for O are of the form $\{< k_0^1, k_0^2, o > |\ f(k_0^1) = g(k_0^2), k_0^1 \in K_0^1, k_0^2 \in K_0^2, o \in \wp O\}$.

- $\mathbf{INT_4}$ for the class $\mathbf{CLS_3}$ for electrical parts – a specialisation of parts with object identifier $\mathbf{1_{INT_4}}$ as the identity functor on $\mathbf{INT_4}$
  arrows:
  $f_9 : \mathbf{1_{INT_4}} \longrightarrow$ voltage
  $f_{10} : \mathbf{1_{INT_4}} \longrightarrow$ capacity

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

- $\mathbf{INT_5}$ for the union (coproduct) $\mathbf{UNI_1} = \mathbf{INT_2} + \mathbf{INT_4}$: identifier K_0^2
  arrows:
  $f_6, \ldots, f_8$ from $\mathbf{INT_2}$
  f_9, f_{10} from $\mathbf{INT_4}$
  $s_1 : K_0^2 \longrightarrow \mathbf{1_{INT_4}}$

The natural language query is *'What are the names and identifiers of suppliers with capitalisation 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 $\mathbf{INT_6} = (K_0^2, \mathbf{1}_{\mathbf{INT_4}}$, $voltage \mid voltage = 90)$);

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,capitalisation $\mid$ capitalisation $>$ 1000000));

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

The first functor X_1 derives the subcategory $\mathbf{INT_6}$ from $\mathbf{INT_5}$ by taking the composition of the arrows $s_1 : K_0^2 \longrightarrow \mathbf{1}_{\mathbf{INT_4}}$ and $f_9 : \mathbf{1}_{\mathbf{INT_4}} \longrightarrow voltage$ 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 capitalisation 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.

2.9.2 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

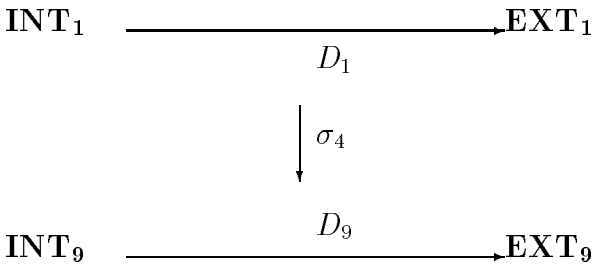


Figure 6: The Query σ_4 as a Natural Transformation with source D_1 and target D_9

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:

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

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

D_1 and D_9 are functors representing intension to extension mapping for the source and target respectively of the query. 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 6 with the query now represented by the natural transformation σ_4.

To be a natural transformation, the square in Figure 6 for our current query σ_4 should commute for every arrow $f_j : \mathrm{dom}(f_j) \longrightarrow \mathrm{cod}(f_j)$ 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 $D_1(\mathrm{dom}(f_j))$ to the values for the codomains of arrows in the target category $D_9(\mathrm{cod}(f_j))$ 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

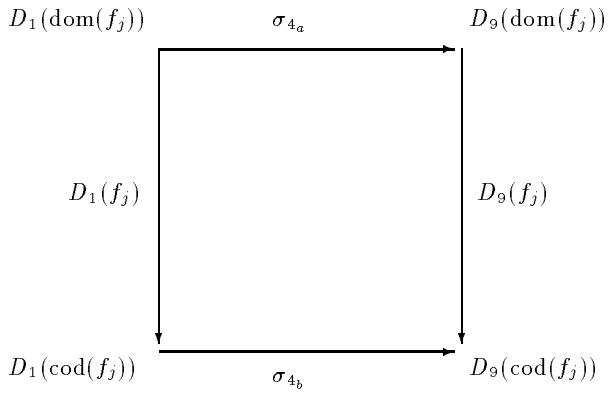


Figure 7: The query σ_4 as a Commuting Target Square with Covariant Natural Transformation σ_4 from functor D_1 to functor D_9

of f_j assigned by D_9 and the subobjects in the target category are defined as the union of $\mathrm{dom}(f_j)$ and $\mathrm{cod}(f_j)$ 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 capitalisations 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 [3].

2.9.3 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 which were 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 8, 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.

2.9.4 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 [3] 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 9 commutes, that is $m_n \circ \eta_{j_a} = \eta_{j_b} \circ m_k$.

