To Formalise and Implement a Categorical Object-Relational Database System

Department of Computing Science

University of Newcastle upon Tyne

PhD Thesis

David Alan Nelson

December 1998

NEWCASTLE UNIVERSITY LIBRARY

098 14257 6

Thesis L6319

1. INTRODUCTION 8

1.1 Overview 8

1.2 Rest of the thesis 10

2. DATA MODEL SURVEY 12

2.1 Introduction 12

2.2 Overview of Data Models 12
2.2.1 Definitions 13

2.3 The Relational Model 17
2.3.1 Structure 17
2.3.2 Rules 18
2.3.3 Manipulation 18
2.3.4 Strengths 19
2.3.5 Weaknesses 19
2.3.6 Extensions to the Relational Model 20
2.3.6.1 Nested Relational Model 20
2.3.6.2 RM/T 20
2.3.6.3 Object modelling 21

2.4 Functional Models 21
2.4.1 DAPLEX 22
2.4.2 Structure 25
2.4.3 Rules 25
2.4.4 Manipulation 26
2.4.5 Strengths 26
2.4.6 Weaknesses 26
2.4.7 P/FDM 27

2.5 Object-Oriented Models 27
2.5.1 Structure 28
2.5.2 Rules 28

2.5.3 Manipulation 29
2.5.4 Strengths 30
2.5.5 Weaknesses 30
2.5.6 Object-Oriented Systems 30

2.6 Discussion 32

2.7 Summary and Conclusions 33

3. OBJECT-RELATIONAL MODELS 35

3.1 Introduction 35

3.2 The Postgres Database System 35
3.2.1 Introduction and Aims 35
3.2.2 The Data Model 36
3.2.3 Data Manipulation in Postgres 39

3.3 Comparison with other models 42

3.4 Experience with Postgres 42

3.5 Other Object-Relational Database Systems 43
3.5.1 Montage 43
3.5.2 Matisse 44
3.5.3 UniSQL/X 45
3.5.4 OpenODB 46

3.6 Tabular Comparison of Data Models 46

3.7 Problems with Object-Relational Systems 48

3.8 Summary 48

4. CATEGORY THEORY 50

4.1 Introduction 50

4.2 Category Theory 50
4.2.1 Categories 50
4.2.2 Subcategories 52

4.2.3 Functors 53
4.2.4 Typing 53
4.2.5 Product Cones 54
4.2.5.1 Product and Projection 54
4.2.5.2 Coproduct and Inclusion 55
4.2.5.3 Pullbacks 56
4.2.5.4 Pushouts 56
4.2.6 Limits 57
4.2.7 Natural Transformations 57

4.3 Why Category Theory 59

4.4 Previous Work 60

4.5 Conclusions 63

5. THE PRODUCT MODEL 64

5.1 Introduction 64

5.2 Objectives 64

5.3 Classes 65
5.3.1 Basic Structures 65
5.3.2 Identifiers 68

5.4 Relationships 71
5.4.1 Enhancements 74
5.4.2 Pullback Identifiers 74
5.4.3 Inheritance 74
5.4.4 Composition 76

5.5 Typing 77

5.6 Objects 77

5.7 Encapsulation 80

5.8 Physical Storage Structures 80

5.9 Families of Categories 81

5.10 Manipulation 81

5.10.1 Query Example 83

5.10.2 Closure in Queries 87

5.10.3 Views on Classes 89

5.10.4 Message Passing 90

5.11 Summary 92

6. IMPLEMENTATION 95

6.1 DAPLEX 95

6.2 An Extended Review of P/FDM 98

6.3 Implementation 102

6.3.1 Introduction 103

6.3.2 Classes 104

6.3.3 Relationships 106

6.3.3.1 Pullback relationships 107

6.3.3.2 Inheritance 109

6.3.4 Typing 110

6.3.5 Objects 113

6.3.6 Encapsulation 114

6.3.7 Physical Storage Structures 114

6.3.8 Families of Categories 115

6.3.9 Functors 115

6.3.10 Manipulation 116

6.3.10.1 Closure 119

6.3.10.2 Queries 119

6.3.10.3 Views 120

6.4 Conclusions 121

7. RESULTS 122

7.1 Introduction 122

7.2 Testing Method 122

7.3 Manipulation Technique 123

7.3.1 Other Manipulation Functionality 123

7.4 Simple Test - Supplier Parts Example 125

7.5 More Complex Test - Product Model Example 133

7.5.1 Query in SQL 134

7.5.2 Query in P/FDM 135

7.5.3 Query in the Product Model 136

7.6 Conclusions 148

8. DISCUSSION AND CONCLUSIONS 150

8.1 Modelling Aspects 150

8.2 Implementation Areas 152

8.3 Problem Areas and Future Thoughts 153

9. REFERENCES 158

Table of Figures

FIGURE 3-1 - COMPARISON OF DATA MODELS 47
FIGURE 4-1 - COMMUTING TRIANGLE 52
FIGURE 4-2 - PRODUCT CONE 54
FIGURE 4-3 - COMMUTING PRODUCT CONE 55
FIGURE 4-4 - COPRODUCT CONE 55
FIGURE 4-5 - PULLBACK 56
FIGURE 4-6 - PUSHOUT 57
FIGURE 4-7 - NATURAL TRANSFORMATION 58
FIGURE 4-8 - NATURAL TRANSFORMATION AS A COMMUTING PRODUCT CONE 58
FIGURE 5-1 - INJECTION OF TRIVIAL AND NON-TRIVIAL FUNCTIONAL 69
FIGURE 5-2 - DIAGRAM OF PULLBACK OF K_0^1 AND K_0^2 OVER O 71
FIGURE 5-3 - COPRODUCT CONE FOR OBJECTS CLS_3 AND CLS_4 75
FIGURE 5-4 - CONE FOR EXTENSION ΠA IN THE CATEGORY OBJ 78
FIGURE 5-5 - COMMUTING DIAGRAM FOR CONSISTENCY OF OBJECTS WITH 79
FIGURE 5-6 - THE QUERY σ_4 AS A NATURAL TRANSFORMATION WITH SOURCE D_1 AND TARGET D_9 87
FIGURE 5-7 - THE QUERY σ_4 AS A COMMUTING TARGET SQUARE WITH COVARIANT NATURAL TRANSFORMATION σ_4 FROM FUNCTOR D_1 TO FUNCTOR D_9 87
FIGURE 5-8 - THE VIEW σ_4 AS A NATURAL TRANSFORMATION WITH UPDATES THROUGH τ_4 91
FIGURE 5-9 - COMMUTING SQUARE FOR MESSAGE η_J BETWEEN M_K AND M_N IN ARROW CATEGORIES $\overrightarrow{CLS_I}$ AND $\overrightarrow{CLS_J}$ RESPECTIVELY 92
FIGURE 5-10 - SYMBOLS EMPLOYED FOR REPRESENTING DATABASE CONCEPTS 94
FIGURE 6-1 - DIAGRAM OF ARCHITECTURE OF P/FDM SYSTEM 99
FIGURE 6-2 - DIAGRAM OF ARCHITECTURE OF PRODUCT MODEL 103
FIGURE 6-3 - NATURAL TRANSFORMATION IN P/FDM 117
FIGURE 8-1 : USE OF PUSHOUTS TO MODEL MULTIPLE INHERITANCE 154

Abstract

The relational data model uses set theory to provide a formal background, thus ensuring a rigorous mathematical data model with support for manipulation. Newer generation database models are based on the object-oriented paradigm, and so fall short of having such a formal background, especially in some of the more complex data manipulation areas. We use category theory to provide a formalism for object databases, in particular the object-relational model. Our model is known as the Product Model.

This thesis will describe our formal model for the key aspects of object databases. In particular, we will examine how the Product 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 and aggregation. We will illustrate the feasibility of this model, by producing a prototype implementation using P/FDM. P/FDM is a semantic data model database system based on the functional model of Shipman, with object-oriented extensions.

1. Introduction

1.1 Overview

The most successful data model developed to date in commercial terms is undoubtedly the relational model [cod70], which was first outlined by Codd in 1970, and is today the most widely used.

One of the main reasons for the success of the relational model is its underlying formal basis, with a model based directly on set theory [dat95]. This has ensured a clear structure for storage of data and a formal manner for manipulating that stored data, namely the relational algebra based on algebraic manipulation of sets, and the relational calculus, which incorporates predicate logic for manipulating the sets. Out of the relational calculus has arisen the standard query language SQL [dat95] first designed by IBM, which has been incorporated in many of the commercial and academic relational database systems that have been produced over the last twenty years or so.

One of the greatest advancements in computing lately has been the introduction of the object-oriented paradigm, initially as the Smalltalk language [gol83], but much more readily accepted with the development of the C++ language [str91], as well as object-oriented analysis and design methodologies such as Booch [boo94], Coad-Yourdon [coa91], Rumbaugh [rum91], Unified Modelling Language (UML) [rat98] and Brathwaite [bra93], and recently in the development of object-oriented databases. The main effects of the object-oriented paradigm have been to provide abstractions in a more natural way compared to previous paradigms, that is objects and their associations more naturally model real world structures and the relationships between them. Some of these abstractions are ideal for modelling complex database structures, as we discuss in a later chapter.

Object models, meanwhile, were developed directly as a result of object-oriented programming techniques and the object-oriented paradigm. However, many of the earlier systems have simply been extensions to object-oriented languages, in particular C++, and therefore they strongly lack a rigorous formal basis. This has led to a natural structure for storage of complex data structures, but problems in querying and other forms of manipulation, such as relational joins, products and closure, which were handled successfully in the relational model.

ODMG-93 - the Object Database Standard [cat94], has been devised by some of the major players currently involved in OO databases. It is strongly based on C++ but can be mapped to other object-oriented systems. ODMG however suffers the same drawback as other object-oriented systems in that it lacks a clear formal basis, and thus is not as good as relational databases in terms of having a theoretical basis and therefore in providing features that arise from a model with a strong underlying basis. Their OQL [cat94] language however is defined as a superset of SQL and thus alleviates the query problems mentioned above, but it is not clear which implementations support it in full at the moment.

Another offspring of object models, and perhaps more successful is the object-relational model [sto94]. This takes the same idea as extended relational models such as the nested relational model [rot88] and RM/T [cod79], but applies object-oriented concepts to its basic structures, in an attempt to provide a model which is the best of both worlds. It introduces the necessary concepts of the object-oriented paradigm to enable the database systems to model the more complex structures that are used in areas such as graphics and audio in multimedia, but at the same time it ensures some form of 'backwards compatibility' so that relational users have a clear path into object databases. In addition, the rigour and formality of relational models is maintained, an aspect which is lacking in pure object-oriented databases.

Some object models have been formalised using set theory, as in relational models, but set theory is not really powerful enough to model naturally the multi-level complex

architecture and abstractions of object databases. We see later that the multi-level architecture of category theory and its rich collection of tools seems much more natural for more complex systems.

This thesis is an attempt to produce a formal model for the object databases, in particular the object-relational model, using category theory [bar90].

1.2 Rest of the thesis

The subsequent chapters of this thesis will be structured in the following manner.

In chapter two, there will be a detailed discussion of the data models in existence, highlighting, with examples, their advantages and disadvantages. This will look mainly at the relational and object-based models, but also the functional data model [sib77] which is a relatively natural data model for manipulation, being based on function composition, and which some object models have tried to incorporate in their systems.

Then, in chapter three we will continue with a more detailed review of object-relational databases, in particular looking at the Postgres [sto86a] system, which was developed as a research project at Berkeley, and has been further developed into the commercial system Montage [ols94]. We will then finish the review of existing database models by comparing the main features of the models presented in this and the last chapter with a table, illustrating why object-relational appears to be the most promising of the newer generation models, and giving our views on future issues in database models.

Chapter four will introduce the concepts of category theory that are used in defining our categorical model, and explain our motivations for using category theory as the formal model.

In chapter five, we will describe in detail the Product Model, our categorical formalism for object models, in detail. We will do this by building on the categorical concepts introduced in chapter four.

Chapter six will look at the implementation, concentrating on two points. Firstly a review of our implementation base P/FDM is given. Secondly we will show how the database concepts we modelled in chapter four were represented in the prototype, in particular showing any changes that were required to the model to allow it to be effectively implemented.

Chapter seven will then discuss the results obtained from prototyping the model with a collection of small test applications.

Finally, in chapter eight we will conclude the work by examining the potential contributions of the Product Model to the current new generation data models, and highlighting how it fits in with the object-relational model. We will also outline the possibilities for future work in this area.

2. Data Model Survey

2.1 Introduction

In this chapter we will look in detail at the main data models currently in existence, from a historical perspective, highlighting the major advantages and disadvantages of each one. The difficulties found in some respect with every model illustrate the need for further models such as the categorical model introduced in chapter five.

2.2 Overview of Data Models

There are three main facilities which any data model must provide in order for it to be usable with real world data:

1. structure, expressed as a schema in terms of projection of attributes, domains, inter-object and intra-object associations, in the form of a data definition language;
2. rules, which are any restrictions on that schema definition, most notably the normalisation and integrity constraints in relational databases;
3. manipulation, where searching and updating of the database is performed by a query language.

Currently, the most widely used data model is the relational model [cod70, dat95], but this model is based on tables (relations) which are an unnatural concept when applied to real world problems. The newer generation databases such as the object-oriented model [atk90], the nested relational model [cod79, rot88] and the functional model [ker76, shi81] have attempted to alleviate the problems of atomicity by adding more natural association abstractions. The most recent data model, the object-oriented model, provides probably the best data structure for representing and manipulating real world data, but the object-oriented model introduces problems of its own, namely that

it lacks a universal formal basis [kim90], a manipulation language as rich as the relational algebra, and a view mechanism. The OQL language [cat94], if successfully implemented, will alleviate some of these problems.

For each model we will provide a description of its major concepts, then look further at structure, rules and operations defined in the model. Then we will look at areas covered in the context of the types of application appropriate for each model. Finally the strengths and weaknesses of each model will be summarised in a table in the following chapter.

2.2.1 Definitions

Throughout the next two chapters, we introduce terms (or concepts) applicable to the various data models, and give a table showing which of these concepts each model supports. Here, we define all of those terms[1]:

- **tree** - the structure used in the hierarchical data model [elm94], where data structures are represented as a hierarchical tree of information, a hierarchy representing a number of related records (by some sort of classification scheme, for example employer employs employees, student is a person).

- **network** - the network model [elm94] (also known as the CODASYL DBTG model) is similar to the hierarchical model. However, the structures are no longer restricted to being trees, but in fact represent records as set-types (one-to-many relationships), that is the nodes are records and the links represent links as set membership operations to other records.

---

[1] We take most of the definitions for our terms from Date [dat95] and Elmasri and Navathe [elm94].

- **relation** - this is a term defined by Codd [cod70] which is the backbone of the relational data model. A relation represents a collection of information about some entity, with data independence, that is it is possible to add new types of stored information typically to represent new types of entity without having to change the application [dat95]. A relation in the relational model is a table, or can be thought of as being equivalent to a set of Pascal records (but with the restriction that none of the elements of the record are pointer elements).

- **referential and entity integrity** - the referential integrity constraint ensures that relations between two tuples (that is rows in a table in relational databases) must be consistent [elm94], that is the relevant tuples in each table must actually exist, and there must be no unmatched **foreign key** values [dat95], that is there must be no foreign key values where there does not exist a matching primary key value in the relevant target relation. A foreign key is a collection of attributes in one table which provide the join to the primary key in another table. The **entity integrity** constraint ensures that no tables have tuples with a null component in the **primary key,** the collection of properties which uniquely defines a tuple in a table.

- **normalisation** - this is a concept which was defined for the relational model as a series of tests to certify whether a relational schema (a group of relations which represent an overall relational database) conforms to certain rules (normal forms) about structure of the properties within relations [elm94]. The various normal forms are well defined in books about the relational database model such as Date and Elmasri and Navathe [dat95, elm94], so we do not define them here. These rules have been found to be very important in ensuring data schemas are highly consistent (especially in terms of ensuring that no information is lost when other information is deleted for example).

- **first normal form (1NF)** - the first normal form, which states that all values must be atomic, that is a particular tuple must only have one value for each attribute, and not a group of attributes.

- **algebra/calculus** - the relational algebra and calculus are the two main formal definitions for query and manipulation languages in the relational model. The algebra is based on mathematical algebraic operations and the calculus is based on predicate calculus operations. The relational calculus is the most widely used, both SQL and QUEL are examples of extensions to the relational calculus (see later).

- **closure** - a term normally applied to queries where the result of a query (in a relational database) is another relation [dat95], which can then be queried further as a first class object. There is no notable difference in the manner in which the new relation and the original relations that exist in the database are defined.

- **views** - a virtual table (in a relational database) which does not actually exist but appears to the user as if it does [dat95]. They are defined in terms of existing tables to give specialist representations of the database for a particular application.

- **joins** - these are the operations associated with relational models where tables are linked together by their common attributes to show the relation (or association) between those relations.

- **aggregation** - this is an abstraction concept for building composite objects from their component objects [elm94], using relationships between the aggregate object and their component objects such as *is-part-of*, for example the collection of parts which go together to make an engine.

- **domain** - a pool of values from which an attribute can take its value. For example, the domain for a supplier number attribute would be all legal supplier numbers. All values within a domain are of the same type.

- **rich types** - all models define properties of entities, which give a type for that entity. Newer models such as the object-oriented model allow us to extend the

collection of basic types offered, using these new types as if they were basic types of the system.

- **nested semantics** - this is the ability to nest structures within structures, that is to define types which contain other types.

- **abstract data types (ADTs)** - a data type which contains not only the data but also the collection of functions (or methods in object-oriented terms) which can be applied to that object. **Encapsulation** is a term used where the data is hidden inside the abstract data type with a collection of methods supplied for manipulating (that is viewing, updating) the data.

- **inheritance** - this is a term applied predominantly to object-oriented models, which is a method of providing type hierarchies (similar to the hierarchical data model), by creating subtypes of some supertype which includes all the properties of its supertype and also adds extra properties to classify stronger entities which have that subtype as its type, for example the student is a person example given in the hierarchical definition where a student type would inherit all of the properties of the person type (for example name, age) as well as extra properties such as the course they are on and their student identification number. The subtype is a **specialisation** of the supertype and the supertype is a **generalisation** of the subtype.

- **methods** - the implementation of an operation on the particular information in the database [dat95]. This is a well used term in object-oriented systems, where methods are functions encapsulated within an object, that is a function which operates on that object.

- **extensibility** - a definition which, in the database sense, implies that a database schema is developed modularly so that the schema can be extended easily without having to unload and reload the entire database. Extensible types are types which can be extended easily.

- **declarative** - declarative programming is a style of programming language as used in Prolog and in functional programming languages. In database languages a declarative language would specify what data is to be retrieved rather than how to retrieve the data [elm94].

2.3 The Relational Model

The relational model is based on the concept of relations, which are derived from set theory. The relational model was first devised by Codd [cod70]. The basic concept is the relation (table) which is equivalent to a set in set theory . However, it should be noted that SQL does allow bags to be created as the result of manipulation. Being basically based on set theory, the relational model is formal, and thus data definition and manipulation follow readily as set theoretic constructions.

2.3.1 Structure

As previously stated, the main concept is the relation, this can be described as a table where columns (attributes) in the table specify a particular piece of information about a certain entity (a row). Rows are not held in any particular order, but sorting on keys and indexing operations allow tables to appear ordered. The most recent relational systems allow abstract data types, where the information can be more complex (equivalent to the nested relational model, see later, with functionality encapsulated within the ADT).

Complex views of the data can be visualised through views, where the user has the capability of restructuring information in the database to a more tailored view to match the needs of the application, as well as allowing the representation of the data to be customised to suit the needs of each user.

2.3.2 Rules

There are many rules possible in relational models, due to their strict formal basis, and many tools which have been developed over the years. These rules are:

- Normalisation - a tool to ensure tables are consistent, particularly with respect to update operations. For example, the second normal form rule ensures that any table has atomic values (first normal form) and every nonkey attribute is fully functionally dependent on the primary key. Therefore a table containing the attributes *student number*, *student name*, *module number* and *exam mark*, with *student number* and *module number* as the primary key would not be in second normal form if *student name* depended solely on *student number*.

- Model level constraints, such as, for example, checks on primary key/foreign key matches (referential integrity) and on completeness of key values (entity integrity). An example of a referential integrity rule would be a check that every student in the above table did appear in a base table containing a list of students.

- User level constraints, which are expressed as set theory operations, basically they are defined as mathematical constraints. An example would be defining an SQL query with a GROUP BY [dat95] operation to check that each student takes exactly twelve modules.

2.3.3 Manipulation

Manipulation of sets is well defined in the relational model, with both the relational algebra and calculus providing formal notions for operations on relations. From the relational algebra query languages such as the Peterlee Relational Test Vehicle [tod76, elm89] have grown, and from relational calculus, SQL and QUEL have been developed. SQL has an ANSI standard (version two is the current one and version

three is under development [ans94], with features from the object models added). As well as manipulation languages, a large set of tools such as Query By Example (QBE) [zlo75] and Query By Forms (QBF), have been devised, many providing graphical or form based interfaces to the originally text-based relational database systems. One of the strengths of manipulation in relational languages is that closure in queries is easily attainable because relational query languages always return tables as their result, and thus the results of queries can be queried further still to allow complex queries to be built using a building block approach.

