Prototyping a Formal Object-Oriented Database in P/FDM

D.A. Nelson, B.N. Rossiter

Dept. of Computing Science, University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK
e-mail: D.A. Nelson@newcastle.ac.uk

July 1995

Abstract

This paper is concerned with a formal model for object databases. Category theory 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. 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.

About the author

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

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

Suggested Keywords

object–relational database model, object–oriented databases, category theory, P/FDM, product model.

This paper was initially presented at the 7th ERCIM Database Research Group Workshop on Object-Oriented Database, Lisbon, Portugal, May 15-16, 1995.

1 Introduction

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 [Barr90] to provide a formalism for object databases[1], known as the product model.

This paper will highlight the key aspects of the product model, 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 do investigate the more common database concepts, such as keys, relationships, aggregation, etc., and we will also discuss how our model approaches these concepts.

A prototype of this model is currently being produced, using P/FDM [Embury95, Gray92], a database system based on the functional data model of Shipman [Shipman81], 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.

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 [Stonebraker94] is similar to our formalism for object-databases, while our relationships are similar in functionality and appearance to those in the entity-relationship model [Chen76]. We also use Boyce Codd Normal Form [Ullman88] as a normalisation constraint when determining the keys in a particular database object, ensuring a high level of consistency in the database.

The three main problems apparent in most object database systems of today are those of creating views, closure from queries, and the query languages themselves. Our categorical model attempts to solve these three problems.

Because many of the current object-oriented databases are based heavily on C++ (or some other object-oriented programming language), they are usually no more than just persistent object-stores. This means that views and closure are difficult to implement

[1] not necessarily object-oriented, but one which contains most of the concepts from an object-oriented model

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 [ANSI94] being written. Many of the current systems usually provide no more than C++ queries though, i.e. the application developer must write any queries needed as C++ methods.

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

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, etc. in the database. Category theory is also based on the arrow as its primitive concept, giving natural modelling of dynamic as well as static aspects. 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.

The categorical data modelling manifesto by Cadish and Diskin [Cadish94], 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.

The rest of this paper will give an overview of the categorical concepts used for the product model, and highlight 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

Our product model is based mainly on the concept of products and coproducts (sums) in category theory, which give us database concepts such as relationships, aggregation, inheritance, etc.

2.1 Classes and Objects

The basic concept of category theory is the category. A category is a collection of arrows which provide mappings between attributes[2]. These arrows may be identity arrows, and are composable if the source of one arrow is the target of another. We use these categories for representing objects in the database. Categories are used to represent both the class intension of the database object and the instances of a class, known as the extension. The attributes of the database object, both intensionally and extensionally, are represented as a partially ordered set (POSET) of projection arrows, where these arrows are the trivial functional dependencies within the object.

By adding the extra non-trivial functional dependencies to the partially ordered set, we can then determine the primary key of the object, by finding the new infimum (greatest lower bound) of the POSET with the extra arrows incorporated. If an infimum does not exist, then the set of maximal lower bounds gives us a choice of candidate keys, where it is then really up to user discretion about which is most suitable for use as the primary key. This method also provides a normalisation test, to check whether the object conforms to Boyce Codd Normal Form (BCNF), i.e. all the determinants (sources of non-trivial functional dependencies) are candidate keys. The example POSET in figure one illustrates this method, with a single non-trivial functional dependency $\{a\} \rightarrow \{b\}$ added.

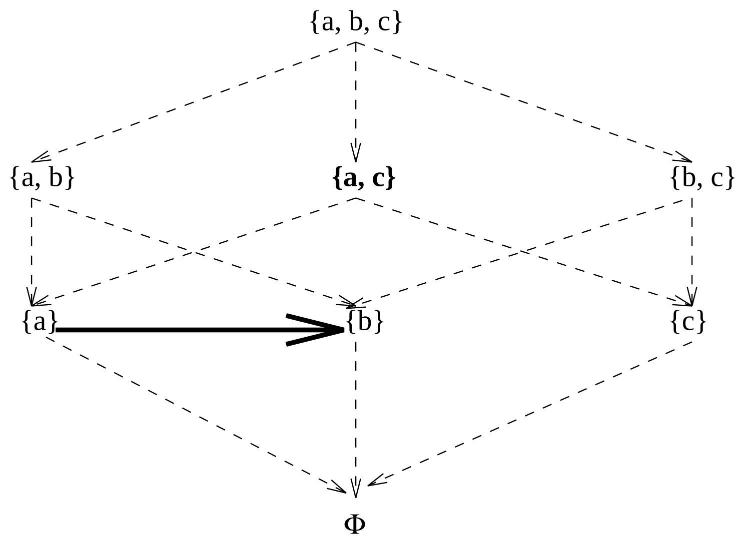

Figure 1: Partially ordered set with non-trivial functional dependency $\{a\} \rightarrow \{b\}$