Figure 9 is the natural transformation target square and shows that the message passing function is a natural transformation between objects in the category of arrows [23]. 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. We also note that updates can be simply performed as a result of a particular message.

3 Prototyping the Model

To implement any system based on category theory requires finding a suitable language for handling categorical data types, and handling multi–level

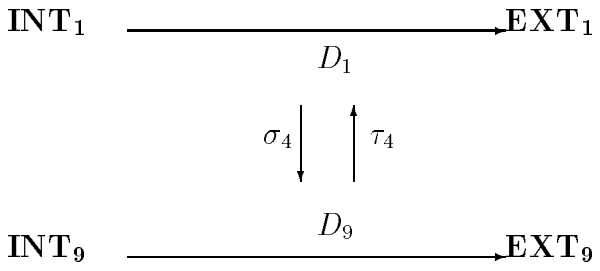


Figure 8: The View σ_4 as a Natural Transformation
with Updates through τ_4

mappings between complex structures. The criteria we have for evaluating languages to determine the most suitable are [16]:

- an ability to handle functions as first class objects;
- a loosely typed language to reduce the difficulty in handling categorical data types;
- the concept of persistency for complex structures, such as categories;
- facilities for a high productivity rate.

Finding a language which best fits these criteria should enable the quick development of the prototype categorical database system. Obviously, if the first three criteria are attainable, then the productivity rate should be quite high, a major advantage in developing a prototype.

An obvious choice was to use a functional language such as ML or Haskell. Previous research by Rydeheard [20, 8] developed a set of categorical data types in the functional language ML, and Duponcheel [9] developed a set of categorical data types in Gofer, a version of Haskell which permits class constructors. The problem with both systems, though, was that functional languages are too strongly typed, and so they do not permit a heterogeneous collection of arrows to be stored easily within a category. Both of their systems really handle only particular types of cartesian closed categories (a category with a continuous function), which is fine for most areas of computing, but falls down when the requirement of a category is to store database properties and functional dependencies, etc.

Another possibility was C++, or some other object–oriented language such as Eiffel or Smalltalk, which may be suitable as they are based on objects, and so should give a natural structure for representing categories. The main problem with object–oriented languages is again in their strong typing, where polymorphism is still too strict to handle the complexity of categorical mappings, and higher order functions would break encapsulation in object–oriented languages. Also, although an object structure can be visualised as being quite

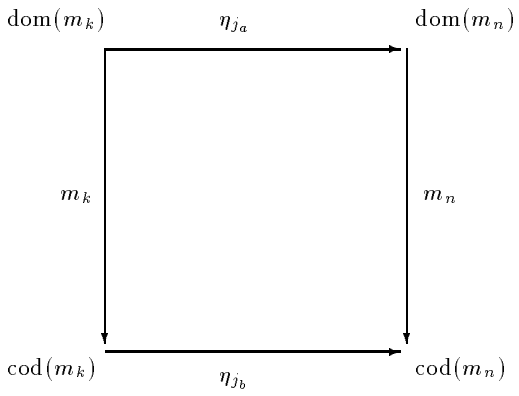


Figure 9: Commuting Square for Message η_j between m_k and m_n in Arrow Categories $\mathbf{CLS}_{\mathbf{i}}^{\rightarrow}$ and $\mathbf{CLS}_{\mathbf{j}}^{\rightarrow}$ respectively

similar to categories, extensibility would be limited in that it would be difficult to add structure to an object once it had been defined.

This led to the P/FDM functional database system, developed by the Object Database group at the University of Aberdeen. P/FDM is a semantic data model database system, with object–oriented extensions. It is based on the functional data model, specifically that of the DAPLEX language, having both a DAPLEX query interface and a query language in SICStus Prolog [22].

The DAPLEX interface is based on the concepts of entities and functions which map entities to other entities, where the functions are either direct (persistent) relations or derived methods. Queries are based on function composition. The use of entities and functions matches quite closely the concepts required for producing a categorical system, and the query language is ideal for handling these categorical structures. Queries and methods in P/FDM can be defined in either DAPLEX or in Prolog, so the system can be enhanced with Prolog extensions when DAPLEX alone is unsuitable. As well as this, P/FDM contains an integral metadata level and support for constraints, which should allow us to perform the necessary consistency checks and type handling that a categorical database would need.