2.3.4 Strengths

The relational model is widely used having built up a large user base with many suppliers since the 1970s. The highly structured and formal basis provides a clear simple approach to modelling data for real world applications. SQL and the other manipulation systems provide a common basis between competing systems (for example Ingres, Oracle), and the entity-relationship tool maps automatically into relational databases. Finally, the basic model facilities have been extended by the large amount of support tools that are available.

2.3.5 Weaknesses

The relational model is set based, so complex data representation is difficult. Joins between data are not difficult to express in the model, but they can not be simply visualised as they are defined by primary-key foreign-key matches between two apparently independent tables. Also the original relational model of Codd is not functionally complete (that is it can not perform all the operations required for it to be classified as a Turing machine). Newer systems add so much functionality to the original model, such as abstract data types which are similar to objects in newer object models, that the usability of systems has declined as these systems have got larger in size.

2.3.6 Extensions to the Relational Model

We will look at two widely known extended relational systems, namely the nested relational model [rot88] and RM/T [cod79].

2.3.6.1 Nested Relational Model

Nested relational models aim to remove the restriction of 1NF (first normal-form, that is attributes are no longer atomic), but this introduces problems of how to perform joins on non-atomic values, that is do we join when all members of a non-atomic set match some other set, or when only some match.

The main structure of a database is a hierarchy, and domains can be nested as powersets, which are very similar to true mathematical relations.

To solve the join problem, two further operations have been added to the relational algebra, *nest* and *un-nest*, as well as a new normal form (partitioned normal form) for ensuring consistency in the data.

2.3.6.2 RM/T

Codd's 'extended-relational model' removes the distinction between entities and relations. There are two main types of relation, *E*-relations, which model the fact that certain entities exist, and *P*-relations which model the properties of those particular entities.

Entities can then be split into three further categories, *kernel* entities which define concepts such as suppliers and parts, *characteristic* entities, which model the fact that

an entity may describe the properties of some other entities, and *associative* entities, which model many-to-many relationships.

As well as these different formal entities, the benefits are that there are a powerful set of integrity rules for ensuring consistency.

One final concept is that sub-types and super-types are easily modelled, in a similar fashion to complex object hierarchies.

2.3.6.3 Object modelling

The entity-relationship model [che76, lay88] can be thought of as a simple database model in itself, although there is no support for manipulation. However, it is a very natural tool for designing relational database applications because its main concepts are the entities, which are equivalent to relations in the relational model, and relationships between these entities, which can show the primary-foreign key matches in the database. Rules can be defined on the functionality and membership class of entities in relationships. The entity-relationship model is actually a form of object structure model but we include it here as it is the main tool used for designing relational databases.

2.4 Functional Models

The functional model was first proposed by Kerschberg and Pacheco [ker76], but the best known example is DAPLEX by Shipman [shi81], so this is the one we concentrate on. As the relational model is based on relations, the functional model is based on functions, giving, as Shipman claims, a more conceptually natural model, that is the function models activity and dependencies as well as the structure of the database.

2.4.1 DAPLEX

'The DAPLEX language is an attempt to provide a database system interface which allows the user to more directly model the way he thinks about the problems he is trying to solve' [shi81].

DAPLEX is a data definition and manipulation language for database systems based on the functional data model. It claims to be a 'conceptually natural' database interface language, being based on two main concepts, the entity and the function.

Entities provide the type definition of an object within the database, for example a Student entity is a Person, where a Person is an ENTITY, an entity being the base type within the system. Within an entity, functions represent values and associations to other entities, where a function can return a single valued set, represented by a single headed arrow ->, or a multi-valued set, represented by ->>. For example:

```
DECLARE name(Student) -> STRING
```

defines a function on a Student which returns his or her name as a character string, whereas:

```
DECLARE courses_completed(Student) ->> Section
```

represents a relationship of a Student entity to the 'many' Section entities which a particular student has completed.

Functions can be of two main sorts, there are the data definition functions represented by the declare statement, and derived data functions, which are used for defining inverses of existing functions, views on the database and aggregation operations, on existing values. For example:

```
DEFINE gradePointAverage(Student) ->
```

```
AVERAGE(grade(Student, Section)
        OVER courses_completed(Student))
```

defines a new derived function for a Student which returns that student's average grade for all courses which he or she has completed.

The derived data also provides a constraint and trigger facility, for instance:

```
DEFINE student(Class) -> INVERSE OF class(Student)

DEFINE TRIGGER overbooked(Class) -> COUNT(student(Class))>45
        SEND MESSAGE(head(dept(Class)), 'Overbooked:',
title(Class))
```

This uses the INVERSE OF operator to define a function to return the name of a Student within a particular Class, and then defines a trigger which operates whenever the number of students in that class is greater than 45, sending a message to the head of the department for that class.

Probably the most notable use of derived functions is that they can provide separate user views of a database. Using derived functions, the view of the database can be altered to suit the taste of the user. Derived functions control the extent of access to the database that the user requires, both for convenience and security. The new user view is defined in a different name space (intension) from the old one, so that access is only given to the new view. The PERFORM .. USING construct is used to define how to perform any updates to derived data, and what that update actually means in terms of the original database, for example the code below would ensure that a query which included an instance of the PERFORM section would include an existing *Course* entity for that student rather than defining a new *Course* entity with that *Title* name.

```
PERFORM
        INCLUDE CourseName(StudentName AS STRING) = Title
USING
        INCLUDE Course(THE Student(StudentName)) =
```

```
THE Course(Title)
```

The final section of the DAPLEX system is its manipulation, which is carried out by the means of function composition, for example

```
FOR EACH Student
      SUCH THAT dname(minor(Student)) = 'Maths'
PRINT name(Student)
```

The manipulation system provides the means for querying, and for update and deletion operations. For example, to change some of the data for the student 'Mary' we could have:

```
FOR THE Student SUCH THAT name(Student) = 'Mary'
BEGIN
      LET minor(Student) = THE Department
            SUCH THAT name(Department) = 'Mathematics'
      EXCLUDE courses_completed(Student) = THE Section
            SUCH THAT sec_no = 1
      INCLUDE courses_completed(Student) =
      {
            THE Section SUCH THAT sec_no = 2,
            THE Section SUCH THAT sec_no = 4
      }
END
```

DAPLEX is a navigational language, all operations are carried out by functions, compared to the subsetting approach of the relational model.

The declaration of the database entities and functions are done at the intensional level of the database, and the values of functions comprise the extensional level of the database. But the intension level is more complex than just described, as a metadata view of the database is provided, whereby the intension of the database can be queried, using the normal manipulation commands. This gives a two level database architecture, represented by the metadata and the data levels.

There has been much research into functional database models in Britain. Kulkarni and Atkinson [kul86, 87] have developed EFDM, the Extended Functional Data Model, which is an extension of DAPLEX including concepts such as a better view facility and graphical navigation and querying, using the language PS-Algol [atk83], which is a variant of the Algol language with persistent extensions. Gray and Paton [gra88] have been looking at the development of the DAPLEX system using Prolog, and have lately extended their work to look at the role of this system for object-oriented databases [gra92]. Poulovassilis and King [pou90] investigated computationally complete functional database models by using lambda calculus expressions to allow more complex persistent functions and data types to be defined. Poulovassilis and Small have recently extended their functional database programming language PFL [sma91] to integrate event condition active rules with a deductive database [pou96].

2.4.2 Structure

There are two main concepts, the entity and the function. Entities represent a unit of information, and are in fact a collection of functions from one entity to a (possibly singleton) set of entities. This simple but well founded structure can lead to an ability to model quite complex data structures.

2.4.3 Rules

Constraints on the types of arrows (that is functions mapping from one set of entities to another set of entities) ensures a very rigid set of normalisation rules, although the data is not necessarily restricted to first normal form (1NF) where every value must be atomic. A function between two value sets A and B (f : A -> B) ensures that A determines B, which is one requirement of second normal form (2NF) if A is a key to B. However, as other aspects of 2NF and 3NF are not enforced, the normalisation

level may be said to only approach $\neg$1NF [jae82], that is where first normal form is bypassed but other normalisation rules can still apply.

2.4.4 Manipulation

The model is based entirely on functions, so operations in the functional model are basically function compositions. This ensures that manipulation in the model is navigational, where we navigate through entities by following functions to find the required result. Derived functions ensure that we have an equivalent capability to object oriented models of seamlessly allowing complex manipulation operations based on computationally complete operations rather than just simple relations.

2.4.5 Strengths

Because queries in the functional model produce new entities containing new functions which act as the source of further functions, we have closure equivalent to that in the relational model with the power of a navigational method of querying the data, which is however much less easy to use than relational query languages.

2.4.6 Weaknesses

Functional models have an overt mathematical basis which may inhibit commercial prospects. Also queries are more difficult to construct and are harder to optimise than in relational systems and there is a shortage of third party tools. Work on recent implementations (see for instance [gra92, pou93, pou96]) has shown the long-term promise of this approach.

2.4.7 P/FDM

P/FDM is an extension to the DAPLEX system developed by the Object Database Group in the University of Aberdeen [gra92]. Basically, it is a semantic data model database that adds object-oriented extensions to DAPLEX, but by being based on Prolog it ensures a complete query language. We discuss P/FDM in chapter six, as it is the implementation base for our Product Model.

2.5 Object-Oriented Models

In this section we discuss the main concepts of object data models, concentrating on object-oriented models. In the next chapter we review in detail the object-relational model.

A simple way to view object-oriented databases is as an extension of the object-oriented programming paradigm to handle persistency. They are an attempt to more closely model the real world by using their structures which can easily model most of the complexities in the real world. Rather than having relations, we now have objects, with direct references (usually by pointers) between them, rather than joins where there is no direct reference and the reference can only be detected by closely examining primary-foreign key matches between tables.

The constructs of object-oriented models are objects (approximating extensions in relational terms), classes (intensions), and types (domains) but some object-oriented systems collapse classes and types into a single concept (unfortunately also known as the class), indeed some people see OODBMS solely as persistent type systems.

2.5.1 Structure

The basic structures are classes, objects, and pointers. Pointers provide the relationship or association abstraction, which can be singular or set based, and can be bi-directional. A number of powerful abstractions including generalisation and aggregation [smi77] provide support for a natural representation of complex data structures and tools for manipulating them. Objects are powerful structures, they support the concepts of aggregation, subtyping, encapsulation, association and inheritance (single and multiple) [mey88]. Objects are very close to real world objects, that is they contain both structure and their associated operations. Classes ensure that all objects are typed, and these types can be more complex than the standard set based types allowable in set theory. These objects can be arranged in a hierarchy, which is an important property in modelling real-world structures and defining complex types.

2.5.2 Rules

Most rules in an object-oriented system are defined through typing operations, that is a value must conform to a particular type or class definition. The object-oriented programming language Eiffel [mey88, mey92] extends the rules concept with post and preconditions which allow users to restrict the values of the supplied objects to a function and the result of a function to particular subsets of its class definition. On the other hand, the programming language C++ [str91] is 'weakly' typed, but rules can be added by restricting types through creation methods.

Object-oriented databases are in principle very strongly typed. Pointers give sets (which can be unary) of objects for relationships between objects, and these are restricted in type. This is similar to functions in the functional data model.

The hierarchical structure of type classes and the ability to specify particular properties of types give integrity close to that which can be defined through normalisation. In particular, a model for object databases has been defined called the ODMG model [cat94] which defines a clear model for object databases and a type hierarchy for values.

2.5.3 Manipulation

Many object-oriented databases are implemented as collections of facilities from either C++, Smalltalk, or similar, libraries. Therefore query languages are based on user implementations using methods, with only a sparse collection of manipulation operations pre-supplied by the system. This enables very powerful operations but requires higher levels of programming skills by the users concerned. A number of object models are trying to incorporate functional query languages, to add the formalism that most of them lack. One such language is discussed in the next chapter. By considering the relationship between entities and functions in the functional model to classes (and objects) and methods in object models, one can see a close similarity between the two models.

The ODMG group have devised a standard for object databases (ODMG-93), but there is no mathematical formalism and the standard is still heavily based on C++ concepts. For example the CORBA Interface Definition Language was influenced by C++ [cat94, pg. 50] and the ODMG object structures map closely to C++ structures. However they have also implemented the object structures in Smalltalk. Some of the major commercial developers are beginning to make their systems conform to some of the ODMG definitions, which is a definite step in the right direction. For future versions to be successful though, it appears that something more formal than the current ODMG model is required to give it the rigour of relational systems.

2.5.4 Strengths

Object-oriented databases have many advantages over relational models. They have a very natural structure, enabling the representation of complex objects, encapsulation, hierarchical structures, inheritance and the other abstractions of object models which we have discussed. They are also very powerful because they have a very close link to the programming languages in which they are implemented, which also means that queries can be very quick. Being very close to object-oriented programming languages also means that there is a huge and ever growing collection of tools that are readily available and simply adaptable to database systems.

2.5.5 Weaknesses

Their advantage of being closely coupled with programming languages also gives them some disadvantages. There are no inherent query languages which makes it difficult for the current relational user base to transfer their skills to object oriented models, although this situation is improving with the OQL query language and SQL3 [ans94], although it is unclear as to what implementations exist and how well developed any implementations are. Interoperable systems are difficult to achieve because object models have no formal basis upon which an interface can be developed to other models.

2.5.6 Object-Oriented Systems

The number of object-oriented database systems is currently growing. Here we provide a short description of some of the more popular systems:

- O_2 [ban92] is a commercial system produced as part of the Altair project, which started in 1986. It is a complete system, developed to try to match the definition in

the Object-Oriented Database System Manifesto [atk90], implemented in C, and programmable in C (CO_2) and Basic amongst others.

- Iris [fis89] is closer to a semantic data model, but its characteristics of objects with specialisation and its schema definitions make it a good candidate to be described as an object oriented database [ber93].

- Orion [kim90] is a system developed by the Advanced Computer Technology Program at MCC, Texas. It aims to address the problems of dynamic schema evolution, query model, automatic query optimisation, secondary indexing, concurrency control, authorisation, multimedia data management, versions, composite objects and notions of private and shared databases. Kim claims it is the first project to deliver foundational papers on many of these topics.

- ObjectStore [lam91] is the most commercially successful object-oriented system. It is based extensively on C++ class libraries, but tools are beginning to be produced for tasks such as schema manipulation.

- POET - a shareware database system, also based entirely on C++ class libraries. Users must be competent C++ programmers however because there are no third party tools for this system. It is however very popular due to it being in the public domain.

- Gemstone is based on the Smalltalk programming system, with a programmatic interface called OPAL [kim90] which extends Smalltalk with data definition and data manipulation capabilities.

2.6 Discussion

The relational model has undoubtedly been the most successful model for the past twenty or so years, taking over from the hierarchical and network data models which came from the CODASYL era. Relational systems are the most used at the moment, although they sometimes lack power for modelling the complex data requirements for aspects such as multimedia. The main reason for their success has been their simple mathematical basis, with gains in provability and rigour, and their basic support which has grown during their existence. Oracle is now much more than a database system, it is a whole business environment.

From the relational model came extended relational systems such as the nested-relational and RM/T, but these never picked up in popularity compared to the standard relational systems. The functional models has also been around for the past fifteen years, but again has never experienced the type of success that the relational model has encountered. It is now being seen as a very powerful model particularly in adding a query language to object-based and semantic data models.

The biggest change over the past few years has been the introduction of the object-oriented paradigm into the database community. Object-oriented databases have proved ideal for handling the types of complex data that relational models find difficult to represent naturally. They have had particular success in CAD and CAE systems, and are also beginning to find their niche in multimedia applications, where complex attributes such as video footage can be simply stored as an object. They are a natural model for representing complex structures and association abstractions, such as inheritance, in the real world.

Their major shortcomings are that they are not easily used by non-expert computer users, and although their data representation is natural, their ability to manipulate this data is still highly dependent on a programming language rather than a data manipulation language. There are also problems in representing atomicity and

functionality in a natural manner as is desirable for instance in object-oriented analysis. Brathwaite [bra93] illustrated the difficulty of mapping methods into a functional analysis technique such as data flow diagrams. In object models functionality can be distributed among a number of objects and aggregation is needed to assemble the activity into a coherent task. Perhaps most importantly, object-oriented databases lack the mathematical formality traditionally associated with database approaches.

The problems with object-oriented models led to the development of the object-relational model which is discussed in detail in the following chapter. This is a new type of extended-relational model, using the concepts of object-oriented models that are required for representing the new complex data, such as encapsulation and inheritance. The object-relational model maintains the sound basis and formalism of relational models as well as ensuring that data manipulation is provided by close relatives to relational algebra and calculus, such as PostQUEL in Postgres and the up and coming SQL3 standard.

2.7 Summary and Conclusions

In this chapter we have looked at data models in general, and have discussed their strengths and weaknesses. The relational model is strong mathematically but lacks the ability to model well the abstractions of the real world, such as inheritance and complex objects. On the other hand, the object-oriented model is strong in representing the abstractions of the real world but is much weaker from a formal point of view. The functional model has a mathematical basis as strong as the relational model, can represent complex objects, and recent extensions show its long term promise. However it has not enjoyed the success of relational and object-oriented systems, perhaps because of problems with its image and usability.

Therefore, models which incorporate object-oriented and relational approaches seem to be a promising route, so in the next chapter we look in detail at the object-relational

model as an indication of the full range of facilities required in future database systems. We do, however, also acknowledge the fundamental appeal of the functional data model and indeed eventually employ this as the basis for our implementation.

3. Object-Relational Models

3.1 Introduction

Object-relational database systems have evolved as a practical solution to the numerous problems that have been associated with the new generation object-oriented database systems. Aspects such as closure, views and query languages have led to developers returning to relational ideas, which encompass these features in a natural and formal manner, but also incorporating features that have grown out of object-oriented database systems, such as inheritance, aggregation and better handling of complex objects.

This chapter will look in particular at the object-relational database system Postgres [row87], a research development from Berkeley which has added object-oriented features to the relational model, while still maintaining the look and feel of an Ingres type database. After discussing Postgres, we will then briefly describe the features of other object-relational systems, such as Montage [ols94], which is a commercial development of the Postgres system, Matisse [mat94], UniSQL [kim94] and OpenODB [hew94].

3.2 The Postgres Database System

3.2.1 Introduction and Aims

The Postgres system is a research development from Berkeley, and is intended to supersede the Ingres [ing94] relational database management system. The intention is

to integrate object-oriented features into a database system, while still maintaining its relational database background.

The main design goals of Postgres are to [sto86a]:

1. provide better support for complex objects;
2. provide user extendibility for data types, operators and access methods;
3. provide facilities for active databases (that is alerters and triggers) and inferencing including forward- and backward-chaining;
4. simplify the DBMS code for crash recovery;
5. produce a design that can take advantage of optical disks, workstations composed of multiple tightly-coupled processors, and custom designed VLSI chips;
6. make as few changes as possible (preferably none) to the relational model.

Postgres is still relationally based, but has the added ability to store complex objects. Complex objects are supported by the ability to define abstract types, and to use these as the type for a column of a relation. Querying is provided via a new query language, known as PostQUEL [rhe90a,b]. This is heavily based on the QUEL [ing94] relational calculus, but with extensions to deal with the object-oriented concepts integrated into Postgres.

We will continue by describing the main concepts of the Postgres data model, and then examine data manipulation, describing both the PostQUEL and C forms of querying.

3.2.2 The Data Model

The Postgres data model is based heavily on the relational data model, with the additional inclusion of classes, inheritance, types and functions [row87]. Whereas in the relational model, the fundamental notion is the relation, or table, in Postgres it is the class: a named collection of instances of objects [rhe90a], used for defining complex types. Tables in Postgres are handled in the same manner as in a relational

database. Classes represent abstract data types (ADTs) in Postgres, a feature which has been incorporated to some extent in more recent relational systems, with varying degrees of success. By treating ADTs as a principle concept, Postgres handles them in a more natural manner, incorporating such concepts as inheritance and user-defined operators and procedures.

The Postgres model allows an ADT class to define the type, or intension [ull88], of a column in a relational table, thereby allowing complex data to be stored in a field of a table. Allowing ADTs to be incorporated in tables provides a method of simulating features from object-oriented and semantic database models, such as aggregation, generalisation, complex objects with shared subobjects, and attributes that can reference tuples in other relations [row87]. In this way, attributes in a table can be user defined types, operators, and programming language functions, or procedures.

A column in a table can be of type procedure, which means that a value in a tuple can be determined by some operation on other values in the tuple, relation or database. Functions in Postgres are very much similar to derived functions in functional databases [shi81], or virtual fields in CODASYL databases.

There are two kinds of type in Postgres, atomic and structured [row87]. Atomic types are those defined as ADTs, whereas structured types define complex data, such as arrays and procedures. Arrays in Postgres can be of undefined size, but they can only store elements of the same type.

Tables can have a key, which may be a composition of attributes. The attributes of the key may include an ADT, so long as a comparison operator is defined for that ADT. Tables also have an unique system identifier, or object identifier (oid), which uniquely references a particular tuple in the database. These object identifiers can be queried by the user, but may not be updated manually.