The primary key of an object is the initial object of the category, and gives us an unique object identifier for each object in the database. This is different to most other object databases, where object identifiers are a system defined concept, and never have any obvious relationship to the data stored other than as a way of referencing a particular object.

[2]usual definition is that mappings are between objects, but to avoid confusion between categorical objects and database objects, we are calling the categorical objects attributes

The primary key determination, and BCNF test, are both carried out on the intension category, which stores the names of the properties of an object. These properties of an object can be both attributes, i.e. the actual persistent data values which we wish to store, and derived data, as the result of some method on other persistent and non-persistent values. As stated previously, in the intension category, the names of attributes, and method names as mappings, are stored. The extension category represents the instances of the intension, and thus stores attribute values. A third category stores the type information for a particular object, storing the domain for each attribute in the object.

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

Figure 2: Mappings between intension, extension and type category

The arrows between the three categories are functors, where a functor is a mapping between the attributes and arrows in a source category to the attributes and arrows in a target category. So, the diagram in figure two commutes, i.e. the values stored in the extension categories are consistent (that is they have a correct class intension, and the types of its values are correct) when $P \circ E = I$, where TYP is the type category, CLS is the intension category, OBJ is the extension category, and P, I and E are the mappings between those three categories.

2.2 Relationships

With the above concepts, we have the capability of representing database objects, ensuring that the naming and typing of attributes and methods in the object are consistent. We will now look at the two forms of relationship in the object-relational model, the binary relation between two objects, and inheritance. Note that aggregation is easily supported as either an arrow within a category mapping to a set of values, or through using the binary relationship concept.

2.2.1 Binary Relations

The categorical concept of a pullback allows us to represent relationships between objects in the database, with the functionality of relationships in the entity-relationship (E-R) model. A pullback is a product between two attributes (in this case, we use the initial objects in the database objects) restricted over some other attribute, such as the set of orders (O) in the relationship between suppliers (S) and parts (P), as shown in figure three.

$$\begin{tikzcd} & S \arrow[dr, "f"] & \\ S \times_O P \arrow[ur, "\pi_l"] \arrow[dr, "\pi_r"'] & & O \\ & P \arrow[ur, "g"'] & \end{tikzcd}$$

Figure 3: Relationship between suppliers and parts as pullback

The typing of the projection arrows gives both the cardinality (many or one) for the particular object in the relationship, and the membership class, i.e. mandatory or optional participation, equivalent to those in the E-R model:

- if π_l is epimorphic (onto) then every S appears at least once in the relationship with each P, i.e. its participation is mandatory, otherwise it is optional;
- if π_l is monic (one-to-one) then every S appears only once in the relationship with each P, otherwise it may appear more than once in the relationship.

The following table outlines some of the possible combinations of cardinality and membership class for a relationship between A and C over B, illustrated with examples, and showing how a relationship in the pullback is equivalent to a relationship in an entity-relationship diagram. For example, N:1 in our model is equivalent to 1:N or one-to-many in the E-R model, and o and m for the membership classes in the table are optional and mandatory respectively.

A	C	B	π_l epic	π_l mon	π_r epic	π_r mon	relationship partic A:C	relationship mapping A:C	relationship memb.cl. A	relationship memb.cl. 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

Table 1: Comparison of Pullback relationships to E-R model

Note that the concept of binary relationships can easily be extended to n-ary by just extending the product over three or more attributes.

2.2.2 Inheritance

Because our model is object-relational, then it seems wise to provide some form of inheritance (specialisation). We use the concept of a coproduct, which is the disjoint union between two attributes. The diagram in figure four illustrates an example, where Student is the new subclass of Person.