Other advantages of P/FDM are that queries can be closed, which gives us a simple mechanism of storing results from our queries back into the database. It also supports automatic definition of inverses, which gives us a solution for deriving categorical concepts such as duals, adjoints, etc., and we can define subclasses, i.e. (Student is a subclass of Person). Subclasses may be overlapping (i.e. Student is a Person and Student is a Staff, for the case where a student is also employed by the university), but we do not have multiple inheritance, which is not a problem because our categorical system does not currently support multiple inheritance either.

Although it would appear to be advantageous to define arrows in category theory as functions in P/FDM, they are after all similar, there are drawbacks in handling arrows as functions in the model. It is restrictive when storing these arrows within categories, because their source and target entities vary for each arrow and so can not simply be stored in a P/FDM set structure, where P/FDM functions are classified by the type of their source entity (or entities). This implies that it is simpler to store arrows as entities, with two main functions in each one, for referencing the source and target. These arrows can then be stored in a heterogeneous list for a category, where the source and target entities have different types for each member of the set of arrows.

Another concern is that P/FDM is statically typed. For any subclasses which redefine a function from the parent, we must specifically instruct a P/FDM query to use the new function, otherwise the parent function is called instead. This is a problem because we need a form of dynamic binding, since attributes in the database are subclasses of some common attribute superclass, so that arrows only need to know about the common superclass. So it is difficult for us to view the value of an attribute, because we do not know directly the type of the subclass. To get round this, a Prolog method has been defined which first finds out the type of the subclass, and then correctly calls the value method for the subclass, giving us a form of dynamic binding.

3.1 Implementing Partially Ordered Sets

Our method for storing categories as partially ordered sets requires storage of the powerset of attributes, along with the majority of projection arrows (which are trivial functional dependencies) and then adding the extra non–trivial functional dependencies. This is very inefficient in storage terms. So in the implementation, we only store the set of attributes, the non–trivial functional dependencies and the key, and we alter the poset method for determining the key.

The Prolog method recursively subtracts permutations of the non–trivial functional dependencies from the maximal element in the poset (e.g. {a, b, c, d} when attributes are {a}, {b}, {c} and {d}), which gives us a list of powerset members which can be the key, and then by examining the minimality of these elements, we can determine which is the primary key, or which are the candidate keys, if we have a choice. For example, if the attributes are as above, and the functional dependencies are {a, b} $\rightarrow$ {c} and {b, c} $\rightarrow$ {d} (note, this is a pseudotransitivity because we can infer that {a, b} $\rightarrow$ {d}) then the sequence of subtractions is:

{a, b, c, d} – ({a, b}$\rightarrow$ {c}) = {a, b, d} (we remove the target)
{a, b, d} – ({b, c} $\rightarrow$ {d}) = {a, b, d} (we can not complete this subtraction, as c is not in the key)

So, from this permutation, {a, b, d} is the key.

For the second permutation we have:

{a, b, c, d} – ({b, c} $\rightarrow$ {d}) = {a, b, c}
{a, b, c} – ({a, b} $\rightarrow$ {c}) = {a, b}

From this second permutation, {a, b} is the key, and it is the infimum (as {a, b} is minimal compared to {a, b, d}), so the primary key is {a, b} as we would expect.

In this algorithm, we have a straightforward test to determine whether the object conforms to BCNF. The simple test is that the sources of the non–trivial functional dependencies (i.e. {a, b} and {b, c}) are candidate keys. In our example, this is not true, as {b, c} is not a candidate key ({a, b} is the only key, and is therefore the primary key). Our algorithm does not pretend to be highly efficient compared to previous algorithms [17] (where Osborn's algorithm is based on determining the set F^+ [28] of all functional dependencies to check whether a relation is in BCNF) for testing whether a relation is in BCNF, but our algorithm also determines the key whereas previous work does not usually give the key.

3.2 Manipulation

To complement the categorical data types, we need to add some form of manipulation, i.e. queries, closure, views and message passing, as well as some system for actually setting up a database. The intention is that the interface to the user will consist of a collection of pre–written Prolog methods for creating objects, etc. and that the eventual query language should look no different to DAPLEX syntax, so that the user just needs to learn DAPLEX queries, with the required categorical extension.

There may be a difficulty in implementing natural transformations, which will be needed for most of the database manipulation parts. This is because the mapping is between functors and across multiple levels, i.e. the mappings are at the object, arrow and functor level, which may be difficult to represent in a DAPLEX schema, or in many other currently available languages [16].