When defining a new type in Postgres, it is usually the case that its internal size needs to be defined for optimal storage and access. For large objects, or BLOBS, this internal size is not known, but Postgres does support these, by allowing its size to be defined as variable. Large objects can be supported either by using the UNIX file system, to store a large object as a file, or the Inversion file system. This breaks up a large object into smaller chunks, using a B-tree for access, which guarantees crash recovery through protected transactions and provides better performance for accessing objects.

Postgres supports inheritance of tables. By allowing table inheritance, the known problem of relational databases in handling rules such as the 'student is a person' concept in table definitions can be handled naturally, that is in a more object-oriented manner. When inheriting tables, the key is also inherited. Multiple inheritance is also supported, but if two tables share common attributes, multiple inheritance from these tables is not permitted.

In [row87], the major aim of Postgres as a contribution to database development is that inheritance can 'easily' be added to a relational database; easily implying that as few changes as possible are made to the underlying relational model. This can be modelled naturally through the class, or ADT, method. They claim that this shows that the major concepts in an object-oriented database can be cleanly and efficiently supported in an extensible relational DBMS.

Another concept which is a feature found in temporal databases [tan93], but causes implementation problems in relational and object-oriented databases is a feature known as time travel. This is where all past revisions to the database are stored, so a user can see the state that the database was in at a certain time in the past, or even query on attributes between two dates, that is 'list all the students who were taking computing between 1 September 1984 and 31 October 1992'. This means that a database can support versioning and snapshots for multi-user use.

3.2.3 Data Manipulation in Postgres

Postgres data manipulation is provided through its own query language PostQUEL. This is an extension to the QUEL [ing94] relational calculus provided with Ingres, and has been extended to deal with the new features of Postgres such as abstract data types and inheritance. There is also a facility whereby Postgres tables can be manipulated from within C, known as LibPQ, which provides the full functionality of PostQUEL to an external program. Abstract data types, or classes, can also be defined in C, as well as complex procedures, in cases where the PostQUEL language does not provide enough functionality for the task of the procedure.

To allow the Postgres system to be used from within C code, and eventually other programming languages, a feature known as portals [sto86a] is provided. These are like cursors, in the way that they allow a programming language to retrieve data from the database, but it also allows, through the LibPQ system, the whole functionality of the PostQUEL calculus to be used externally.

In principle, PostQUEL supports all of the constructs provided in QUEL, although with some variations in syntax. In addition, the query language should provide the ability to use the new concepts [rhe90a] which have been added to the Postgres system, such as:

- user defined types (or ADTs), which is done by supplying input and output procedures for that ADT;
- query language and programming language functions, so that the query can use derived data;
- user defined operators, which are useful for providing operations such as for testing equality of a user defined type;
- fixed and variable length arrays, whereby it would be useful to look for a particular element in an array, and return all tuples which satisfy the query;
- functions of an instance, where we want the query to apply to all tables which inherit from the table being queried;

- a rules system, so that alerters and triggers can be supported.

In practice, the version of Postgres that we used had some limitations. In particular, concepts such as those given below are not fully implemented in the version 4.1 [rhe90a], which we were using:

- adding attributes to an existing class;
- keys in classes;
- snapshots;
- exceptions;
- merging of two classes;
- removing functions from a class;
- asynchronous portals.

PostQUEL functions can also handle iterative queries as a system supplied function, where the result of the function can be transitively closed. This means, for example, that a query to return a child's ancestors will not only return the parents, but will also return the grandparents, great grandparents and so on, presuming the information can be derived from the table. This allows recursive, or iterative, querying - something that the relational and object-oriented models do not support very well. In a relational system this would need to be implemented within a loop in a host language, and would need to be implemented by the user as a method in an object-oriented database.

The PostQUEL language also supports the notion of 'time travel' in queries. This provides a means of historical queries, as mentioned previously. By allowing time varying data, versions and snapshots of data can be stored and queried, although updates to versions are not carried through to the underlying table. Any changes to the underlying table will result in changes to the version, unless that version was initially created from a snapshot of the underlying table.

Because inheritance of tables is supported, the PostQUEL system must be able to handle the inherited procedures as well. A table which has inherited a procedure from

some parent table must be able to use that procedure on itself, resolving any problems with attributes in the new table, that is an attribute may have changed its type, or may have been overridden. Multiple inheritance must also be supported, for the case where a table inherits two procedures with the same name from two different parents. This is handled by a structure called an IPL (inheritance procedure list), which keeps a hierarchical list of procedure instances, so a query can decide which instance of a procedure should be used for a particular query.

Queries can also explicitly follow an inheritance hierarchy. This is different to procedure inheritance, here the need is for a query to be able to explicitly apply itself to all tables which directly or indirectly inherit from the table that is being queried. For example, for a list of all people, the user could explicitly define the query so that it looked at all students as well. This is something which is not directly supported in relational databases, where if there was to be a student table, then the query would have to be used twice, once for the person table, and then again for the student table.

Rules can also be incorporated into the data manipulation, so that alerters and triggers [sto86a] can be supported. An alerter is the result of a rule applied to a retrieve query, whereas a trigger is the result of a rule applied to an update. There are actually two rules systems, instance level and query rewrite. An instance level rule is one that will only ever apply to a few instances in the table, whereas a query rewrite rule is one that usually applies to most instances. It is best to always define the type of rule for performance measures.

Finally, transaction management is provided, simply by the ability to enclose a range of queries within a *begin ... end* construct. This ensures that in the case of a crash, then any changes that were made to the state of the database within the transaction will be aborted, otherwise the transaction commits.

3.3 Comparison with other models

The Postgres system, and other object-relational type database systems, attempt to amalgamate the functionality of the current models; relational, object-oriented and in some sense, functional. The model is very relational based, but adds the best features of the object-oriented models, such as inheritance and support for complex objects. The main additions to the relational model are support for non-atomic values, and table inheritance.

By incorporating procedures into the query language, the object-relational model can claim to be very close to the functional model. The object-relational model adds a feature which semantic and functional models cannot handle very well. This is the ability to represent data with uncertain structure, for example in a graph some of the nodes may be missing. The object-relational model can aggregate complex data of uncertain structure through standard outer join operations.

Object-oriented models cannot easily deal with objects which have a variety of shared sub-objects, such as overlapping subclasses [emb95]. Postgres claims to be better at this, again objects can be left standalone and can be integrated at run-time through the use of outer join operations. Postgres also claims a performance improvement in representing object-oriented pointer chains as standard relations.

3.4 Experience with Postgres

Postgres has been used in a number of student projects in the Department of Computing Science at Newcastle University. Kim, Nelson and Rossiter [kim94] investigated the use of Postgres for a student administration system. Derived data, such as overall student performance, was an important feature of the work. While the delivered system did meet the user requirement, it was thought that the implementation time was excessive compared to say that for a SQL system because of the diverse

facilities and their complex interaction. Further projects by Holford [hol94] on spatial data and Smith [smi94] on triggers showed that while the apparent capability of the system was high, productivity was low again because of conflicts between different parts of the system.

3.5 Other Object-Relational Database Systems

As previously mentioned, Postgres is a research-based object-relational database system, developed at Berkeley. Other object-relational and object-based database systems have recently evolved, a selection of which we will now briefly discuss.

3.5.1 Montage

Montage[2] [ols94] is the commercial version of Postgres. The main differences to Postgres are that it now uses SQL rather than PostQUEL, provides a more object-oriented view for class definitions and the provision of Data Blades, which provide sets of defined data types and functions.

Firstly, Montage now uses SQL for its query language, but claims to add the following concepts to SQL: [sto94]

1. unique identifiers;
2. user defined types;
3. user defined operators;
4. user defined access methods;
5. complex objects;
6. user defined functions;
7. overloading;

---

[2] Note that Montage is now known as Illustra.

8. dynamic extendibility;

9. inheritance of data and functions;

10. arrays.

The Montage system claims to improve Postgres by adding [ols94] an improved library interface (via data blades), better user defined function support, more security, better support for inheritance, simpler rules by removing the need for an instance level rule system, backup in the case of media crashes, browsers, and a performance speed up of between two and 100 times compared to Postgres. Also, most of the features mentioned in the initial Postgres papers, but which were never implemented because of research constraints, have now been developed.

Montage now provides a more object-oriented support for objects, encapsulating types and functions in a class, so that attributes are now stored in a class, making it look less 'table like'. Composite types are supported, these are like *structs* in C and C++, and the constructed types sets, arrays and pointers (refs) are also supported.

Probably the biggest improvement is the provision of Data Blades, which are libraries of defined data types, with their associated functions, providing bolt-on units to the Montage system. The four current data blades are the foundation data blade, which provides the traditional data types, now numbering over forty; the text data blade, for handling variable length text; the image data blade, for handling rasterised graphic data; and the spatial data blade, for the handling of spatial data.

3.5.2 Matisse

Matisse [mat94] provides an object based approach for application development while trying to incorporate some relational features, such as referential integrity, versioning, triggers and rules. It is very object-oriented based, and was developed to model highly complex business applications. There is support for binary large objects, and C function calls to the API, ensuring that any language can use the API. The most

notable aspect is that it supports automatic inverse relationships, and achieves a high level of schema consistency.

3.5.3 UniSQL/X

The UniSQL [kim94] system, as its name implies, is based on the SQL language. Although a few object-oriented database systems make this claim, the UniSQL/X database can boast that its SQL language is an extension of the full ANSI SQL language, and not a chopped down version.

The extensions that the UniSQL data model provide over relational databases are [kim94] nested relations, where a column in a relation can be another relation; encapsulation; an inheritance hierarchy on relations, in a directed acyclic graph format; and provision of sets as attributes of columns, where a set of values can even be of more than one arbitrary data type. The UniSQL system also supports the core object model defined by the OMG [cat94]. UniSQL also handles persistency of objects. This is different to other object-oriented systems, such as ObjectStore [lam91] where a class has to be defined as either persistent or memory resident, and it is difficult to interchange between the two.

The main extensions that the UniSQL/X language provide over ANSI SQL are for path queries. This means that queries can be over nested classes, can include methods, can return nested objects and can handle sets.

The UniSQL/M system provides for a multi-database level, that is a federation of multiple, distributed, heterogeneous databases. It provides the full functionality of UniSQL/X, and is fully distributed. Databases in UniSQL/M are in fact views of relational UniSQL/X databases.

3.5.4 OpenODB

OpenODB [hew94] is an object-based database system from Hewlett Packard. Whereas most database systems are developed from scratch, OpenODB is actually built on top of a relational database system, Allbase/SQL. This means that the data and code is stored internally in a relational database, and OpenODB keeps all the functionality, consistency and security of the relational database. On top of this, all the required object-oriented features are developed, such as inheritance and complex objects, although this is then all mapped down to relational storage.

The definition and manipulation language of OpenODB is known as OSQL, and is a functionally extended semantic superset of SQL. The OSQL language can be both interpreted and provided as an interface in any language that can make C function calls. Some of the features in the OSQL query language are being adopted in the new SQL3 [ans94] standard which is currently being produced.

3.6 Tabular Comparison of Data Models

We refer the reader back to section 2.2.1 for definitions of the concepts used in the table below.

	Structure					**Rules**			**Manipulation**						**Advanced Concepts**
Model	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Relational	✔	✖	✖	✖	✖	5NF	✔	✖	✔	✖	✔	✔	✔	✖	✖
Nested Relational	✔	✔	✖	✖	✖	¬1NF	✔	✖	✔	✔	✔	✔	✔	✖	✖
RM/T	✔	✖	✖	✔	✔	5NF	✔	✖	✔	✖	✔	✔	✔	✖	✖
Object Relational	✔	✔	✖	✔	✔	¬1NF	✔	✔	✔	✔	✔	✔	✔	✔	✔
Hierarchical	✖	✔	✖	✖	✖	¬1NF	✔	✖	✖	✔	✖	✖	✖	✖	✖

Network	✖	✔	✔	✖	✖	✖	✔	✖	✖	✔	✖	✖	✖	✖	✖
Object-oriented	✖	✔	✔	✔	✔	✖	✔	✔	✖	✔	✔	✖	✖	✔	✔
Functional (DAPLEX)	✖	✔	✔	✔	✖	¬1NF	✔	✖	✖	✔	✔	✖	✖	✔	✖
Extended functional (P/FDM)	✖	✔	✔	✔	✖	¬1NF	✔	✖	✖	✔	✔	✖	✔	✔	✖

1. Relations
2. Trees
3. Networks
4. Inheritance
5. Aggregation
6. Normalisation level
7. Referential Integrity
8. Rich types
9. Algebra/Calculus
10. Nested semantics
11. Declarative Queries
12. Closure
13. Views
14. Methods
15. ADTs

Figure 3-1 - Comparison of Data Models

The table in figure 3-1 attempts to summarise the concepts that the main database models provide, in terms of the differing types of structure, rules, manipulation and advanced facilities that have been required by database systems. A '✔' symbol means that the model supports the particular concept defined, whereas a '✖' symbol implies that the model does not support the concept.

The table shows the promise of object-relational systems in advancing database modelling. The model has strong support for all of the structures defined in other models apart from networks (although pointers provide this in some basic sense). It supports referential integrity, normalisation rules (apart from the over strict first-normal form rule), and rich types. It has excellent support for manipulation, with only its complete support for methods open to question, and it contains the advanced concepts of encapsulation and abstract data types. No other model in the list supports so many database concepts.

3.7 Problems with Object-Relational Systems

The object-relational systems as developed today may offer virtually all the functionality currently required. The problem is that in doing so they have become very large and cumbersome systems, as shown with Postgres, and this affects their usability. The very simple set-theoretic relational model has been extended in complex ways beyond its natural scope and without a comprehensive theoretical basis. Complex objects and abstractions such as inheritance are more naturally represented by graphs than as sets [lev91, pou94]. This will affect the usability of the systems and may make further extensions to handle future user demands more difficult. It is desirable to investigate whether the current object-relational functionality can be achieved in a more fundamentally sound way, by providing a rigorous mathematical basis, from which extensions can in future more readily be made.

In this thesis, as discussed in the next chapter, we look at the use of category theory for providing a sounder basis for advanced modelling, with its emphasis on the arrow and on multi-level constructions.

3.8 Summary

This chapter has confirmed the promise of object-relational systems in advancing database modelling. On a comparison of capabilities, the object-relational model outperforms all other approaches. However, there are problems with their usability and with their future extendibility, because the object-relational models are still based on set theory due to their relational framework. This framework is not ideal for the representation of complex objects and their associated mappings and abstractions. To this end, newer formalisms such as category theory may be needed. Multi-level mathematical architectures appear ideal for the complexity required for modelling

object database structures and applications. We discuss this further in the next chapter.

4. Category Theory

4.1 Introduction

In the previous chapter it was shown that the object-relational database model had a number of advantages over other advanced models with respect to its handling of complex data structures while retaining the beneficial properties from relational and functional models. However, there is no generally accepted formal basis for the model. The sophisticated modelling requirements may be better satisfied with higher-order logic [bee92], an example of which is category theory.

In this chapter we outline our main motivations for choosing category theory as the formal basis for our object-relational database model, and then we will provide a simple introduction to category theory, explaining some of the concepts which we use in the Product Model, which is introduced in the next chapter.

4.2 Category Theory

In this section we provide an introduction to the main concepts of category theory that are used in our formal definition of the model in the next chapter. Rather than providing a general introduction to category theory, we refer to either Barr and Wells [bar90] or Simmonds [sim90] for an in-depth treatment of the subject.

4.2.1 Categories

The fundamental construction in category theory is the *arrow*, or *morphism*. An arrow can be thought of as a function in set theoretic terms, providing a mapping from a

source to a target entity, where the source and target entities appear in some domain and codomain respectively.

As in sets, arrows can have an inverse, be an identity arrow, and are composable. We define the basic categorical axioms in detail:

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

$$1_A : A \to A$$

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

3. Identity arrows may be composed with other arrows:

$$\text{If } f : A \to B, \text{ then } f \circ 1_A = 1_B \circ f = f$$

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

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

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

The *category* is a collection of arrows between named entities, or objects. For example, the category[3] **C** has two arrows f and g, where:

$$f : A \to B \quad g : C \to D$$

[3] Categories are denoted in bold throughout this thesis.

An object in a category where there is precisely one arrow from it to every other object is termed an *initial object.* Dually, the *terminal object* is where there is only one arrow to it from every other object in the category.

We use diagrams for displaying arrow composition within categories. *Commuting diagrams* within categories exist where there are two distinct paths between two objects, for example

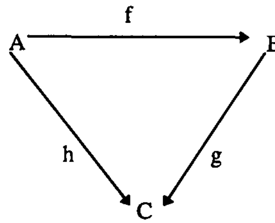


Figure 4-1 - Commuting Triangle

Thus, the diagram in Figure 4-1 is said to commute when the equation $g \circ f = h$ holds, that is the two paths from object A to object C give the same result.

4.2.2 Subcategories

A *subcategory* **E** of a category **D** is one where all of the objects and arrows of **E** can be found in **D**, the sources and targets of arrows in **E** are the same as those in **D**, the identity arrows in **E** are as in **D**, and composition rules for arrows in **E** are as in **D**, that is

$$\text{obj}\mathbf{E} \subseteq \text{obj}\mathbf{D} \wedge \text{Hom}\mathbf{E}(p,q) \subseteq \text{Hom}\mathbf{D}(p,q) \quad (p,q \in \text{obj}\mathbf{D})$$

where, objD is the set of objects in the category **D**, objE is the set of objects in the category **E**, Hom**D**(p, q) is the collection of arrows in the category **D** between objects p and q, and Hom**E**(p, q) is the collection of arrows in category **E** between objects p and q.

There are two special cases of subcategories. When **E** contains all the arrows of **D**, then it is termed a *full* subcategory of **D**, and if **E** has all the objects of **D**, then it is a *wide* subcategory. Any category is a full wide subcategory of itself.

4.2.3 Functors

Functors are arrows providing a mapping between categories, in effect they are arrows between categorical objects. For example, the functor $F : \mathbf{C} \rightarrow \mathbf{D}$ assigns arrows in category **C** to arrows in category **D**, and preserves compositions that are in the source category within the target category.

If the shape of the source and target category is the same, the target category is said to have the *shape* of the functor, that is there is a homomorphism between the source and target categories. A functor which loses information, for example one mapping entities and arrows in the source category to just entities in the target category is termed a *forgetful functor*. Otherwise, a functor which maps all the structure of a source category onto a target category, where the target category may have additional structure itself, is termed a *free functor*.

4.2.4 Typing

Elements a in a set A can be represented categorically by $a : \mathbf{1} \rightarrow A$, where $\mathbf{1}$ is the single category of one discrete object. This is represented in set theory as $a \in A$ Typing is added by indicating the category from where the item is taken, so the arrow $a : \mathbf{1}_{\mathrm{C}} \rightarrow A$ makes the element a in set A of type **C**. Typing can also be based on

arrows, where, for example, monic arrows (that is an arrow which is a monomorphism [bar90], which is equivalent to an injective arrow in set theory) in a category or object could be considered of type **M**, where **M** is a category representing the universe of monics (that is the universe of monic arrows).

4.2.5 Product Cones

We introduce the two main types of product cone defined in category theory, the product and projection, with their specialised versions pullbacks and pushouts.

4.2.5.1 Product and Projection

The product and projection operations of relational algebra can be represented categorically by cones [bar90]. The cone in Figure 4-2 shows the product $A \times C$, with π_l and π_r projection arrows.

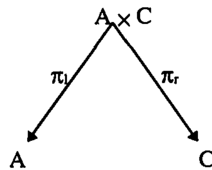


Figure 4-2 - Product Cone

Figure 4-3 below shows an alternative representation, where the diagram now commutes. For any object V and arrows $q_1 : V \rightarrow A$ and $q_2 : V \rightarrow C$, there is a product U with projections A and C such that the diagram commutes for the two equations $\pi_l \circ q = q_1$ and $\pi_r \circ q = q_2$.

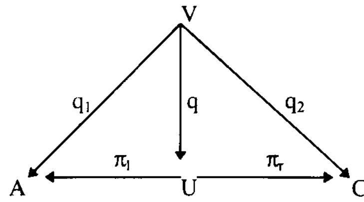


Figure 4-3 - Commuting Product Cone

4.2.5.2 Coproduct and Inclusion

The *coproduct* is the dual of the product where the coproduct is the disjoint union $A + C$, and i_l and i_r are inclusion arrows. In a dual the direction of all arrows is reversed.

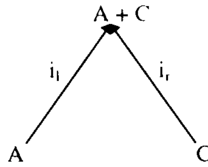


Figure 4-4 - Coproduct Cone

Note that both product and coproduct need not be binary. The concept can quite simply be extended to n-ary or finite products.

4.2.5.3 Pullbacks

A *pullback* is a product restricted over some object. For example, the subproduct $A \times_B C$ over an object B is:

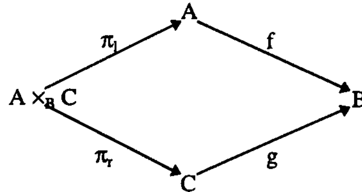


Figure 4-5 - Pullback

where $f(a) = g(c)$, and $f(a), g(c) \in B, a \in A, c \in C$. If this diagram commutes then $f \circ \pi_l = g \circ \pi_r$. Pullbacks correspond to relations in set theory, with $A \times_B C$ being thought of as the relationship of A and C in the context of B (for example the relationship of suppliers and parts in the context of orders).