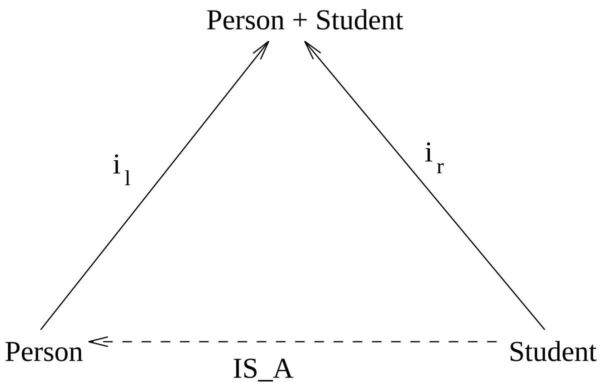

Figure 4: Inheritance coproduct diagram

At the base of the coproduct cone is the Person object and the Student object, representing the additional mappings which are to be included in the new subclass. The top of the cone is the new subclass, created via inclusion arrows from the base objects. Note the IS_A arrow added to the usual coproduct structure: it is there to show the direction of the inheritance.

Because the operation is a coproduct, i.e. a disjoint union, then this method does not easily handle multiple inheritance. This is because the disjoint union would exclude any properties or arrows which appeared in both parent categories at the base of the cone.

In any case, the semantics of multiple inheritance are very difficult to conceptualise and thus it is not a bad thing to exclude multiple inheritance from the model. Our model does handle a simple hierarchy though, as the top of one cone can easily form the base of another coproduct cone.

2.3 Manipulation

Our categorical model uses natural transformations for mappings between intension-extension object pairs (where an object is represented as an intension category and a collection of values are represented as an extension category), giving intension-extension subobject pairs (as subcategories), for naturally modelling the concepts of queries, closure and views. A natural transformation in category theory is a mapping between two functors, although, as with functors, the mapping is across multiple levels, i.e. it involves the attribute, arrow, category, etc., levels of the source and target functors. We also use natural transformations for message passing.

2.3.1 Queries and Closure

In most object-databases, closure, which is where the result of a query can be used in a further query, or stored back into the database schema, is a difficult concept to handle. For example, in C++, closure from a query would involve automatically producing a new class intension for the resulting properties produced by the query, quite often as a combination of properties from more than one class in the schema.

We use subcategories for providing closed objects. A subcategory is a category which contains some of the mappings and attributes of a parent category, i.e. a form of 'subset', but on mappings as well as attributes. There are two specific types of subcategory. One which contains all of the arrows for each pair of attributes in the parent category is known as a *full* subcategory, whereas a subcategory which contains all of the attributes as in the parent category is a *wide* subcategory. Obviously, any category is a full wide subcategory of itself.

So, if we ensure that the results of our queries are subcategories, then because they are essentially no different from normal categories, they can be included back into the schema, and can be queried further. In fact, the ability to query further a subcategory leads us to a method for producing queries as a sequence of category - subcategory transformations.

A simple query in the model is that of creating a subcategory, where the only instances which exist are those that satisfy the query condition. We want our queries to be a collection of simple query steps, i.e. a composition of category - subcategory transformations (functors). So in effect, our queries (at an intensional level) are compositions of functors.

Because we want the query to map over both intension and extension, then we are in fact mapping between functors, i.e. a natural transformation. A single step in the query is a natural transformation, illustrated by the commuting square in figure five:

$$\begin{tikzcd} I \arrow[r, "D"] \arrow[d, "S"'] & E \arrow[d, "T"] \\ I' \arrow[r, "D'"'] & E' \end{tikzcd}$$

Figure 5: Natural transformation square

where D is the functor mapping source intension to extension, D′ is the functor mapping target (i.e. result of the query as subcategories) intension to extension, S is a mapping between source and target intension, and T is a functor mapping source and target extension. Compositions of these natural transformations, one per stage of the query, gives complete queries, where the query condition could be any predicate logic or calculus.

Finally, note that because pullbacks are represented in the schema as categories, then joins can be performed by just operating on a pullback instead of a database object, returning a subcategory which can again be used further in the query.

2.3.2 Views

Views are similar to queries, in that we want to produce some 'subset' of the data currently in the database, to give the user a different perception of the schema. The only differences are that rather than producing a view on a single category, we may wish it to be something more complex than this, i.e. a combination of categories. In this case, we use the concept of families of categories [Barr90], where a collection of categories can be visualised as one category, so that the properties of the category are categories.

Also, because a view may be updated, then these updates need to be propagated back to the original version of the database, i.e. the source categories. So whereas a query has a natural transformation from source to target, we need also a natural transformation, as a dual, from target to source.