4 Conclusions

We consider that our work is a positive contribution towards solving the three main problems in object–oriented databases simply by using subcategories and natural transformations in category theory, and that our use of P/FDM provides the necessary functionality for actually implementing this categorical model. The implementation is a non–trivial problem because category theory constructs do not map in a direct manner onto the constructions of current programming languages, but the combination of DAPLEX and Prolog appears promising.

5 Acknowledgements

Finally, we must thank Professor Peter Gray and his associates at Aberdeen University for making the P/FDM system available for our use. In particular, Doctor Suzanne Embury for the many hours spent fixing any problems that have occurred so far. We would also like to thank Doctor Michael Heather at

the University of Northumbria for the valuable discussions relating to category theory.

References

[1] *ISO–ANSI SQL 3 Working Draft.* Digital Equipment Corporation, Massachusetts, March 1994.

[2] M. Atkinson, et. al. The Object–Oriented Database System Manifesto. In F. Bancilhon, et. al. *The Story of O_2: Implementing an Object–Oriented Database System*, Morgan Kaufmann, 1992.

[3] M. Barr, C. Wells. *Category Theory for Computing Science.* Prentice–Hall International Series in Computer Science, 1990.

[4] B. Cadish, Z. Diskin. Algebraic Graph–Oriented = Category Theory Based: Categorical Data Modelling Manifesto. Frame Inform Systems, Database Design Laboratory, Latvija, *DBDL Research Report FIS/DBDL–94–02,* July 1994.

[5] L. Cardelli. A Semantics of Multiple Inheritance. *LNCS*, 173:51–67, 1984.

[6] P. P. Chen. The Entity–Relationship Model : Toward a Unified View of Data. *ACM TODS*, 1(1):9–36, March 1976.

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

[8] E. Dennis–Jones, D. E. Rydeheard. Categorical ML – Category–Theoretic Modular Programming. *Formal Aspects of Computing* , 5(4):337–366, 1993.

[9] L. Duponcheel. *Gofer Experimental Prelude.* Alcatel, Belgium, 1994.

[10] S. M. Embury, et. al. *User Manual for P/FDM Version 9.0.* University of Aberdeen, Technical Report AUCS/TR9501, January 1995.

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

[12] P. M. D. Gray, K. G. Kulkarni, and N. W. Paton. *Object–Oriented Databases: A Semantic Data Model Approach.* Prentice–Hall International Series in Computer Science, 1992.

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

[14] S. Mac Lane, I. Moerdijk. *Sheaves in Geometry and Logic, A First Introduction to Topos Theory.* Springer–Verlag 1991.

[15] D. A. Nelson, B. N. Rossiter, and M. A. Heather. *The Functorial Data Model – An Extension to Functional Databases.* University of Newcastle upon Tyne, Technical Report Series, No. 488, 1994.

[16] D. A. Nelson, B. N. Rossiter. *Suitability of Programming Languages for Categorical Databases.* University of Newcastle upon Tyne, Technical Report Series, No. 511, March 1995.

[17] S. L. Osborn. Testing for Existence of a Covering Boyce Codd Normal Form. *Information Processing Letters*, 8(1):11–14, January 1979.

[18] 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, University of Newcastle upon Tyne.

[19] B. N. Rossiter, M. A. Heather. *Database Architecture and Functional Dependencies Expressed with Formal Categories and Functors.* University of Newcastle upon Tyne, Technical Report Series, No. 432, 1993.

Categorical

[20] D. E. Rydeheard, R. M. Burstall. *Computational Category Theory.* Prentice–Hall International Series in Computer Science, 1988.

[21] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM TODS*, 6(1):140–173, March 1981.

[22] *SICStus Prolog User's Manual, Edition 2.1, Patch #7.* Swedish Institute of Computer Science, January 1993.

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

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

[25] M. Stonebraker, L. A. Rowe. The Design of Postgres. In *Proceedings ACM SIGMOD Conference*, pages 340–355, 1986.

[26] M. Stonebraker. *Object–Relational Database Systems.* Montage Software Inc., 1994.

[27] D. Tsichritzis. ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Data Base Management Systems. *Information Systems*, 3(3):173–192, 1978.

[28] J. D. Ullman. *Principles of Database and Knowledge–Base Systems* **1**. Computer Science Press, 1988.