4.2.5.4 Pushouts

The *pushout* is the dual of the pullback, and is a restricted coproduct. For the diagram in Figure 4-6 below, S is the disjoint union of A and C, restricted over the object B.

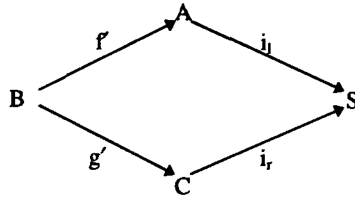


Figure 4-6 - Pushout

4.2.6 Limits

A *limit* in its simplest terms can be thought of as a supremum or infimum. An example of a limit is a terminal object of a family of cones [bar90], the limit being the infimum for all product cones, where the limit precedes each commuting cone, and only exists if every cone in the family of product cones commutes.

The dual of a limit is the *colimit*, which can be considered the initial object of a family of coproduct cones.

4.2.7 Natural Transformations

The natural transformation is a (isomorphic) mapping between functors. It is an arrow between functorial objects. That is, if we have a functor F : $\mathbf{A} \rightarrow \mathbf{B}$ and a functor G : $\mathbf{C} \rightarrow \mathbf{D}$, then a natural transformation σ would map from the functor F to G:

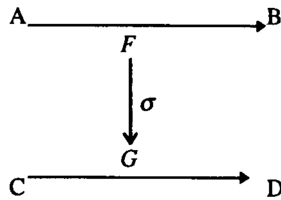


Figure 4-7 - Natural Transformation

For the mapping to be meaningful, F and G must be of the same type as there is no advantage to be gained from comparing the mapping between functors, with either categorical domains A and C or codomains B and D, based on different type systems.

In terms of a commuting target square a natural transformation could be represented as given in the diagram below, that is four commuting functors where F and G are the original functors and S and T are the new functors which ensure that the source and target categories of the functors are correctly mapped.

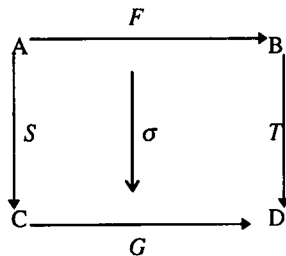


Figure 4-8 - Natural Transformation as a Commuting Product Cone

The diagram explains why natural transformations are isomorphic, that is the mapping σ maps between two functors which have the same shape.

4.3 Why Category Theory

Our motivations for using category theory can be defined as follows:

- The multi-level architecture of category theory reduces the complexity of modelling multi-level database schema and mappings, in that natural transformations and functors provide a much clearer method of mapping complex structures, compared to the method required to model multi-level structures in set theory where all mappings are basically at one level. Natural transformations can not easily be modelled in set theory. The clarity of these mappings facilitates integration of heterogeneous systems because we have the capability for mapping between complex structures.

- The arrows give natural modelling of dynamic (such as methods, queries) as well as static (such as attributes, objects, relationships) aspects of the database, with the added benefit that dynamic and static arrows have no distinction in a category (similar to derived functions in the functional data model), that is it is only the type and definition of the arrow that determines what it actually represents, while in set theory, two basic concepts of the element and function are involved, in category theory only one concept - the arrow - is employed.

- The diagrammatical tools (that is diagram chasing) and consistency tests are vital additions to any data model in ensuring that the database is consistent. We can employ diagram chasing for enforcing constraints such as the normalisation and integrity tests in relational databases, which are both powerful tools in their area.

- Category theory is used increasingly for expressing programming semantics. In particular, categorical type systems are becoming more frequent in functional programming languages, for example the use of monads as advanced type structures with state, in particular for I/O systems [gor95, pey93].

4.4 Previous Work

Previous research into category theory has highlighted its relevance in many areas of computer science:

Goguen [gog89] has produced a categorical manifesto which demonstrated the relevance of category theory in all aspects of computing science, where he highlighted uses of the major five concepts: categories, functors, natural transformations, adjoints, and co-limits. He shows their relevance in computer science, giving guidelines on how the various concepts could be applied. His claims are that category theory would interest computer scientists because of being able to formulate definitions and theories, carrying out proofs, discovering and exploiting relations with other fields, formulating conjectures and research directions, and dealing with abstractions and representation independence.

Cadish and Diskin [cad96, dis93] have already illustrated the effectiveness of using category theory for modelling relational databases. They used the categorical concept of sketches, a form of semantic diagrams; for example the entity-relationship model can be regarded as a sketch. They believe that category theory has many advantages for modelling databases, such as:

- its graph based nature giving a form of evidence,
- the algebraic nature providing ease of manipulation,
- rigour from its mathematical background,
- universality from its polymorphic nature,
- the capability to model dynamic structures and concurrency.

They claim that the database field needs a unified data definition/manipulation language with the advantages given above, summarised as an algebraic graph-oriented formal language for specifying structure and dynamics of the world.

Rydeheard [ryd88, den93] developed categorical data types in the functional language ML. Early work by Rydeheard looked at adding categorical types to standard ML, whereas later work with Dennis-Jones looked in more detail at producing a categorical ML language, where the general structures of ML have been replaced by specific categorical structures (categories, functors, natural transformation and adjunctions). One point they make is that the polymorphic type system of ML even with modules is too restrictive to express categorical concepts in their full generality. Poigné [poi92] confirms the difficulty of expressing categorical constructs in a functional programming language but does cite one possible way forward of a categorical abstract machine as described by Cosineau et al [cos87].

Duponcheel [dup94] has also investigated categorical types in a functional language, Gofer. Gofer is a simplified version of Haskell, one of the main differences is that it does not use the module system. It is more advanced that ML in that one of its main concepts is classes, where data types and operations can be encapsulated, and can be defined to handle polymorphic types. Note that both of these systems appear to be restricted to cartesian closed categories, a type of category which may be powerful enough for most areas of computing, but we believe is too restrictive for representing objects in a database, where the categories are of a general POSET shape (that is the arrows in the category form a partially ordered set). A category is cartesian closed if it has a terminal object, finite products, and for each pair of objects A and B an exponential A^B representing all the arrows from B to A (equivalent to currying in functional programming).

Ghelli [ghe90] looked at modelling features of object-oriented languages in second order functional languages with subtypes. By using second order calculus (which extends on the kind of polymorphism found in languages such as ML) in the functional language Fun [car85], they show how higher order models are appropriate for modelling object-oriented models, in particular they expand on Cardelli's work on multiple inheritance [car84]. He states that object-oriented languages lack a formal foundation, but by embedding the basic mechanisms (object identity and state, late

binding, class encapsulation, inheritance) in a strongly typed functional language then the mechanisms have a mathematical semantics and a set of strong type rules. They show that existential types are not powerful enough, particularly in using sub-typing for inheritance, to model object hierarchies, because types are bound at run-time.

Wolfengangen [wol95] produced a review of categorical methods for object-oriented databases, but no implementation exists. Ehrich [ehr87] investigated a means of introducing coproducts for representing aggregation. Cartmell [car85b] formalised the network and hierarchical data models in one of the first applications of categorical logic to databases. Vickers [vic91] explored the relationship, expressed in observational logic, between the data definition and data values in database concepts.

Lehalli and Spyratos [leh91, 92] used the graph form of category theory for complex object structures, relational model, functional dependencies and some features from object-oriented systems. They show the potential for consistent categorical modelling in terms of functional dependencies by employing limits, but their formalisms are based more on graph structures rather than more complex categorical concepts, and they have not looked into more detailed object-oriented concepts such as normal forms, the association abstraction and querying and views [ros95]. Also, at the time of writing, no implementation exists.

There are also categorical algebraic specification languages: OBJ is a general programmable type system from Oxford [gog92] which can be used for expressing type systems, and is executable [woo88]. OBJ is first order, although OBJ3 includes the concepts of objects, of theories which define properties of some object which may be satisfied by some other object, and of views which bind entities declared in some theory to entities in another module. Theories are found in no other implemented language, although standard ML has been influenced by this approach. OBJ has been used at Oxford to implement a combined object-oriented and functional language FOOPS [gog86].

Clear is a specification language used for program correctness proofs, using theories and sorts for proving algebraic constructions. Burstall [bur80] has defined semantics for Clear based on category theory, using the principles of category, functors, co-limits and pushouts.

4.5 Conclusions

The published work on representing databases in category theory is still at an immature stage. In particular, approaches have tended to be piecemeal with only particular facets being supported, and implementations have been virtually non-existent. Therefore, in the present work we intend to produce a much more complete database model, taking the facilities of the object-relational database model as the target. This formalism should underpin the object-relational model in a more satisfactory way than if set theory only were used.

The other major stage of the work is to make a preliminary investigation of the feasibility of implementing a categorical representation. An important aim here would be to investigate the varying merits and drawbacks of different programming paradigms in representing categorical concepts.

Over the following chapters we will define this formalism for our model, looking in particular at representing entities and relationships in a natural manner and using the multi-level formalisms for providing querying and closure in a closely coupled manner. We then develop a prototype implementation of this model to demonstrate the feasibility of implementing categorical representations and perform some initial tests.

5. The Product Model

5.1 Introduction

In the previous chapter we provided an introduction to the constructs of category theory which we now use in this chapter to define our formalism for object-relational databases. In particular, we will look at the representation of all the abstractions that we have discussed as being part of the object-relational model, and we will show how these can be modelled in category theory

A table at the end of this chapter summarises all of the concepts and symbols that are used throughout this chapter.

5.2 Objectives

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:

- A clear separation between intension (class) and extension (object) structures with a rigorous mapping defined between them.

- Object encapsulation.

- A single unified 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.

- Constraints on class structures as represented by the concept of primary and candidate keys, normal forms such as BCNF (Boyce-Codd Normal Form) and cardinality and membership class (that is whether the participation of an attribute in the relationship is mandatory or optional) in object (E-R) models.

- The standard information system abstractions formulated in the1970s [smi77] and which are prime targets of current object-oriented databases [atk90] and object-relational systems [sto86b, sto94]. These abstractions include inheritance (generalisation and specialisation); composition such as aggregation; classification and association.

- Message passing facilities between methods located in any part of the system.

- 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 with other such structures already existing in the database.

- A multilevel architecture like that in the ANSI/SPARC standard [tsi78] with definitions of views, global schemata and their internal structures and the mapping between them.

5.3 Classes

5.3.1 Basic Structures

The class construction represents the intension of a database. Each class is represented by a category ($\mathbf{CLS}_i \mid 1 \leq i \leq c$) where c is the number of classes in the database. The class name is the name of the category. Category theory keeps distinct intensional and

extensional forms of a data dictionary. For example, $\mathbf{1}_{\mathrm{CLS}}$ (that is the identity arrow of the category **CLS**) types a database entity in general, $\mathbf{1}_{\mathrm{CLS1}}$ types the database entity *suppliers* and $\mathbf{1}_{\mathrm{CLS2}}$ types the database entity *parts*. Then $\mathbf{CLS_1}$ is the class of *suppliers* and $\mathbf{CLS_2}$ is the class of *parts*.

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

Each arrow has a domain and a codomain. Within the universal category **SET**, domains and codomains are sets but in general they can be of arbitrary complexity. Domain and codomain names are the names of variables defined within the class. In the next section, we describe the identification of one or more domains as candidate keys and the selection of one of these as the primary key. The types of arrows, domains and codomains are defined by naming the categories upon which their data types are based. All arrow constructions as regards composition and association must conform to the four axioms of category theory as given earlier [bar90].

Formally, each category $\mathbf{CLS_i}$ is a collection of k arrows or morphisms $F = \{f_j \mid 1 \leq j \leq k\}$ where f_j has domain dom(f_j) and codomain cod(f_j). The domain and codomain names are not necessarily distinct. $\{\mathrm{dom}(f_j) \cup \mathrm{cod}(f_j) \mid 1 \leq j \leq k\}$ is the set of variables in the class which we call V with cardinality q. In order to permit complex actions and dependencies, domains may be structured, that is contain more than one variable. For database applications, codomains are normally considered to comprise a single variable although category theory itself need not be restricted to minimal covers [fre90] but can cope well with open covers [mac91]. Minimal covers are where the right hand sides of functional dependencies are restricted to single attributes, whereas in open covers this restriction does not apply. Variables may be either *persistent variables* given by a set $A = \{a_j \mid 1 \leq j \leq n\}$ comprising the persistent component of the class, or *memory*

variables given by a set $U = \{u_j \mid 0 \leq j \leq n'\}$ comprising the transient component of the class. Note that A and U are both subobjects of V and $n + n' = q$.

Using an alternative notation given in the previous chapter, V corresponds to obj_{CLSi} and F to $Hom_{CLSi}(v,v')$ for all $v,v' \in V$.

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

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

Functional dependencies can be composed. Thus the composition of d_1: $\{a\} \longrightarrow \{b\}$ and $d_2 : \{b\} \longrightarrow \{c\}$ gives $d_2 \circ d_1 : \{a\} \longrightarrow \{c\}$. Such compositions are represented without difficulty in the partially-ordered structures that we introduce later as a natural consequence of the transitivity rule (if $\{a\} \leq \{b\}$ and $\{b\} \leq \{c\}$, then $\{a\} \leq \{c\}$). However, in some circumstances, partial composition occurs, giving rise to a collection of pseudotransitivity arrows [ull88] $P = \{p_i : x \longrightarrow \{y\}\}$ $(x \in \wp A, y \in A, 0 \leq i \leq r'')$.

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

For each arrow that is a method, $m_i : x \longrightarrow y$ $(0 \leq i \leq s)$, then $x \in \wp V$, $y \in V$, that is the domain may be any subobject of the persistent and memory variables and the codomain is a singleton persistent or memory variable. If required, memory variables can be considered as derived [shi81] or virtual variables which can be manipulated by database operations.

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

5.3.2 Identifiers

As we shall see later, we need a way of deriving identifiers for use in our relationship representations. Identifiers can be natural (primary keys) or system assigned (object identifiers). Both the forms of identifiers are initial objects in categories as there is an arrow from the identifier to every other object in the category. Initial objects are normally denoted by 0 in category theory - hence we adopt K_0 as the notation for the key. The key K_0 is derived as shown below for each class category **CLS** [ros93] following a lattice approach [dem92] rather than an algorithmic one [ull88]. However, the work in [ros93] has been extended to deal with pseudotransitivity arrows (see point 4 below) as illustrated below. The lattice formalism lends itself more to a categorical approach with its emphasis on poset constructions. We employ the identifiers and dependencies to test whether our class structures correspond to BCNF (Boyce-Codd Normal Form). This normal form is adopted because it is more powerful than 3NF and can easily be deduced from functional dependencies making it ideally suited to a lattice approach. The procedure is as follows:

1. Generate, from the collection of persistent variables A, the poset category (that is the category is of a partial ordered shape) **PRJ** with elements $p,q \in \wp A$ and projected orderings $(p \times q \leq \pi_l(p \times q); p \times q \leq \pi_r(p \times q))$ as the arrows, that is to take the projections by applying the free functor $G : A \longrightarrow$ **PRJ**.
2. Generate, from the collection of persistent variables in the functional dependencies E, the poset category **DEP** with elements $p,q \in E$ and arrows $\{d_i \mid 0 \leq i \leq r'\}$ as the orderings, that is to apply the free functor $G' : E \longrightarrow$ **DEP**.
3. Generate, from the collection of persistent variables in the functional dependencies E, the poset category **PSU** with elements $p,q \in E'$ and arrows $\{p_i \mid 0 \leq i \leq r''\}$ that is to apply the free functor $G'' : E' \longrightarrow$ **PSU**.
4. Take **DEP** and **PSU** representing respectively the non-trivial functional dependency arrows declared in the previous section and the pseudotransitivity arrows (dependencies inferred from the postulated functional dependencies and their combinations [ull88]) between $p,q \in \wp A$. Inject these into **PRJ**, that is add the arrows of **DEP** and **PSU** to those already in **PRJ** as shown in the diagram below:

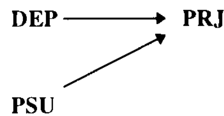


Figure 5-1 - Injection of Trivial and Non-trivial Functional Dependencies into PRJ

5. Test that **PRJ** is still a poset by checking for anti-symmetry (if $p \leq q$ and $p \geq q$, then $p = q$). Cycles in the ordering would give a *preset*[4] (pre-ordered set) which would need to be partitioned by applying a suitable quotient functor to produce a number of posets which can then be handled collectively. Each **PRJ** as a poset E^+ corresponds to the definition of F^+ by Ullman [ull88], the closure of the set of functional dependencies E for the set of attributes A. Each class (record-type) has its own E^+.
6. The infimum or meet of the elements of A in **PRJ** ($\wedge A$) is the primary key PK. If there is no infimum, the set of maximal lower bounds is the set of candidate keys CK.
7. The class is in BCNF if each source of a functional dependency arrow is PK or is a member of CK.
8. The identifier K_0 is either PK or a user-selection from CK. When it is necessary to distinguish the keys for each class, consider $K_0{}^i$ as the identifier for the ith class $\mathbf{CLS}_i$.
9. Other persistent attributes may be labelled $K_1 \ldots K_r$ where $r = n - c$ with c as the number of attributes in the key. In the simplest situations, $r = r'$, where r' is the cardinality of the set of dependencies D but in many cases such as classes with no dependencies or with multiple candidate keys or with classes that are not in BCNF, this will not be true.

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

Our final task is to transfer our results from **PRJ** into the class category **CLS**. This is necessary as, particularly if the key is composite, K_0 is not guaranteed to be a variable in the class **CLS**. We apply an injective functor from a view of the poset **PRJ** into

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

CLS. The category that we inject into C is the exponential construction $\mathbf{PRJ}^{K0}$ (that is all of 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 $n \leftarrow n+1$ and $k \leftarrow k+r$.

5.4 Relationships

The association abstraction between classes is represented in object models by notation based on the Entity-Relationship [che76] (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 5-2, 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.

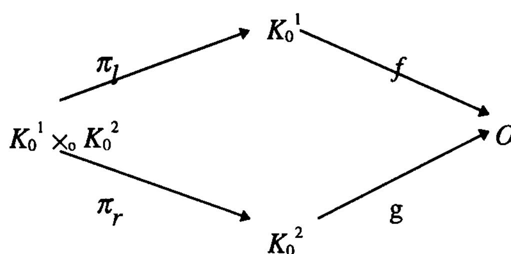


Figure 5-2 - Diagram of Pullback of K_0^1 and K_0^2 over *O*

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 cardinality 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 **CLS**$_1$.
- K_0^2 is the identifier for the *parts* class **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 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 **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. In set theoretic terms, it is the relationship between suppliers and parts over orders.

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.

5.4.1 Enhancements

So far we have considered binary relations (relationships between two entity-types) and have neglected n-ary and involuted relationships (a relationship between two entities of the same entity-type, for example an entity-type *part* is connected to many other *parts*), 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 subproducts [ros92]. 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.

5.4.2 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 5-2 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.

5.4.3 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 5-3 which yields a disjoint union of two or more objects. Consider:

- a category $\mathbf{CLS}_3$ (employers) with the set of arrows $\mathrm{Hom}_{\mathrm{CLS3}}\, p,q$ between objects p,q and set of domains and codomains $\mathrm{obj}_{\mathrm{CLS3}}$; and

- a category $\mathbf{CLS}_4$ (managers) with the set of arrows $\mathrm{Hom}_{\mathrm{CLS4}}\, p,q$ and the set of domains and codomains $\mathrm{obj}_{\mathrm{CLS4}}$.

The coproduct $\mathbf{CLS}_3$ + $\mathbf{CLS}_4$ is the disjoint union of the arrows ($\mathrm{Hom}_{\mathrm{CLS3}}\, p,q$ + $\mathrm{Hom}_{\mathrm{CLS4}}\, p,q$) and the domains and codomains ($\mathrm{obj}_{\mathrm{CLS3}}$ + $\mathrm{obj}_{\mathrm{CLS4}}$).

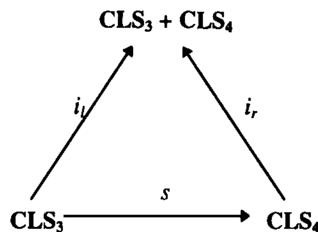


Figure 5-3 - Coproduct Cone for Objects $\mathbf{CLS}_3$ and $\mathbf{CLS}_4$

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 1_{CLS4} of the subclass category:

$$s : K_0^3 \longrightarrow 1_{CLS4}$$

Since a coproduct can, in turn, be the base of another cone, it is a simple matter to construct inheritance hierarchies [nel94]. 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 [bar90] 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 [car84].

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 ($\mathrm{Hom}_{CLS3}\, p,q$) are also available automatically in the set ($\mathrm{Hom}_{CLS3}\, p,q$ + $\mathrm{Hom}_{CLS4}\, p,q$).

5.4.4 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 [kup93] 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. Our technique follows closely that of Ehrich [ehr87] mentioned in the previous chapter.

5.5 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.