2.3.3 Message Passing

Message passing is the ability to call a method either within an object (intra-object) or between two objects (inter-object), which may also change the state of some property in the category. In our model, message passing is a function between arrows, within the category of arrows [Barr90], where the attributes in the categories are arrows, and the mappings are between arrows. For example:

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

where $CLS_i^{\rightarrow}$ is the arrow category for the ith object in the schema, and η_j is a message from the the property (or morphism) m_k in arrow category $CLS_i^{\rightarrow}$ to m_n in arrow category $CLS_j^{\rightarrow}$.

Again, we have a natural transformation square, similar to the one for a query, where the message is a natural transformation between properties in the category of arrows.

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 mappings between complex structures. The criteria we have for evaluating languages to determine the most suitable are [Nelson95]:

- 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 quick development of a 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 [Rydeheard88, Dennis-Jones93] developed a set of categorical data types in the functional language ML, and Duponcheel [Duponcheel94] 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 break encapsulation in object-oriented languages. Also, although an object structure can be visualised as quite 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 data definition and manipulation language, with both a DAPLEX query interface, and a query language in SICStus Prolog [SICStus93].

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 specialisation 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, i.e. it is restrictive when storing them 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. 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 tell 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 that 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 (i.e. {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} → {c} and {b, c} → {d} (note, this is a pseudotransitivity [Rossiter95] because we can infer that {a, b} → {d}) then the sequence of subtractions is:

{a, b, c, d} - ({a, b}→ {c}) = {a, b, d} (we remove the target)
{a, b, d} - ({b, c} → {d}) = {a, b, d} (we can not complete this subtraction, as b, c and d are 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} → {d}) = {a, b, c}
{a, b, c} - ({a, b} → {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 Boyce Codd Normal Form. 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 [Osborn79] (where Osborn's algorithm is based on determining the set F+ [Ullman88] of all functional dependencies to see 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 extensions.

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 [Nelson95].

4 Conclusions

We consider that our work is a positive contribution towards solving the three main problems in object-oriented databases simply by subcategories and natural transformations in category theory, and that our using 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.

6 References

[ANSI94]
ISO-ANSI SQL 3 Working Draft. Digital Equipment Corporation, Massachusetts, March 1994.

[Barr90]
Barr M, Wells C. *Category Theory for Computing Science.* Prentice-Hall International Series in Computer Science, 1990.

[Cadish94]
Cadish B, Diskin Z. *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.

[Chen76]
Chen P P. *The Entity-Relationship Model : Toward a Unified View of Data.* ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976.

[Dennis-Jones93]
Dennis-Jones E, Rydeheard D E. *Categorical ML - Category-Theoretic Modular Programming.* Formal Aspects of Computing, Vol. 5, No. 4, 1993.

[Duponcheel94]
Duponcheel L. *Gofer Experimental Prelude.* Alcatel, Belgium, 1994.

[Embury95]
Embury S M, et. al. *User Manual for P/FDM Version 9.0.* University of Aberdeen, Technical Report AUCS/TR9501, January 1995.

[Gray92]
Gray P M D, Kulkarni K G, Paton N W. *Object-Oriented Databases : A Semantic Data Model Approach.* Prentice-Hall International Series in Computer Science, 1992.

[Nelson95]
Nelson D A, Rossiter B N. *Suitability of Programming Languages for Categorical Databases.* University of Newcastle upon Tyne, Technical Report Series, No. 511, March 1995.

[Osborn79]
Osborn S L. *Testing for Existence of a Covering Boyce Codd Normal Form.* Information Processing Letters, Vol. 8, No. 1, January 1979.

[Rossiter95]
Rossiter B N, Nelson D A, Heather M A. *The Categorical Product Data Model as a Formalism for Object-Relational Databases.* University of Newcastle upon Tyne, Technical Report Series, No. 505, February 1995.

[Rydeheard88]
Rydeheard D E, Burstall R M. *Computational Category Theory.* Prentice-Hall International Series in Computer Science, 1988.

[Shipman81]
Shipman D W. *The Functional Data Model and the Data Language DAPLEX.* ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981.

[SICStus93]
SICStus Prolog User's Manual, Edition 2.1, Patch #7. Swedish Institute of Computer Science, January 1993.

[Stonebraker94]
Stonebraker M. *Object-Relational Database Systems.* Montage Software Inc., 1994.

[Ullman88]
Ullman J D. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.