5.6 Objects

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

There is a mapping:

$$V_i : \mathbf{CLS}_i \rightarrow \mathbf{OBJ}_i$$

from each class $\mathbf{CLS}_i$ to the instances for each object-type $\mathbf{OBJ}_i$. This 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 dom(f) in $\mathbf{CLS}_i$ to a set of instances $V_i(\mathrm{dom}(f))$ in $\mathbf{OBJ}_i$, each codomain 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 $(K_j \mid 1 \leq j \leq r)$ to a set of instances $V_i(K_j)$ and each functional dependence $(d_j \mid 1 \leq j \leq r)$ to a set of arrow instances $V_i(d_j)$. All assignments by the functor V_i are of values for arrows, domains and codomains.

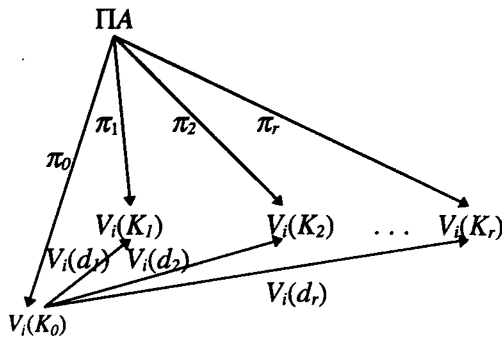


Figure 5-4 - Cone for extension ΠA in the category OBJ

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 5-4 should commute for every cone where ΠA is the product of $(V_i(K_0) \times V_i(K_1)\ \ldots \times V_i(K_r))$, the projection co-ordinate from ΠA is $(\pi_j \mid 0 \leq j \leq r)$, and $\{V_i(d_j) : V_i(K_0) \longrightarrow V_i(K_j) \mid 1 \leq j \leq r\}$ are the postulated functional dependencies. The commuting requirement is for all $V_i(K_j)$ where $(1 \leq j \leq r)$ it is true that $V_i(d_j) \circ \pi_0 = \pi_j$.

Informally, the diagram in figure 5-4 shows that there is a collection of commuting triangles, each with common nodes of the product of the attributes in the intension ΠA and the set of instances of the key $V_i(K_0)$. The variable node in each case is the set of values for a non-key attribute $V_i(K_j)$. There are two paths to each set of values for a non-key attribute – one directly from the intension and the other indirectly via the key values. These paths should return the same results if non-key values are indeed determined by the key values. Therefore for each non-key attribute in turn we compare the equations $V_i(d_j) = \pi_j$. If this equations is satisfied in every case then the extension is consistent with respect to the constraints in the intension.

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 ΠA.

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 5-5 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:

- $\{v_i\colon \mathbf{1}_{\mathbf{TYPi}} \longrightarrow V_i\}$, representing the constraint that each instance v_i of an object V_i $(1 \le i \le q)$ is found in the category $\mathbf{TYP}_i$.

- $\{f_i\colon \mathbf{1}_{\mathbf{TYPi}} \longrightarrow F_i\}$, representing the constraint that each instance f_i of an arrow F_i $(1 \le i \le k)$ is found in the category $\mathbf{TYP}_i$.

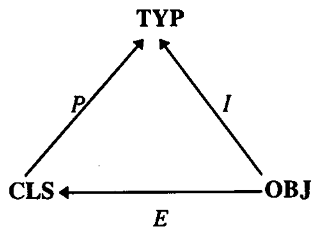


Figure 5-5 - Commuting Diagram for Consistency of Objects with Classes and Types

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.

5.7 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.

5.8 Physical Storage Structures

In a similar way to the mapping between classes and objects, it is straight-forward to define mappings as functors between categories for objects and categories representing disk structures, say, hash tables or indexes. In earlier work Rossiter and Heather [ros92] considered the various approaches to hashing in categorical terms.

5.9 Families of Categories

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

- The category **INT** representing the intension as a family of *c* classes **CLS**, *p* association definitions **ASS** and *g* coproducts **UNI** representing inheritance.

- The category **EXT** representing the extension as a family of *c* objects **OBJ** and *p* association instances **LNK**.

- The functor *D* mapping from category **INT** to category **EXT**. This functor is called *D* (for database) because this is effectively the purpose of a database management system.

Between any two intension categories $\mathbf{INT}_i$ and $\mathbf{INT}_j$ (not necessarily distinct), *m* message passing routes (see later) can be defined using arrows of the form η described earlier between the corresponding arrow categories [bar90] $\mathbf{INT}_i^{\rightarrow}$ and $\mathbf{INT}_j^{\rightarrow}$ respectively. An arrow category is one where, for example, the objects of the category $\mathbf{INT}_i^{\rightarrow}$ are the arrows of the category $\mathbf{INT}_i$.

5.10 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 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 [bar90] 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:

$$obj_{INTj} \subseteq obj_{INTi} \wedge Hom_{INTj}(p,q) \subseteq Hom_{INTi}(p,q) \quad (\forall p,q \in obj_{INTj})$$

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 [shi81], 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.

5.10.1 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:

- **INT$_1$** for the class **CLS$_1$** for suppliers: identifier $K^1{}_0$ (supplier number)

arrows:

$f_1 : K^1{}_0 \longrightarrow sname$

$f_2 : K^1{}_0 \longrightarrow saddress$

$f_3 : K^1{}_0 \longrightarrow no.shares$

$f_4 : K^1{}_0 \longrightarrow share.price$

$f_5 : \ (no.shares \times \ share.price) \longrightarrow capitalization$

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

- **INT$_2$** for the class **CLS$_2$** for parts: identifier $K^2{}_0$ (part number)

arrows:

$f_6 : K^2{}_0 \longrightarrow pname$

$f_7 : K^2{}_0 \longrightarrow size$

$f_8 : K^2{}_0 \longrightarrow weight$

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

- for the pullback $\mathbf{ASS}_1$ of suppliers and parts over orders as in Figure 5-2: identifier $K_0^1 \times_O K_0^2$ (subproduct of supplier and part numbers over orders)

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

- $\mathbf{INT}_4$ for the class $\mathbf{CLS}_3$ for electrical parts - a specialisation of parts with object identifier $1_{\mathbf{INT4}}$ as the identity functor on $\mathbf{INT}_4$

arrows:

$f_9 : 1_{\mathbf{INT4}} \longrightarrow voltage$

$f_{10} : 1_{\mathbf{INT4}} \longrightarrow capacity$

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^2_0

arrows:

$f_6, \ldots, f_8$ from **INT**$_2$

f_9, f_{10} from **INT**$_4$

$s_1 : K^2_0 \longrightarrow 1_{\mathbf{INT4}}$

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

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

1. X_1 : **INT**$_6 \longrightarrow$ **INT**$_5$

(Hom-set in **INT**$_6 = f_9, s_1$; subobjects in **INT**$_6$ = (K_0^2, $1_{\mathbf{INT4}}$, *voltage* | *voltage = 90*));

2. X_2 : **INT**$_7 \longrightarrow$ **INT**$_3$

(Hom-set in **INT**$_7 = \pi_1$; subobjects in **INT**$_7$ = ($K_0^1 \times_O K_0^2, K_0^1$ | $K_0^2 \in$ **INT**$_6$));

3. X_3 : **INT**$_8 \longrightarrow$ **INT**$_7$

(Hom-set in **INT**$_8$ = { }; subobject in **INT**$_8 = K_0^1$);

4. X_4 : **INT**$_9 \longrightarrow$ **INT**$_1$

(Hom-set in **INT**$_9 = f_1, f_3, f_4, f_5$; subobjects in **INT**$_9$ = (K_0^1, *sname, no.shares, share.price, capitalization* | *capitalization > 1000000*));

5. X_5 : **INT**$_{10} \longrightarrow$ **INT**$_9$

(Hom-set in $\mathbf{INT}_{10} = f_1$; subobjects in $\mathbf{INT}_{10} = (K_0^1,\ sname \mid K_0^1 \in obj_{\mathbf{INT8}})$);

The first functor X_1 derives the subcategory $\mathbf{INT}_6$ from $\mathbf{INT}_5$ by taking the composition of the arrows $s_1 : K^2{}_0 \longrightarrow 1_{\mathbf{INT4}}$ and $f_9 :\ 1_{\mathbf{INT4}} \longrightarrow voltage$ to determine which part identifiers $K^2{}_0$ 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_1 and on the source of π_1 to cases where the part is in the subobject $K^2{}_0$ derived by X_1.

The third functor X_3 takes the output $\mathbf{INT}_7$ from X_2 and restricts it further to produce the subcategory $\mathbf{INT}_8$ with no arrows and subobject K_0^1. This subobject represents suppliers who supply parts rated at 90 volts.

The fourth functor X_4 produces subcategory $\mathbf{INT}_9$ from $\mathbf{INT}_1$ with the arrows f_1, f_3, f_4, f_5 and subobjects, including $(K_0^1,\ sname)$, for which the application of f_3, f_4, f_5 to K_0^1 gives a capitalization of more than a million pounds.

The final functor X_5 produces the required 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.

5.10.2 Closure in Queries

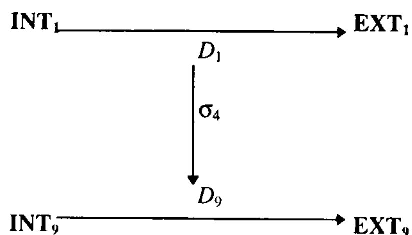


Figure 5-6 - The Query σ_4 as a Natural Transformation with source D_1 and target D_9

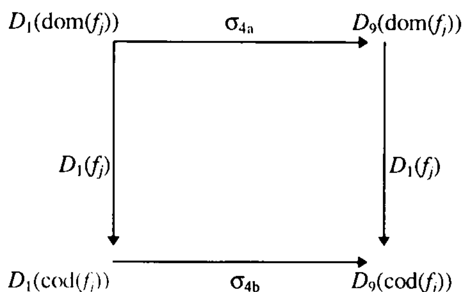


Figure 5-7 - The query σ_4 as a Commuting Target Square with Covariant Natural Transformation σ_4 from functor D_1 to functor D_9

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

The answer is that we have defined a series of subcategories $\mathbf{INT}_6 \ldots \mathbf{INT}_{10}$ in intensional terms but have omitted to define the corresponding extensional subcategories. The relationship between each intension $\mathbf{INT}_i$ and extension $\mathbf{EXT}_i$ is given by the mapping $D_i : \mathbf{INT}_i \longrightarrow \mathbf{EXT}_i$. Therefore for a query earlier, say no.4, we can write in more detail:

$D_1 : \mathbf{INT}_1 \longrightarrow \mathbf{EXT}_1$

$D_9 : \mathbf{INT}_9 \longrightarrow \mathbf{EXT}_9$

as functors for the query representing intension and extension mapping respectively. Each query therefore involves a mapping between an intension-extension pair as source and an intension-extension pair as target. We can represent this structure as shown in Figure 5-6 with the query now represented by the natural transformation σ_4.

To be a natural transformation, the square in figure 5-7 for our current query σ_4 should commute for every arrow f_j : dom(f_j)$\longrightarrow$ 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 $\sigma_{4b} \circ D_1(f_j) = D_9(f_j) \circ \sigma_{4a}$, that is our two paths from the values for domains of arrows in the source category D_1(dom(f_j)) to the values for the codomains of arrows in the target category D_9(cod(f_j)) should be equal. One path A involving σ_{4a} 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_{4b} has the same starting and finishing points but navigates via codomain values in the source category.

In path A, the arrow σ_{4a} 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_{4b} creates a sub object

of the codomains for arrows f_j in $\mathbf{EXT}_1$ to be assigned to the extension category $\mathbf{EXT}_9$. Referring back to the syntax used in our query examples, the hom-set of the target category is defined as the set of f_j assigned by D_9 and the subobjects in the target category are defined as the union of dom(f_j) and cod(f_j) for arrows f_j assigned by D_9.

The output from σ_4 is clearly a structure which can be held in our database, ranking equally with other classes and objects in the system. Typing constraints will continue to be enforced in the output structure. So the typing for objects and arrows in $\mathbf{INT}_9$ will be based on that in $\mathbf{INT}_1$ with the additional constraint that capitalizations must be greater than one million pounds. In computing terms, we are expressing the constraint that no object can exist in our database which is not fully described by a class definition.

In categorical terms, we are expressing a query as a natural transformation. Each functor can be considered as a continuous function (infimum preserving) between two posets with limits: each structure $D_i : \mathbf{INT}_i \longrightarrow \mathbf{EXT}_i$ is then viewed as a closed cartesian category where D_i is a continuous function preserving the infimum (as key) within the poset $\mathbf{INT}_i$ in $\mathbf{EXT}_i$. Closed cartesian categories have been used in other areas of computing science, in formalisms such as Scott domains, as they are equivalent in theoretical power to the typed lambda calculus [bar90].

5.10.3 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:

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

2. 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 5-8 , we can achieve updateable 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.

5.10.4 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 [bar90] which is written $C^{\rightarrow}$ to view the arrows of C as objects in $C^{\rightarrow}$. 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 5-9 commutes, that is $m_n \circ \eta_{ja} = \eta_{jb} \circ m_k$.

Figure 5-9 is the natural transformation target square and shows that the message passing function is a natural transformation between objects in the category of arrows [sim90]. A simple way to realise that inter-arrow morphisms are natural transformations is to consider that the mapping between **CLS** and $\mathbf{CLS}^{\rightarrow}$ is a functor; hence a mapping between **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.

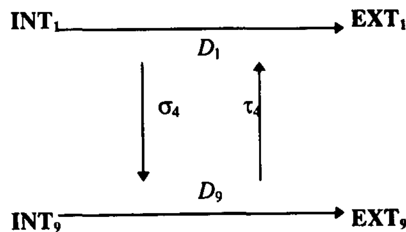


Figure 5-8 - The View σ_4 as a Natural Transformation with Updates through τ_4

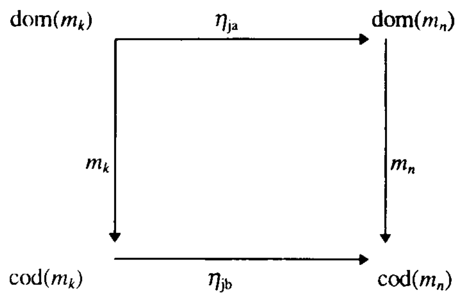


Figure 5-9 - Commuting Square for Message η_j between m_k and m_n in Arrow Categories $\overrightarrow{CLS_i}$ and $\overrightarrow{CLS_j}$ respectively

5.11 Summary

In this chapter we have developed a formal definition for a new data model which is capable of representing the standard abstractions of the object-relational model in category theory. Some of these aspects have been modelled better than others. Further work is required for modelling some of the more advanced concepts such as multiple inheritance and families of categories. However, the Product Model produced does provide a coherent integrated representation of the basic abstractions employed in current information systems. In the future the model could be refined and extended using constructions such as topoi and adjoints [bar90].

In the next chapter we discuss in depth the development of the prototype using the semantic data model database system P/FDM [emb95].

Level	symbol	instance	range i	concept
Category	**ASS**	**ASS**$_i$	$1 .. p$	association intension
	CLS	**CLS**$_i$	$1 .. c$	class
	CLS$^{\rightarrow}$	**CLS**$_i^{\rightarrow}$	$1 .. c$	class with arrows considered as arrow-objects
	DEP	**DEP**$_i$	$1 .. c$	dependencies (in poset)
	EXT	**EXT**$_i$	$1 .. c + p$	database extension
	INT	**INT**$_i$	$1 .. c + p + g$	database intension
	INT$^{\rightarrow}$	**INT**$_i^{\rightarrow}$	$1 .. c + p + g$	intension with arrows considered as arrow-objects
	LNK	**LNK**$_i$	$1 .. p$	association extension
	OBJ	**OBJ**$_i$	$1 .. c$	database object
	PRJ	**PRJ**$_i$	$1 .. c$	persistent variables (in powerset ordered by projection)
	PSU	**PSU**$_i$	$1 .. c$	pseudotransitivities (in poset)
	TYP	**TYP**$_i$	≥ 1	types
	UNI	**UNI**$_i$	$1 .. g$	coproduct (inheritance)
Arrow	D	d_i	$0 .. r'$	dependencies
	F	f_i	$0 .. k$	all arrows within a class
	M	m_i	$0 .. s$	methods
	P	p_i	$0 .. r''$	pseudotransitivity
	S	s_i	$0 .. g$	supertype-subtype
Object	A	a_i	$1 .. n$	persistent variables
	E	e_i	$0 .. r'$	persistent variables in arrows D
	E'	e'_i	$0 .. r''$	persistent variables in arrows P
	K_0^i	k_0^i	$1 .. c$	initial object (key) in **CLS**$_i$
	K_j^i $(1 \leq j \leq r)$	k_j^i	$1 .. c$	non-key attributes in **CLS**$_i$
	U	u_i	$0 .. n'$	memory variables
	V	v_i	$1 .. q$	all variables
Functor	D	D_i	$1 .. c + p$	map intension to extension
	E	E_i	$1 .. c$	map object to class
	G	G_i	$1 .. c$	map variables A to **PRJ**
	G'	G'_i	$1 .. c$	map variables E to **DEP**
	G''	G''_i	$1 .. c$	map variables E' to **PSU**
	I	I_i	$1 .. c$	map object to type

	P	P_i	$1 .. c$	map class to type
	R	R_i	$1 .. p$	map association intension to extension
	V	V_i	$1 .. c$	map class to object
	X	X_i	≥ 0	query mapping intension to intension
Natural transformation	σ	σ_i	≥ 0	query/view deriving one **INT : ENT** pair as a 'subset' of another
	τ	τ_i	≥ 0	dual of query/view σ
	η	η_i	$0 .. m$	message from arrow-object in $\mathbf{INT}_i^{\rightarrow}$ to arrow-object in $\mathbf{INT}_j^{\rightarrow}$

Figure 5-10 - Symbols Employed for Representing Database Concepts

6. Implementation

In this chapter we will discuss the topic of implementing the prototype. We consider the suitability of the semantic database system P/FDM, based on the DAPLEX functional data manipulation language, for the implementation. The techniques involved in obtaining a correct implementation are described. Then finally, we will briefly discuss any improvements that, with foresight, could have been made to the implementation. We will also illustrate any alterations to and omissions from the formal model given in chapter five.

6.1 DAPLEX

DAPLEX is the best known functional database model, developed by Shipman [shi81]. It is based on the concepts of entities and mappings, where entities represent the basic objects in the system, as types in the database, and the arrows (mappings) are functions from one entity, or a group of entities, to a (possibly single valued) set of entities, thereby representing attributes in an object.

DAPLEX is not really a programming language but is a database query language, based on functions and function composition. The entity type can be readily visualised as a category, and function handling in DAPLEX should be equivalent to that which is needed for a categorical database.

The original DAPLEX, as devised by Shipman, has been extended, initially by Kulkarni and Atkinson, and is now known as EFDM [kul86], Extended Functional Data Model. A noticeable weakness of DAPLEX was its inability to attach a 'general computation' function to one of the arrows, that is arrows could not be used for purposes other than expressing relationships. Therefore no general purpose computation other than those

supported by the DBMS could be made. EFDM added limited facilities for supporting general purpose computations, including PS-Algol [kul87] features in their self-contained EFDM system [kul86], as well as facilities for specifying integrity constraints and for handling named views of databases. Later extensions, by Dayal [day89] and by Gray [gra88], to the DAPLEX language, added more general functions to the language. Dayal devised OODAPLEX, an extension of DAPLEX, which also incorporated some object-oriented concepts, but the technique with which functions could be utilised relied too much on object oriented methodology rather than on a functional one.

Gray added object-oriented concepts as well as general purpose functions into EFDM in a much more natural manner than was already supported, so that, for example, a method could act on two different entity types. For instance (grade(Student, Course)) is in essence a join of the two entities Student and Course, something which OODAPLEX could not do. This is a very useful concept for a categorical system to have, as it is equivalent to the arrow concept in category theory. Their system is known as P/FDM, and we discuss in detail this system in the following section. The P/FDM system is embedded in SICStus Prolog [sic93], with very close integration into Prolog, so that a database can be defined in both Prolog and DAPLEX terms, providing a computationally complete query system.

Another useful concept of DAPLEX is that queries can be closed. A FOR EACH statement allows four possible modes for the returning of results. The result can be either:

1. printed to the screen,
2. nested within another query,
3. an UPDATE statement, as defined in the language syntax [gra88, shi81],
4. a general purpose language statement in a language in which DAPLEX is embedded.

For example, the following sample of code would, for each student who is in their third year, produce a new entity which stored the name of their individual project.

```
FOR EACH s in Student
      SUCH THAT year(s) = 3
      FOR A NEW p IN Project
      BEGIN
            LET student(p) = s
            LET title(p) = individual_project(s)
      END
```

The UPDATE option means that as the result of a query, new entities can be defined, or entities already in the database can be updated.

Finally, DAPLEX supports the concept of defining an inverse, so that a new function can be defined which is the inverse of an existing function, or of a composition of functions. This concept would give us a solution for deriving such categorical concepts as duals and adjoints.

Possible difficulties arise in DAPLEX's handling of dynamic types. Declaration of some basic objects to be used in the database categories could be based on the following definitions:

```
declare Attribute() =>> ENTITY

declare IntensionAttribute() =>> Attribute
      declare name(IntensionAttribute) => STRING
      declare value(IntensionAttribute) => STRING

declare ExtensionAttribute() => Attribute
      declare name(ExtensionAttribute) => STRING
      declare value(ExtensionAttribute) => Object
```

The last line of the above definitions would allow the definition of an attribute where its type is to be defined at run-time as some sub-class of a predefined object entity.

DAPLEX does not allow dynamic binding, so this may cause problems, as it is trying to override the type checking system of DAPLEX, where we have in effect defined a function with an undefined type. Use of metadata in P/FDM [emb92], which can be accessed uniformly in a DAPLEX query (that is there is no apparent difference between metadata and application data in a query) should allow some form of type manipulation to be added for simulating dynamic typing.

6.2 An Extended Review of P/FDM

P/FDM, developed by the Object Database Group at the University of Aberdeen, is a semantic data model database system, based primarily on the functional data model database definition and manipulation language DAPLEX, developed by Shipman in 1981. DAPLEX claims to be a conceptually natural database interface language [shi81]. It is based on functions and function composition, that is navigation in the language is based on functions operating on entities, and composition of those functions to perform quite complex operations.

The original functional data model of Kerschberg and Pachecho [ker76] was developed to address the lack of naturality in previous data models. They noticed the similarity in previous models in how they treated relationships and navigation, so they defined the functional model in an attempt to add a theoretical basis to bridge between the models with a more natural query system than SQL.

DAPLEX was developed in 1981, at the same time as the Semantic Data Model (SDM) by Hammer and McLeod [ham81]. DAPLEX was more successful than SDM, even though Shipman never actually produced an implementation, because DAPLEX had a much clearer and simpler model. Actual implementations of DAPLEX came later, firstly as ADAPLEX, which embedded a subset of DAPLEX in Ada and was developed at Computer Corporation of America [smi81], and secondly as P/FDM, which is the system we use. Other implementations of the DAPLEX language came

from Dayal who devised OODAPLEX [day89], an attempt to produce an object oriented version of DAPLEX, and EFDM, which was an early implementation using the language PS-Algol and developed by Atkinson [atk83].

P/FDM can be classified as a semantic data model, that is strongly based on DAPLEX. It has a DAPLEX interface, as well as the ability to perform the same functions as defined in the DAPLEX language directly in Prolog code. It has the capability of object-oriented databases and could be termed object-functional.

The two main concepts of DAPLEX are the entity and the function, where functions are equivalent to arrows in our model (although later we explain a different method for representing arrows), and categories can be thought of as similar to DAPLEX entities. The concepts of DAPLEX are therefore very similar to those in a categorical database, with constructions based on arrows, as we have previously discussed. An obvious problem is that DAPLEX is essentially flat (that is the arrows are basically represented at only one level as with set theory), so we still have the problem of representing a multi-level formalism on top of this. We do not see this as a problem though because it is one we would have in most other programming languages.

The architecture of the P/FDM system is as represented in the diagram below.

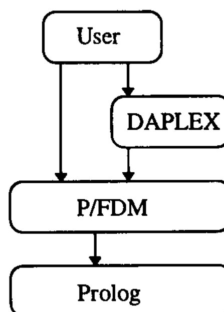


Figure 6-1 - Diagram of Architecture of P/FDM System

From this diagram, we can see how the system is constructed in layers. The main development environment is based on Quintus or SICStus Prolog [sic93], SICStus being the system which we use. A hash file manager is incorporated, where gdbm, which is a GNU version of ndbm hash file manager, is used. P/FDM is then built on top of these to produce the P/FDM Prolog interface, and on top of this is provided a DAPLEX interpreter. The user has access to either DAPLEX or Prolog, and features defined in one can be used in the other in a reasonably straight-forward manner.

The schema is written in DAPLEX, Prolog is mainly used when the functionality of the DAPLEX language is inadequate for a particular method, for example, when we need to include some general purpose computation, or to override the DAPLEX type system. The latter occurs quite often in our system, mainly because the type of an attribute may not be known when we parse the schema but will be known later at run-time.

The advantages that the DAPLEX system gives us over other implementation bases were discussed in the previous section, but the availability of DAPLEX is just one of the reasons why P/FDM seems a suitable vehicle for developing the prototype. We now detail further the advantages that P/FDM gives us:

- The whole of the functionality of the DAPLEX language is provided in the Prolog interface of P/FDM, and there is also the ability to use standard Prolog to enhance DAPLEX queries. So, in Prolog we have the facility for writing quite complex queries and updates to the database and metadata (see below). Note that any methods defined in DAPLEX have a version in Prolog as well, and *vice versa*, to avoid the problem of not being able to use DAPLEX methods in Prolog.
- The whole of the functionality of the DAPLEX language is also provided in P/FDM's DAPLEX interface, with a few extensions and changes, notably to make it more 'object-oriented'; to update the metadata layer and constraints system; and to improve the print command to make it compatible with a declarative style. Shipman's original definition of the model was developed on the basis that it would

be embedded in an imperative language such as Pascal and thus he relied heavily on existing language features.

- The physical database implementation in P/FDM is provided by using the gdbm hashing interface, incorporated into Prolog. This means that we do not have to worry about implementing our own physical layer, a cumbersome task which is not an important part of our work.
- The Prolog basis ensures that there are limited typing restrictions compared to other languages such as Pascal or C, but also results in the absence of dynamic types. Because P/FDM is based on Prolog, we do not have to worry about any type restrictions, in particular Prolog allows heterogeneous collections of values, but we have to carry out our own type checking for example in displaying objects where we do not always know the true type of the object (we only know that it is a subtype of the general *attribute* type). A difficulty is that P/FDM adds its own type system to that of basic Prolog which can be too strict for our needs, especially when querying in DAPLEX, but it does have subtype support, so we can effortlessly store heterogeneous lists inside DAPLEX structures and use our own type checking routines to resolve ambiguities when processing these structures.
- The latest release of P/FDM contains constraints [emb94] which we could use for type checking and subtyping in a future version. It is based on, for example, being able to add restrictions to the types of an entity in the schema definition (similar to pre- and post-conditions in Eiffel), but we had to write type checking functions because its constraint facility was not available at the time we developed our prototype.
- DAPLEX has some similarity to category theory in that entities = categories and functions = arrows, although as with most set based systems, these features can not be simply extended to provide multi-level mappings.
- P/FDM has closure at the entity level. We can use P/FDM closure for easy development of categorical queries, where new structures need to be created as the result of a query. But the closure we eventually want is at the functor level.

- P/FDM has automatic inverses. If new structures are created with a link from the old structures, a link is retained to the old structure as well. This could be used as a method for creating the categorical concept of duals.
- The metadata level of P/FDM is very well defined, and the interface to the metadata level is completely transient, so it could be used for checking types.
- One of the attractive features of P/FDM is the uniform use of DAPLEX and Prolog, so we can use the complete functionality of Prolog to overcome inadequacies in DAPLEX. This is especially useful for our overriding of the P/FDM type checking

6.3 Implementation

In this section we will explain in detail our implementation of the Product Model, following the definition of this formal model given in chapter five. We will show any alterations to the formal model which were required for a feasible implementation, suggesting the reason why something was implemented in a particular way, and highlighting the main problems that occurred and how a solution was found.

The implementation is based on P/FDM, for reasons which were discussed in the previous section. The architecture, involving the insertion of the Product Model into figure 7-1, is shown in figure 7-2.

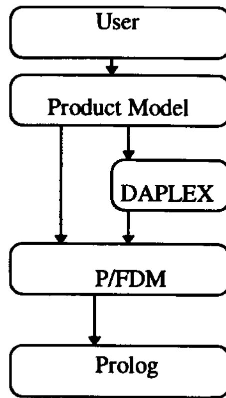


Figure 6-2 - Diagram of Architecture of Product Model

Figure 7-2 shows that the categorical database system conveniently sits on top of the DAPLEX and P/FDM (Prolog interface) layers, ensuring a seamless integration of the Product Model into the P/FDM system (that is a user notices no difference apart from the introduction of categorical types). The interface is a collection of DAPLEX (and Prolog) functions which can be used consistently in both the P/FDM DAPLEX and Prolog interfaces. This is because the categorical system is implemented basically as a database schema defined in the DAPLEX language. This saves us having to write a database modelling and manipulation language, all the Product Model functionality and data are defined in standard P/FDM terms. It also means that the database is interoperable in terms of integration with non-categorical DAPLEX databases.

6.3.1 Introduction

We now look at each property of the Product Model as defined in the Product Model chapter, highlighting the technique we used for implementation in P/FDM. Fragments of P/FDM code including schema and functions are given throughout.

6.3.2 Classes

The basic structure is defined in section 5.3.1. Attributes are defined as subtypes (for example integer, string) of a type *attribute*, and we then define value and display functions for each subtype. These are then stored inside an object set, so that we can store more complex values, that is non-atomic and heterogeneous values, as a group. Attribute positions within the object sets are indexed so that the functors can correctly map between attributes in object sets. Arrows are then defined as an entity containing two object sets, a source and a target, as described in the previous section. A special type of arrow is defined for storing identity arrows, in case an attribute cannot be stored in the category as a source or target of some other arrow. This is so that categories then just contain arrows and any objects which do not participate in an arrow mapping can be stored as identity arrows. An example of such a structure would be an object that is 'all-key', that is all of its attributes are part of the key. The code is implemented as a single module [emb95] in P/FDM, called *product_model*. This is shown in the definition of derived functions such as *display* below, where the function name is qualified as being *in* the module *product_model*.

Attributes are the base entities in the system. They have an *identity integer* (equivalent to the method of identifying objects in an object-oriented database) which provides an unique identifier for each attribute, a type name stored as a string and a type identifier, where all types in the system also have an unique identifier. *attr_type* stores the type of the attribute (*i* for intension, *e* for extension). An *object_set* is then just a collection of attributes, and there is a function *os_position* which uniquely identifies the position of any attribute in any object set. The subtype *c_int* is shown as the class of all attributes of type integer, with the function *display* defined for displaying integers.

```
declare attribute ->> entity
      declare id(attribute) -> integer
      declare type_name(attribute) -> string
```

```
      declare type_id(attribute) -> integer
      declare attr_type(attribute) -> string
      key_of attribute is id

declare object_set ->> entity
      declare id(object_set) -> integer
      declare elements(object_set) ->> attribute
      key_of object_set is id

declare os_position(object_set, integer) -> attribute

declare c_int ->> attribute
      declare value(c_int) -> integer

define display(c in c_int) -> integer in product_model
      value(c);
```

Arrows are defined using the *cat_arrow* entity, which is basically a structure containing a source object set, a target object set and an unique name for the arrow. Then, categories are defined as containing a set of these *cat arrows*, with a group of objects which is calculated as all object sets of all the arrows in the category.

```
declare cat_arrow ->> entity
      declare name(cat_arrow) -> string
      declare source(cat_arrow) -> object_set
      declare target(cat_arrow) -> object_set
      key_of cat_arrow is name

declare category ->> entity
      declare name(category) -> string
      declare arrows(category) ->> cat_arrow
      declare objects(category) ->> object_set
      key_of category is name
```

Currently the only type of arrows we store within our categories are static (of type *D* in section 5.3.1), that is they are just relations between attributes. But it is not a difficult task to incorporate dynamic arrows, they are just a special case of a static arrow. We simply redefine the target of an arrow to be a defined method, as illustrated

in the second example in the following chapter. In fact, in the next chapter we show how to define a dynamic arrow for the *number of shares* arrows in the *electrical parts* example, although we do not actually execute it due to the amount of effort involved in establishing the environment.

For determination of keys, rather than store the whole poset as described in section 5.1.2, our class implementation simply holds the collection of functional dependencies, as mappings from the primary key (initial object) to all attributes within the object.

We have a simplified method to determine the primary key from this information and to test whether the object conforms to BCNF, which is the level of normalisation required, except that the atomicity rule (1NF) has been relaxed.

The normalisation technique of section 5.3.2 has not yet been incorporated into the prototype. Currently it exists as a Prolog program, so could be integrated into the P/FDM system in the future. The method works by recursively subtracting permutations of the non-trivial functional dependencies from the maximal element in the partially ordered set of the attributes, until the greatest lower bounds or infimum have been found. An example of this is given in [nel95]. This algorithm is also capable of determining whether a given set of functional dependencies conforms to BCNF because it produces the complete set of candidate keys. Our algorithm is perhaps not as efficient as previous algorithms [osb79] but gives us the advantage of also determining the key.

6.3.3 Relationships

We have looked at the two principal types of relationship defined in the Product Model: binary relationships and inheritance.

6.3.3.1 Pullback relationships

We have pullbacks for intension and extension as defined in section 5.4, with a functor mapping between them. Our pullbacks are defined as relationships between limits and initial objects, with consistency checks to determine the cardinality and membership class of a pullback, in terms of epimorphisms and monomorphisms. The pullbacks are defined as a group of limits with initial objects mapping into categories. Some of the code for pullbacks, and the functions for testing their type is given below. The code includes an initial object entity, which maps from an identity object set to the category which that object set references. Then, *pi_arrow* as well as *f_arrow* and *co_eq_arrow* are defined as collections of initial objects and limits. Finally a *pullback* is defined as a collection of the above types of arrow.

```
declare init_object ->> entity
      declare element(init_object) -> object_set
      declare refs(init_object) -> category
      key_of init_object is key_of(element)

declare limit ->> entity
      declare left(limit) -> init_object
      declare right(limit) -> init_object
      key_of limit is key_of(left), key_of(right)

declare pi_arrow ->> entity
      declare source(pi_arrow) -> limit
      declare target(pi_arrow) -> init_object
      key_of pi_arrow is key_of(source), key_of(target)

declare f_arrow ->> entity
      declare source(f_arrow) -> init_object
      declare target(f_arrow) -> init_object
      key_of f_arrow is key_of(source), key_of(target)

declare co_eq_arrow ->> entity
      declare source(co_eq_arrow) -> limit
```

```
      declare target(co_eq_arrow) -> init_object
      key_of co_eq_arrow is key_of(source), key_of(target)

declare pullback ->> entity
      declare name(pullback) -> string
      declare pi_l_arrows(pullback) -> pi_arrow
      declare pi_r_arrows(pullback) -> pi_arrow
      declare f_arrows(pullback) -> f_arrow
      declare g_arrows(pullback) -> f_arrow
      declare co_eq_arrows(pullback) -> co_eq_arrow
      declare left_category(pullback) -> string
      declare right_category(pullback) -> string
      declare over_category(pullback) -> string
      key_of pullback is name

define pi_l_monic(p in pullback) -> boolean in product_model
      if some o in target(pi_l_arrows(p)) has
            count(a in target(pi_l_arrows(p)) such that a = o) > 1
                  then false else true;

define pi_l_epic(p in pullback) -> boolean in product_model
      if some o in init_object has
            count(a in target(pi_l_arrows(p)) such that a = o) = 0
            and name(refs(o)) = left_category(p))
                  then false else true;
```

The function *print_arrow_types* below examines properties of the pullback, in particular the projection arrows, to test if the pullback is monic and epic to give the cardinality and membership class of the relationship, as discussed in section 5.4.

```
define print_arrow_types(p in pullback) in product_model
      print('pi_l is
            if pi_l_monic(p) = true then 'monic (1)' else 'not monic
(N)',
            'and'
            if pi_l_epic(p) = true then 'epic (mand)' else 'not epic
(opt)'),
      print('pi_r is
            if pi_r_monic(p) = true then 'monic (1)' else 'not monic
(N)',
```

```
            'and'
            if pi_r_epic(p) = true then 'epic (mand)' else 'not epic
(opt)'),
      print('Relationship is', if pi_r_monic(p) = true then '1' else
'N',
            ':', if pi_l_monic(p) = true then '1' else 'N');
```

We do not as yet have a facility for *n*-ary relationships, but it could be done relatively simply by pasting together binary pullbacks, and ensuring that our limits are *n*-ary rather than just binary.

6.3.3.2 Inheritance

In the Product Model, we have a simple co-product structure as in section 5.4.3 that automatically creates a new intension category holding the sum of the attributes in the specialisation and generalisation. For implementation purposes we encountered the classical inheritance problem [car84, mey88] of how to handle the extension: if a sub-class is created, do we store the parent information in that sub-class also, or use the parent class as well? Our method stores both the supertype and the new attributes for the new type as two separate categories, and we have a method for producing the co-product of these two as a new category. The code for this is shown below. This can introduce storage anomalies because of redundancy. Any future system would have to be able to cope with this problem. Again, we ignore multiple inheritance; previous work [car85, bra93] has shown that it is a difficult problem to solve philosophically.

The code below shows the definition of a co-product as a collection of injection arrows and strings identifying which categories are being summed. Then, the function *make_sub_class* makes a new co-product category as a union of the two categories in the co-product, ensuring that the key dependency is preserved by creating a new arrow between the initial objects of the two categories, as discussed in section 5.4.3.

```
declare co_limit ->> limit
```

```
declare inj_arrow ->> entity
      declare source(inj_arrow) -> init_object
      declare target(inj_arrow) -> co_limit
      key_of inj_arrow is key_of(source), key_of(target)

declare co_product ->> entity
      declare name(co_product) -> string
      declare left_inj(co_product) -> inj_arrow
      declare right_inj(co_product) -> inj_arrow
      declare left_category(co_product) -> string
      declare right_category(co_product) -> string
      key_of co_product is name

declare co_prod_category ->> category
      declare union(co_prod_category) -> co_product

define do_left(c in co_prod_category) in product_model
      for each a in arrows(refs(source(left_inj(union(c)))))
            include {a} into arrows(c);

define preserve_key(c in co_prod_category) in product_model
      make_arrow(name(c),
            element(source(left_inj(union(c)))),
            element(source(right_inj(union(c))))),
      include {the a in cat_arrow such that name(a) = name(c)} into
arrows(c);

define make_sub_class(p in co_product) in product_model
      create a new c in co_prod_category with key = (name(p)),
      let union(c) = p,
      do_left(c),
      do_right(c),
      preserve_key(c);
```

6.3.4 Typing

We have implemented a simple type system, in two major components. First of all we have a *type* category as defined in section 5.6, which stores all information about a

particular type - a relationship to a particular type information object and a set of attributes which conform to that type both intensionally and extensionally.

```
declare sub_type ->> entity
      declare id(sub_type) -> integer
      declare parent(sub_type) -> string
      key_of(sub_type) is id, parent

declare integer_range ->> sub_type
      declare lower_limit(integer_range) -> integer
      declare upper_limit(integer_range) -> integer

declare type_category ->> entity
      declare type_defn(type_category) -> sub_type
      declare int_attr_set(type_category) ->> attribute
      declare ext_attr_set(type_category) ->> attribute
      key_of type_category is key_of(type_defn)
```

In addition, we have *type* entities, which store for a particular type its description, including, name of the super type (that is integer, string), with a lower and upper limit on any acceptable value of that type (which may or may not be defined so that we can represent ranges), as well as other concepts which could define a sub-type, such as the maximum or minimum length for string values.

We have developed a collection of routines which test if an attribute is of a particular type, look at attributes and see which type they belong to. The code for the function *type_check* which performs the major task of ensuring that a particular attribute is of a particular type is shown below. The types that have been implemented in our system are basic scalar types where methods are available for defining new types as ranges (sub-range and super-range inclusively). It was quite a complex task to do anything more than this with the version of P/FDM we were using because of the need to define the actual type representation as discussed in section 6.2. Note though that this is an easier task in newer versions of P/FDM where constraints can be defined [emb94].

```
define attr_type_check(i in attribute, e in attribute) -> boolean in
product_model
      if type_check(e, the s in sub_type such that
            id(s) = type_id(i)) = true
      then true else false;

type_check([attribute, sub_type], [Attribute, SubType],
BooleanResult) :-
(
      getfnval(parent, [SubType], ShouldType),
      getfnval(type_name, [Attribute], IsType), !
      ShouldType == IsType,
      instance_type(Attribute, AttrInsType),
      getentity(AttrInsType, Attribute, _, [H1|_]),
      getfnval(value, [H1], Value),
      getfnval(id, [SubType], TypeId),
      (
            (
                  (TypeId == 1 ; TypeId == 2 ; TypeId == 3 ; TypeId
== 4)
            );
            instance_type(SubType, SubInsType),
            getentity(SubInsType, SubType, _, [H2|_]),
            (
                  (
                        getfnval(lower_limit, [H2], Lower),
                        getfnval(upper_limit, [H2], Upper),
                        TestType is 1
                  );
                  …
                  , !,
                  (
                        (
                              TestType = 1,
                              Lower =< Value,
                              Value =< Upper
                        )
                  )
            )
      ),
      BooleanResult = 'true'
```

```
); BooleanResult = 'false'.
```

6.3.5 Objects

In our Product Model, objects are the extension categories in the database, although there is still difference of opinion among object-oriented workers whether an object is both intension and extension or just the extension. However, in our Product Model, we have taken the decision to consider objects as extension categories. So, we have an extension category which stores all instances of values as defined by an intension, and a functor between them which stores the mapping from object names (and types) in the intension to object values in the extension. The functor is simply a relation, it can check that an instance does conform to the intensional definition, and that it is of the correct type, which is done through use of the type triangle shown in figure 5-5. In our system, the type triangle is checked by simple tests of extensional values to see if these values conform to the type specified by the intension. The code for this is shown below. We therefore have no need of the third type category **TYP**, although our tests ensure that we have the consistency defined by that type category.

```
define attr_type_check(i in attribute, e in attribute) -> boolean in
product_model
      if type_check(e the s in sub_type such that
            id(s) = type_id(i)) = true
      then true else false;

define compare_arrow_types(i in cat_arrow, e in cat_arrow) -> boolean
in product_model
      if some j in {1 to count(elements(source(i)))} has
            attr_type_check(os_position(source(i), j),
os_position(source(e), j))
            = false
      or some k in {1 to count(elements(target(i)))} has
            attr_type_check(os_position(target(i), k),
os_position(target(e), k))
            = false
      then false else true;
```

```
define compare_functor_types(f in pm_functor) -> boolean in
product_model
      if some i in arrow_mappings(f) has
            compare_arrow_types(source(i), target(i)) = false
      then false else true;
```

Another constraint which we could enforce, but currently do not, is that the functional dependencies hold in the data. This is quite a difficult test, more than just checking that the instance arrows are the same as the extension. We could however compare arrows because our model allows us, through use of the *object_set* construction, to map down to the attribute level within any object within a category.

6.3.6 Encapsulation

As detailed in section 5.7, categories enclose all their attributes and operations as a collection of arrows within the category itself. All properties of a category can be referenced by manipulating the initial object of the category, an object which can directly or indirectly map onto every other property within the category.

6.3.7 Physical Storage Structures

We do not have to worry about how our database is stored physically, as by defining our categorical database as a DAPLEX schema and functions, our data is stored using the standard P/FDM hashing system. This ensures that although our system is only a prototype, accessing of the data in its physical stored form should have performance directly related to that of P/FDM, without any major overheads, apart from those arising in the Prolog code that may have been required as extensions to the DAPLEX schema.

6.3.8 Families of Categories

Topoi [bar90] could be used as a way of grouping categories into families, so that our database system could then be represented formally by a single topos. This aspect is still under consideration at the theoretical level and no implementation of topoi has been attempted.

6.3.9 Functors

We have defined two types of functor with code given below. `pm_functor` provides the simple mapping between intension and extension, and `pullback_functor` maps between intension pullback and extension pullback. Because we can not expect lists in P/FDM to be order preserving, attributes are always stored in object sets in an indexed order, so when we link two object sets we know that P/FDM has not affected the ordering of attributes. Again, the majority of our functors are defined as relations between categories, rather than anything more dynamic, although in our natural transformations we do compute functorial relationships.

```
declare arrow_mapping ->> entity
      declare source(arrow_mapping) ->> cat_arrow
      declare target(arrow_mapping) ->> cat_arrow
      key_of arrow_mapping is key_of(source), key_of(target)

declare pm_functor ->> entity
      declare name(pm_functor) -> string
      declare source(pm_functor) -> category
      declare target(pm_functor) -> category
      declare arrow_mappings(pm_functor) ->> arrow_mapping
      key_of pm_functor is name

declare pullback_mapping ->> entity
      declare id(pullback_mapping) -> integer
      declare pi_1_source(pullback_mapping) -> pi_arrow
```

```
        declare pi_l_target(pullback_mapping) -> pi_arrow
        declare pi_r_source(pullback_mapping) -> pi_arrow
        declare pi_r_target(pullback_mapping) -> pi_arrow
        declare f_source(pullback_mapping) -> f_arrow
        declare f_target(pullback_mapping) -> f_arrow
        declare g_source(pullback_mapping) -> f_arrow
        declare g_target(pullback_mapping) -> f_arrow
        declare co_eq_source(pullback_mapping) -> co_eq_arrow
        declare co_eq_target(pullback_mapping) -> co_eq_arrow
        key_of pullback_mapping is id

declare pullback_functor ->> entity
        declare name(pullback_functor) -> string
        declare source(pullback_functor) -> pullback
        declare target(pullback_functor) -> pullback
        declare pullback_mappings(pullback_functor) ->>
pullback_mapping
        key_of pullback_functor is name
```

6.3.10 Manipulation

Natural transformations provide the basis for manipulation. Although in the formal model in figure 5-8 a natural transformation is defined as a mapping between functors, they are implemented as a commuting target square of four functors, as shown in figure 5-7. This is because it is easier for the user to understand and is therefore simpler to implement in P/FDM because a new level of mappings is then not required. The morphism (functor) D exists already (it is the database functor or mapping from intension to extension), σ is defined as mapping the intension types to their new types (defined by the new category $\mathbf{I}'$), and then the two new functors D' and τ are automatically created as the result of the natural transformation, to produce the new category $\mathbf{E}'$.

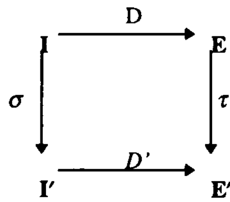


Figure 6-3 - Natural Transformation in P/FDM

The four functors ensure we have the equivalent of a natural transformation by ensuring that all arrows in the source and the target categories are consistently mapped and thus the diagram above commutes for any query. The code for natural transformations is given below, showing that a natural transformation is basically a collection of four functors (which are discussed in section 6.3.9):

```
declare nat_trans ->> entity
      declare name(nat_trans) -> string
      declare source(nat_trans) -> pm_functor
      declare target(nat_trans) -> pm_functor
      declare left_functor(nat_trans) -> pm_functor
      declare right_functor(nat_trans) -> pm_functor
      key_of nat_trans is name

define new_functor(n in string, i in category, e in category) in
product_model
      create a new p in pm_functor with key = (n),
      let source(p) = i,
      let target(p) = e;

define add_arrow_map(a in cat_arrow, b in cat_arrow, f in pm_functor)
in product_model
      create a new m in arrow_mapping with key = (key_of(a),
key_of(b)),
      include {m} into arrow_mappings(f);
```

The function *perform_closed_nat_trans* as shown below completes a natural transformation by creating the category $\mathbf{E}'$ as well as the functors τ and D' (known as *right_functor* and *target* respectively in the implementation). The natural transformation expects the functors *source* and *left_functor* to exist already. *source* is the intension to extension functor in the existing database, *left_functor* is a mapping from the original intension to the new intension category (which is also defined by the user) and shows which arrows are to be kept in the query result and also ensures that the type conversions between old intension and new intension attributes are defined.

The method used ensures that we can also roll back a natural transformation to the state it was before the query was produced, a very simple form of transaction. This is done simply be removing the τ and D' functors and removing the category $\mathbf{E}'$.

```
define closed_nat_trans_pt3(n in nat_trans, a in arrow_mapping, s in
arrow_mapping, c in category) in product_model
      for the e in cat_arrow such that name(e) =
            string_concat(name(target(a)), '_~')
            include {e} into arrows(c),
            add_arrow_map(target(s), e, target(n)),
            add_arrow_map(target(a), e, right_functor(n));

define closed_nat_trans_pt2(n in nat_trans, c in category) in
product_model
      for each a in arrow_mappings(source(n))
            for the s in arrow_mappings(left_functor(n)) such that
                  source(a) = source(s) and
                  compare_arrow_types(target(s), target(a)) = true
                        make_prime_arrow2(target(s), target(a)),
                        closed_nat_trans_pt3(n, a, s, c);

define perform_closed_nat_trans(n in nat_trans) in product_model
      create a new c in category with key =
(string_concat(name(target(source(n))), '_~')),
      new_functor(string_concat(name(source(n)), '_~'),
target(left_functor(n)), c),
      new_functor(string_concat(name(left_functor(n)), '_ext'),
target(source(n)), c),
```

```
      let target(n) = the f in pm_functor such that name(f) =
string_concat(name(source(n)), '_~'),
      let right_functor(n) = the g in pm_functor such that name(g) =
string_concat(name(left_functor(n)), '_ext'),
      closed_nat_trans_pt2(n, c);
```

At the moment we do not have natural transformations which are capable of mapping between pullback functors, although this should be feasible in the future. A decision is needed to determine whether the natural transformation should be capable of mapping pullbacks to standard categories, enabling our natural transformations to produce ***sub-sets*** of the arrows within a pullback, or just of the objects within the pullback, that is the pullback is treated as a category in the mapping, which would mean that rather than implementing a new type of natural transformation, a simple conversion utility for converting pullbacks to basic categories would be needed.

6.3.10.1 Closure

Through the natural transformation a query produces a new category ($\mathbf{E}'$) and new functors (D' and τ) which provide a consistent mapping between the old and the new representation of the data. That is the diagram in figure 7-3 commutes. These categories and the functor D' are first-class data in the same way as the original categories in the system, which means that they can be referenced in further queries without any change in syntax or method. Our queries therefore satisfy the closure requirements as detailed in section 5.10.2.

6.3.10.2 Queries

To perform queries, each natural transformation will be more than just a relation. It will need to actually perform operations, in particular in mapping between related types, such as restricting the set of integers to those within a particular range. This is done by using typing in a very tight manner, where we decide which attributes satisfy

the constraints of the query by defining type mappings. That is we map a type in the source intension to a new subtype in the target intension and thus the mappings between values in the source extension and new target extension also hold the same typing constraints.

The user supplies a new intension, with types defined, and a functor from this new intension to the old intension, showing the actual type mappings which should take place. The new extension is then created using the code shown in section 6.3.10 above, and the commuting square is completed, by creating the functors τ and D'. Our new categories can then be used within the database for further querying, allowing us to perform complex query transformations, and also ensuring we have closure in our system.

It should be emphasised that we do not automatically create the whole of the natural transformation. It is much simpler to manually define the new intension and its relation to the old intension by defining the functor σ ensuring that we have defined the required type mappings to map between source and target extensions. Then, the task our system has to perform to complete the natural transformation is to create the functors as explained above, and returning the category $\mathbf{E}'$ as the result. However, this is still preferable to the current solution in some object oriented systems which restrict queries to single objects [ber93 p 87/88] and thus restrict nested queries.

6.3.10.3 Views

Views have not been implemented in the present study. Read-only views could be constructed very simply in the same manner as queries as shown in section 6.3.10. Views with update facilities could be constructed by adding a natural transformation in the dual direction to a query. One method could be simply to use DAPLEX inverses, but this does not give what is needed because DAPLEX inverses work only at the lowest function level whereas our requirements are multi-level. However, managing

multi-level inverses could prove difficult and costly to maintain, so it is much easier to reverse the source and target of an arrow mapping, and then when we update an instance, we could check the arrow maps to see if a parent, or a view needs updated, ensuring that all typing constraints still hold.

6.4 Conclusions

Categories representing classes, objects, binary relationships, inheritance and type systems were defined. Functors were developed to relate classes, at the intensional level, to objects, at the extensional level. Natural transformations, as expected, were more difficult to develop and may need further thought in subsequent implementations. We succeeded in constructing natural transformations with query closure in specific circumstances but more universal mapping capabilities are desirable in the long run to handle general categories as objects including pullbacks. We did not implement views, but their implementation is relatively straightforward given more time.

No interface other than through DAPLEX has been provided. Because category theory naturally uses the arrow as its major concept at various levels, and diagram chasing for proofs, then it seems that a graphical interface would be the ideal interface to a categorical database system, especially one which closely matches an entity-relationship model. Graphical interfaces are discussed further in Nelson [nel93] and Kappel [kap92] which investigates the features for which the current object-oriented database management systems provide an interface.

In the next chapter we illustrate use of the prototype by producing two small sample applications. The first is a very basic example, whereas the second is closely based on the example given in chapter five.

7. Results

7.1 Introduction

In this chapter we will illustrate use of the prototype with two sample applications. The first test suite is a simple application involving a basic collection of classes and a simple relationship, the type of application which could be handled easily by the relational model. A simple query on one of the tables is illustrated. The second example is taken from chapter four, where we produce part of the solution to the electrical parts query. We do not perform the complete query here because the method of setting up a query in the prototype system is very time consuming, and usability to the level required would involve further work which is beyond the scope of this thesis. In particular, as noted in section 6.3.10, we have not as yet implemented a system for producing natural transformations between pullback functors - the method is similar to that for natural transformations between normal categories, and could be achieved by extensions to that work.

7.2 Testing Method

The method we use both for defining the schema (in terms of a DAPLEX schema definition, DAPLEX and Prolog functions) and for defining the sample database applications, is the bulk load facility described in the P/FDM user manual [emb95], which allows us to specify all the data values for the entities and the relationships between entities. It eliminates the need for producing complex DAPLEX entity definition commands, although by looking at the size of the bulk-load files even for a simple database, it is still quite a complex task.

7.3 Manipulation Technique

In our Product Model there are two methods for manipulating the database, we can produce results with output or first-class data structures, or perform the query without closure. We are also able to completely unroll a closed query, that is we can remove from the system the functors, the resulting category, and the other structures that are created during execution of the query. Basically, we run DAPLEX queries as follows:

```
for the n in nat_trans such that name(n) = 'our_nat_trans'
perform_nat_trans(n);
```

This produces the non-closed version of the natural transformation, displaying the results of the query, but not storing the new extension category produced. To execute the query with closure, we define the following DAPLEX query (which runs the function shown in section 6.3.10):

```
for the n in nat_trans such that name(n) = 'our_nat_trans'
perform_closed_nat_trans(n);
```

which will run the identical query, this time giving no output to the screen, but storing the results of the query. We can unroll the query as follows:

```
for the n in nat_trans such that name(n) = 'our_nat_trans'
unroll_closed_nat_trans(n);
```

7.3.1 Other Manipulation Functionality

To display the actual information stored within the database, we have facilities to display categories, functors, pullbacks and pullback functors, as well as the type information and the list of arrows.

To print the categories, we can either use the Prolog function `show_categories` or perform the DAPLEX query:

```
for each c in category print_category(c);
```

To print a pullback `p` we can use either the DAPLEX functions `print_pullback_info(p)` or `print_pullback(p)`. The first prints complete details of the pullback including the results of the type tests on the arrows, giving the cardinality and membership class of the relationship. The second only prints the co-equaliser arrows and again the type tests on the pullback.

Co-products in our system work at the intensional level only. We use the DAPLEX function `make_sub_class(p)` for some defined co-product `p`, which makes a new category containing the union of two predefined categories (the subtype and the supertype).

To display functors, we can use `print_functor(f)` for a functor `f` in the same way as above, which prints all functors, including those stored within natural transformations. We have two functions for printing functors between pullbacks, `print_pullback_functor_info(f)` and `print_pullback_functor(f)`, the first printing full detail (all the arrow mappings), the second printing the minimal detail (only the co-equaliser arrow mappings) between the source and target pullbacks.

Finally, to print all type information, which will test all simple attributes in the database with all relevant types, we have the Prolog function `show_types`. To show all arrows including those in the lowest level structure where objects are represented by identity arrows, we have the Prolog function `show_hom_sets`.

7.4 Simple Test - Supplier Parts Example

In this example, we produce a very simple database consisting of the following:

- supplier category (intensional and extensional), containing the arrow:
    - supp_no -> supp_name (intension)
- parts category (intensional and extensional), containing the arrow:
    - part_no -> part_name (intension)
- an orders category, containing values for order numbers for each supplier part relationship in the pullback, represented as identity arrows.
- a pullback representing a relationship between the supplier and parts categories, over the set of orders represented by the orders category
- a simple intensional definition of inheritance represented by a co-product, extending the supplier category. We also convert this to an extensional sub-type by applying a P/FDM function to the database
- a functor mapping each intension to each extension
- a pullback functor for the supplier parts relationship
- a typed natural transformation on the supplier category representing the query '*find the supplier with supplier number 1*'.

This simple example shows that it is possible to define natural transformations of the type specified in the formal Product Model defined earlier in chapter five, but that it takes considerable effort to define all the constructs required for such a simple query.

A further prototype could ease this work by providing a user interface which would automatically define most of the properties required, but this is not part of this thesis.

Sample output from the system is shown below in a number of stages

1. First of all, we print the categories that exist, *supplier* and *part* are the intension categories, *orders_category* is the *orders* intension, which is used in the pullback.

suppliers_ext, parts_ext, orders_ext are the respective extension definitions for the above categories. In addition *subs_cat* is the extra arrow defined for the co-product and *new_supp_int* defines the new intension category for the query of the suppliers category with the type of *supp_no* changed to be the set of integers equal to one. The code below prints the categories in the system, listing the arrows within each category. Each attribute is qualified by its type name, and a number in brackets, which is the actual type identifier of the type name, where *1* is the set of all integers, *3* is the set of all strings, and in category *new_supp_int*, *100* is the new integer type explained above which is restricted to be the set including the sole integer 1.

```
|: for each c in category print_category(c);

Correctly formed imperative.




Category supplier

fd2 :: supp_no:integer(1)  -> supp_name:string(3)

Category part

fd4 :: part_no:integer(1)  -> part_name:string(3)

Category orders_category

order_arrow :: orders:string(3)  -> orders:string(3)

Category supplier_ext

fd2ext_1 :: 1:integer(1)  -> Smith:string(3)
fd2ext_2 :: 2:integer(1)  -> Jones:string(3)

Category part_ext
```

```
fd4ext_1 :: 1:integer(1)  -> brick:string(3)
fd4ext_2 :: 2:integer(1)  -> plank:string(3)

Category orders_ext

order_ext_arrow1 :: order1:string(3)  -> order1:string(3)
order_ext_arrow2 :: order2:string(3)  -> order2:string(3)



Category subs_cat

subs_arrow :: subs:string(3)  -> saddr:string(3)

Category new_supp_int

parts_1_arrow :: supp_no:integer(100)  -> supp_name:string(3)
```

2. Three functors are defined. The first two are the functors between the supplier intension and extension and the part intension and extension respectively. The third functor then defines the intension functor from the original supplier intension to the new supplier intension for the natural transformation.

```
|: for each f in pm_functor print_functor(f);

Correctly formed imperative.



Functor supplier_funct from Category supplier to Category
supplier_ext
  [ fd2 :: supp_no:integer(1) -> supp_name:string(3)  ] ==> [
fd2ext_1 :: 1:integer(1) -> Smith:string(3)  ]
  [ fd2 :: supp_no:integer(1) -> supp_name:string(3)  ] ==> [
fd2ext_2 :: 2:integer(1) -> Jones:string(3)  ]

Functor part_funct from Category part to Category part_ext
```

```
  [ fd4 :: part_no:integer(1) -> part_name:string(3)  ] ==> [
fd4ext_1 :: 1:integer(1) -> brick:string(3)  ]
  [ fd4 :: part_no:integer(1) -> part_name:string(3)  ] ==> [
fd4ext_2 :: 2:integer(1) -> plank:string(3)  ]

Functor intension_functor from Category supplier to Category
new_supp_int
  [ fd2 :: supp_no:integer(1) -> supp_name:string(3)  ] ==> [
parts_1_arrow :: supp_no:integer(100) -> supp_name:string(3)  ]
```

3. Next, we print the pullbacks. Again, there are two, one for the intension pullback and one for the extension. The intension is correctly identified as having a one-to-one relationship (as defined in the entity-relationship model), and the extension pullback is identified as a one-to-many relationship.

```
|: for each p in pullback print_pullback(p);

Correctly formed imperative.




Relationship supplier and part over orders_category
   <{supp_no:integer(1)  refs supplier} x {part_no:integer(1)  refs
part}>  ->  {orders:string(3)  refs orders_category}

pi_l is monic (1) and epic (mand)
pi_r is monic (1) and epic (mand)
Relationship is 1 : 1

Relationship supplier_ext and part_ext over orders_ext
   <{1:integer(1)  refs supplier_ext} x {1:integer(1)  refs
part_ext}>  ->  {order1:string(3)  refs orders_ext}
   <{1:integer(1)  refs supplier_ext} x {2:integer(1)  refs
part_ext}>  ->  {order2:string(3)  refs orders_ext}

pi_l is not monic (N) and not epic (opt)
pi_r is monic (1) and epic (mand)
Relationship is 1 : N
```

4. We then print the pullback functor, which maps between the intension and the extension pullbacks. For each of the two intension to extension ,we display the π_l arrow, π_r arrow, f arrow, g arrow and co-equaliser arrow.

```
|: for each p in pullback_functor print_pullback_functor_info(p);

Correctly formed imperative.



Functor sp_functor from Pullback sp_pullback to Pullback sp_ext

   <{supp_no:integer(1)  refs supplier} x {part_no:integer(1)  refs
part}>  ->{supp_no:integer(1)  refs supplier}
   <{1:integer(1)  refs supplier_ext} x {1:integer(1)  refs
part_ext}>  ->  {1:integer(1)  refs supplier_ext}

   <{supp_no:integer(1)  refs supplier} x {part_no:integer(1)  refs
part}>  ->{part_no:integer(1)  refs part}
   <{1:integer(1)  refs supplier_ext} x {1:integer(1)  refs
part_ext}>  ->  {1:integer(1)  refs part_ext}

   {supp_no:integer(1)  refs supplier}  ->  {orders:string(3)  refs
orders_category}
   {1:integer(1)  refs supplier_ext}  ->  {order1:string(3)  refs
orders_ext}

   {part_no:integer(1)  refs part}  ->  {orders:string(3)  refs
orders_category}
   {1:integer(1)  refs part_ext}  ->  {order1:string(3)  refs
orders_ext}

   <{supp_no:integer(1)  refs supplier} x {part_no:integer(1)  refs
part}>  -> {orders:string(3)  refs orders_category}
   <{1:integer(1)  refs supplier_ext} x {1:integer(1)  refs
part_ext}>  ->  {order1:string(3)  refs orders_ext}

   <{supp_no:integer(1)  refs supplier} x {part_no:integer(1)  refs
part}>  ->{supp_no:integer(1)  refs supplier}
```

```
   <{1:integer(1)  refs supplier_ext} x {2:integer(1)  refs
part_ext}>  ->  {1:integer(1)  refs supplier_ext}

   <{supp_no:integer(1)  refs supplier} x {part_no:integer(1)  refs
part}>  ->{part_no:integer(1)  refs part}
   <{1:integer(1)  refs supplier_ext} x {2:integer(1)  refs
part_ext}>  ->  {2:integer(1)  refs part_ext}

   {supp_no:integer(1)  refs supplier}  ->  {orders:string(3)  refs
orders_category}
   {1:integer(1)  refs supplier_ext}  ->  {order2:string(3)  refs
orders_ext}

   {part_no:integer(1)  refs part}  ->  {orders:string(3)  refs
orders_category}
   {2:integer(1)  refs part_ext}  ->  {order2:string(3)  refs
orders_ext}

   <{supp_no:integer(1)  refs supplier} x {part_no:integer(1)  refs
part}>  ->{orders:string(3)  refs orders_category}
   <{1:integer(1)  refs supplier_ext} x {2:integer(1)  refs
part_ext}>  ->  {order2:string(3)  refs orders_ext}
|: |:
```

5. Our co-product display simply involves making a new intension category, merging the previous supplier category with the *subs* category, and ensuring that the arrow which provides the mapping between the two is also created. Our co-products are intensional concepts. Future work will look at the production of a corresponding extension. It is debatable whether the implementation of the extension would be on the newly created co-product as an abstraction, or the individual component categories summed together. Again, for our display of the categories, we have removed extraneous information.

```
|: for each c in co_product make_sub_class(c);

Correctly formed imperative.
```

```
|: |: for each c in category print_category(c);

Correctly formed imperative.



Category subs_cat

subs_arrow :: subs:string(3)  -> saddr:string(3)

Category subs_product

fd2 :: supp_no:integer(1)  -> supp_name:string(3)
subs_arrow :: subs:string(3)  -> saddr:string(3)
subs_product :: supp_no:integer(1)  -> subs:string(3)
|: |:
```

6. Finally, we display the natural transformation for the previously defined query. That is we select all suppliers with supplier number 1, by restricting the intension attribute *supp_no* to have the restricted type of all integers between (in this case) 1 and 1, i.e. 1.

   The output shows the type tests which are performed. Note that the test *X = 2* fails because it is out of the range defined. The only *supp_no* objects that are allowed in the extension are those with the value 1. The code given below shows three type tests, which basically test the two types in the original category to see which ones match the query. Testing arrow *1 -> Smith* succeeds for both source and target of the arrow, therefore this instance is reproduced in the result of the query - shown by the tests *X=1 : integer | 1 <= X <= 1* and *X = Smith | base type = string*. For the arrow 2 -> *Jones*, the test *X = 2* fails because 2 is not in the range *1 <= X <= 1*, therefore we do not need to test Jones, and this arrow is not reproduced in the resulting category.

When mapping between arrows in the source and target intension categories, a corresponding mapping is performed on the extension. The appropriate values for attributes for the instances, where the constraint *supp_no = 1* applies, are stored in the new target extension with a mapping to them from the source extension. Instances which do not match this constraint are, of course, not held in the target extension.

Finally, to display the natural transformation, we re-display all of the functors (we have removed the functor *part_funct* from the output because it plays no part in the query). The four functors displayed, shown in the commuting square in figure 7-3, make up the natural transformation (the two previously defined to create the natural transformation, and the two which have been created as the result of the query). We do not display the new category as the information about the arrows that this category contains are shown in the targets of the two new functors. However the new target extension could be displayed to show the resulting instances that the query has produced.

```
|: for each n in nat_trans perform_closed_nat_trans(n);

Correctly formed imperative.



Testing type integer in type range integer
Base types match
Testing X = 1 : integer | 1 <= X <= 1
Testing type string in type range string
Base types match
X = Smith | base type = string
Testing type integer in type range integer
Base types match
Testing X = 2 : integer | 1 <= X <= 1
|: |:
|: for each f in pm_functor print_functor(f);

Correctly formed imperative.
```

```
Functor supplier_funct from Category supplier to Category
supplier_ext
  [ fd2 :: supp_no:integer(1) -> supp_name:string(3)  ] ==> [
fd2ext_1 :: 1:integer(1) -> Smith:string(3)  ]
  [ fd2 :: supp_no:integer(1) -> supp_name:string(3)  ] ==> [
fd2ext_2 :: 2:integer(1) -> Jones:string(3)  ]


Functor intension_functor from Category supplier to Category
new_supp_int
  [ fd2 :: supp_no:integer(1) -> supp_name:string(3)  ] ==> [
parts_1_arrow :: supp_no:integer(100) -> supp_name:string(3)  ]

Functor supplier_funct_~ from Category new_supp_int to Category
supplier_ext_~
  [ parts_1_arrow :: supp_no:integer(100) -> supp_name:string(3)  ]
==> [ fd2ext_1_~ :: 1:integer(100) -> Smith:string(3)  ]

Functor intension_functor_ext from Category supplier_ext to Category
supplier_ext_~
  [ fd2ext_1 :: 1:integer(1) -> Smith:string(3)  ] ==> [ fd2ext_1_~
:: 1:integer(100) -> Smith:string(3)  ]
|: |:
```

7.5 More Complex Test - Product Model Example

Our second example is the example given in section 5-10, where we illustrate a more complex query based on a sequence of natural transformations. The query in natural language is '*What are the names and identifiers of suppliers with capitalization greater than one million pounds who supply an electrical part with voltage rating of 90 volts?*'

To illustrate this example in detail we will compare how the query would be performed using SQL, P/FDM and our categorical system.

7.5.1 Query in SQL

The tables we would construct in a relational system would be as follows...

Suppliers

sno	sname	saddress	no_shares	price	capitalization
S1	WH Smith	Newcastle	150,000,000	6.64	996,000,000
S2	British Gas	London	800,000,000	5.60	4,480,000,000
S3	Philips	Eindhoven	125,000,000	5.25	656,250,000
S4	Osram	Birmingham	42,000,000	3.30	138,600,000
S5	Westinghouse	Pennsylvania	56,000,000	16.25	910,000,000

Parts

pno	pname	size	weight
P1	Gas fire	regular	10,000
P2	Guardian	daily	300
P3	Light bulb	bright	15
P4	Light bulb	standard	13
P5	Transformer	small	2,000,000

Orders

sno	pno	order
S1	P1	2
S2	P1	6
S2	P1	7
S2	P1	8
S3	P3	4
S3	P3	5
S4	P3	9
S5	P5	12
S5	P4	9

Electricals

pno	voltage	capacity
P3	100	230

P4	90	150
P5	2,000,000	416,000

The SQL for the query would be...

```
SELECT Suppliers.sno, Suppliers.sname
FROM Suppliers, Orders, INT5
WHERE
      Suppliers.sno = Orders.sno AND
      Electricals.pno = Orders.pno AND
      Electricals.voltage = 90 AND
      Suppliers.capitalization > 1000000
```

The result would be...

sno	sname
S5	Westinghouse

7.5.2 Query in P/FDM

We now show the DAPLEX schema that would be defined for this example...

```
declare suppliers ->> entity
      declare sno(suppliers) -> string
      declare sname(suppliers) -> string
      declare saddress(suppliers) -> string
      declare no_shares(suppliers) -> integer
      declare price(suppliers) -> integer

declare parts ->> entity
      declare pno(parts) -> string
      declare pname(parts) -> string
      declare size(parts) -> integer
      declare weight(parts) -> integer

declare orders(suppliers, parts) ->> integer
```

```
declare elec_parts ->> parts
      declare voltage(elec_parts) -> integer
      declare capitalization(elec_parts) -> integer;

define capitalization(s in suppliers) -> integer in example2
      share_price(s) * no_shares(s);
```

And our query would be constructed as follows...

```
for each s in suppliers
      such that capitalization(s) > 1000000
      and for each o in orders(s, p) such that
            voltage(p as elec_parts) = 90
print(sno(s), sname(s));
```

with the result being a display of the row shown in the SQL example.

7.5.3 Query in the Product Model

The query in the Product Model requires considerable initial work. There are five functors which need to be defined, one of which operates over a pullback - which we do not have the capability to do properly at the moment. A method for doing this could be implemented by converting the original pullbacks to categories and then using the existing natural transformation code, although this would mean the result would always be a simple category and not a pullback. We will therefore produce one of the natural transformations, but as well as this we hope to demonstrate that dynamic arrows are possible in the model.

The natural transformation we choose is equivalent to the composition of parts X4 and X5 in the example given in section 5.10.1, where we basically return the arrow f_5 with its new types. We also have to alter the query to ensure that satisfactory output is created. If the query was kept as it is, then the natural transformation would in fact succeed for every instance in the extension. We also note a problem with the

implementation of the model here: if we returned more than one set of arrows for each instance in the extension, then the instances returned may not be compatible, that is the arrows returned for each instance would depend on whether the tests succeeded for each arrow, and not if all tests succeed for all arrows. This is a problem with the definition that we use for extension categories. We store all values for all instances in one category, rather than storing one instance per category, that is all instances within the database for any intension category are stored within one extension category. Therefore a query could incidentally return arrows which should not be returned. A solution to the problem is described in section 8.3.

Sample output is shown below:

1. Again, first of all we display all of the categories. Note, the category *INT5* is not shown as it is not relevant in this query. (Note however, we do produce the pullback in the example).

```
| ?- daplex.
|: for each c in category print_category(c);

Correctly formed imperative.




Category int1

f1 :: sno:string(3)  -> sname:string(3)
f2 :: sno:string(3)  -> saddress:string(3)
f3 :: sno:string(3)  -> no_shares:integer(1)
f4 :: sno:string(3)  -> price:float(2)
f5 :: no_shares:integer(1) price:float(2)  ->
capitalization:integer(1)

Category int2

f6 :: pno:string(3)  -> pname:string(3)
```

```
f7 :: pno:string(3)  -> size:string(3)
f8 :: pno:string(3)  -> weight:integer(1)
```

Category int4

```
f9 :: oid:integer(1)  -> voltage:integer(1)
f10 :: oid:integer(1)  -> capacity:integer(1)
```

Category ord1

```
o1 :: order:integer(1)  -> order:integer(1)
```

Category Suppliers

```
f1a :: S1:string(3)  -> WH Smith:string(3)
f1b :: S2:string(3)  -> British Gas:string(3)
f1c :: S3:string(3)  -> Philips:string(3)
f1d :: S4:string(3)  -> Osram:string(3)
f1e :: S5:string(3)  -> Westinghouse:string(3)
f2a :: S1:string(3)  -> Newcastle:string(3)
f2b :: S2:string(3)  -> London:string(3)
f2c :: S3:string(3)  -> Eindhoven:string(3)
f2d :: S4:string(3)  -> Birmingham:string(3)
f2e :: S5:string(3)  -> Pennsylvania:string(3)
f3a :: S1:string(3)  -> 150000000:integer(1)
f3b :: S2:string(3)  -> 800000000:integer(1)
f3c :: S3:string(3)  -> 125000000:integer(1)
f3d :: S4:string(3)  -> 42000000:integer(1)
f3e :: S5:string(3)  -> 56000000:integer(1)
f4a :: S1:string(3)  -> 6.64:float(2)
f4b :: S2:string(3)  -> 5.6:float(2)
f4c :: S3:string(3)  -> 5.25:float(2)
f4d :: S4:string(3)  -> 3.3:float(2)
f4e :: S5:string(3)  -> 16.25:float(2)
f5a :: 150000000:integer(1) 6.64:float(2)  -> 996000000:integer(1)
f5b :: 800000000:integer(1) 5.6:float(2)  -> 4480000000:integer(1)
f5c :: 125000000:integer(1) 5.25:float(2)  -> 656250000:integer(1)
f5d :: 42000000:integer(1) 3.3:float(2)  -> 138600000:integer(1)
f5e :: 56000000:integer(1) 16.25:float(2)  -> 910000000:integer(1)
```

Category Parts

```
f6a :: P1:string(3)  -> Gas fire:string(3)
f6b :: P2:string(3)  -> Guardian:string(3)
f6c :: P3:string(3)  -> Light bulb:string(3)
f6d :: P4:string(3)  -> Light bulb:string(3)
f6e :: P5:string(3)  -> Transformer:string(3)
f7a :: P1:string(3)  -> regular:string(3)
f7b :: P2:string(3)  -> daily:string(3)
f7c :: P3:string(3)  -> bright:string(3)
f7d :: P4:string(3)  -> standard:string(3)
f7e :: P5:string(3)  -> small:string(3)
f8a :: P1:string(3)  -> 10000:integer(1)
f8b :: P2:string(3)  -> 300:integer(1)
f8c :: P3:string(3)  -> 15:integer(1)
f8d :: P4:string(3)  -> 13:integer(1)
f8e :: P5:string(3)  -> 2000000:integer(1)

Category Orders

o1a :: 2:integer(1)  -> 2:integer(1)
o1b :: 6:integer(1) 7:integer(1) 8:integer(1)  -> 6:integer(1)
7:integer(1) 8:integer(1)
o1c :: 4:integer(1) 5:integer(1)  -> 4:integer(1) 5:integer(1)
o1d :: 9:integer(1)  -> 9:integer(1)
o1e :: 12:integer(1)  -> 12:integer(1)
o1f :: 9:integer(1)  -> 9:integer(1)

Category Int4ext

f9a :: 1:integer(1)  -> 100:integer(1)
f9b :: 2:integer(1)  -> 90:integer(1)
f9c :: 3:integer(1)  -> 2000000:integer(1)
f10a :: 1:integer(1)  -> 230:integer(1)
f10b :: 2:integer(1)  -> 150:integer(1)
f10c :: 3:integer(1)  -> 416000:integer(1)

Category INT1p

f5p :: no_shares:integer(1) price:float(2)  ->
capitalization:integer(5)
|: |:
```

2. Next, we display the functors between intension and extensions, and also the *intension_functor1* functor, which again provides the definition of the natural transformation we are going to produce.

```
|: for each f in pm_functor print_functor(f);

Correctly formed imperative.




Functor suppliers_funct from Category int1 to Category Suppliers
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1a ::
S1:string(3) -> WH Smith:string(3)  ]
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1b ::
S2:string(3) -> British Gas:string(3)  ]
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1c ::
S3:string(3) -> Philips:string(3)  ]
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1d ::
S4:string(3) -> Osram:string(3)  ]
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1e ::
S5:string(3) -> Westinghouse:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2a ::
S1:string(3) -> Newcastle:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2b ::
S2:string(3) -> London:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2c ::
S3:string(3) -> Eindhoven:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2d ::
S4:string(3) -> Birmingham:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2e ::
S5:string(3) -> Pennsylvania:string(3)  ]
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3a ::
S1:string(3) ->150000000:integer(1)  ]
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3b ::
S2:string(3) ->800000000:integer(1)  ]
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3c ::
S3:string(3) ->125000000:integer(1)  ]
```

```
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3d ::
S4:string(3) ->42000000:integer(1)  ]
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3e ::
S5:string(3) ->56000000:integer(1)  ]
  [ f4 :: sno:string(3) -> price:float(2)  ] ==> [ f4a ::
S1:string(3) -> 6.64:float(2)  ]
  [ f4 :: sno:string(3) -> price:float(2)  ] ==> [ f4b ::
S2:string(3) -> 5.6:float(2)  ]
  [ f4 :: sno:string(3) -> price:float(2)  ] ==> [ f4c ::
S3:string(3) -> 5.25:float(2)  ]
  [ f4 :: sno:string(3) -> price:float(2)  ] ==> [ f4d ::
S4:string(3) -> 3.3:float(2)  ]
  [ f4 :: sno:string(3) -> price:float(2)  ] ==> [ f4e ::
S5:string(3) -> 16.25:float(2)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5a :: 150000000:integer(1)
6.64:float(2) -> 996000000:integer(1)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5b :: 800000000:integer(1)
5.6:float(2) -> 4480000000:integer(1)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5c :: 125000000:integer(1)
5.25:float(2) -> 656250000:integer(1)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5d :: 42000000:integer(1)
3.3:float(2) -> 138600000:integer(1)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5e :: 56000000:integer(1)
16.25:float(2) -> 910000000:integer(1)  ]

Functor parts_funct from Category int2 to Category Parts
  [ f6 :: pno:string(3) -> pname:string(3)  ] ==> [ f6a ::
P1:string(3) -> Gas fire:string(3)  ]
  [ f6 :: pno:string(3) -> pname:string(3)  ] ==> [ f6b ::
P2:string(3) -> Guardian:string(3)  ]
  [ f6 :: pno:string(3) -> pname:string(3)  ] ==> [ f6c ::
P3:string(3) -> Light bulb:string(3)  ]
  [ f6 :: pno:string(3) -> pname:string(3)  ] ==> [ f6d ::
P4:string(3) -> Light bulb:string(3)  ]
  [ f6 :: pno:string(3) -> pname:string(3)  ] ==> [ f6e ::
P5:string(3) -> Transformer:string(3)  ]
```

```
  [ f7 :: pno:string(3) -> size:string(3)  ] ==> [ f7a ::
P1:string(3) -> regular:string(3)  ]
  [ f7 :: pno:string(3) -> size:string(3)  ] ==> [ f7b ::
P2:string(3) -> daily:string(3)  ]
  [ f7 :: pno:string(3) -> size:string(3)  ] ==> [ f7c ::
P3:string(3) -> bright:string(3)  ]
  [ f7 :: pno:string(3) -> size:string(3)  ] ==> [ f7d ::
P4:string(3) -> standard:string(3)  ]
  [ f7 :: pno:string(3) -> size:string(3)  ] ==> [ f7e ::
P5:string(3) -> small:string(3)  ]
  [ f8 :: pno:string(3) -> weight:integer(1)  ] ==> [ f8a ::
P1:string(3) -> 10000:integer(1)  ]
  [ f8 :: pno:string(3) -> weight:integer(1)  ] ==> [ f8b ::
P2:string(3) -> 300:integer(1)  ]
  [ f8 :: pno:string(3) -> weight:integer(1)  ] ==> [ f8c ::
P3:string(3) -> 15:integer(1)  ]
  [ f8 :: pno:string(3) -> weight:integer(1)  ] ==> [ f8d ::
P4:string(3) -> 13:integer(1)  ]
  [ f8 :: pno:string(3) -> weight:integer(1)  ] ==> [ f8e ::
P5:string(3) -> 2000000:integer(1)  ]

Functor intension_functor1 from Category int1 to Category INT1p
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5p :: no_shares:integer(1)
price:float(2) -> capitalization:integer(5)  ]
|: |:
```

3. Now, we display the intension and extension pullback. Note that the extension pullback has been identified as many-to-many.

```
|: for each p in pullback print_pullback(p);

Correctly formed imperative.




Relationship supplier and part over order
   <{sno:string(3)  refs int1} x {pno:string(3)  refs int2}>  ->
{order:integer(1)  refs ord1}
```

```
pi_l is monic (1) and epic (mand)
pi_r is monic (1) and epic (mand)
Relationship is 1 : 1

Relationship Suppliers and Parts over Orders
   <{S1:string(3)  refs Suppliers} x {P2:string(3)  refs Parts}>  ->
{2:integer(1)  refs Orders}
   <{S2:string(3)  refs Suppliers} x {P1:string(3)  refs Parts}>  ->
{6:integer(1) 7:integer(1) 8:integer(1)  refs Orders}
   <{S3:string(3)  refs Suppliers} x {P3:string(3)  refs Parts}>  ->
{4:integer(1) 5:integer(1)  refs Orders}
   <{S4:string(3)  refs Suppliers} x {P3:string(3)  refs Parts}>  ->
{9:integer(1)  refs Orders}
   <{S5:string(3)  refs Suppliers} x {P5:string(3)  refs Parts}>  ->
{12:integer(1)  refs Orders}
   <{S5:string(3)  refs Suppliers} x {P4:string(3)  refs Parts}>  ->
{9:integer(1)  refs Orders}

pi_l is not monic (N) and epic (mand)
pi_r is not monic (N) and epic (mand)
Relationship is N : N
|: |:
```

4. Now we display the pullback functor. This time we display it in simplified form, that is we only show the co-equaliser arrows.

```
|: for each p in pullback_functor print_pullback_functor(p);

Correctly formed imperative.



Functor ordpb_funct from Pullback ass1 to Pullback Int3ext

   <{sno:string(3)  refs int1} x {pno:string(3)  refs int2}>  ->
{order:integer(1)  refs ord1}
 ==>
   <{S1:string(3)  refs Suppliers} x {P2:string(3)  refs Parts}>  ->
{2:integer(1)  refs Orders}
```

```
   <{sno:string(3)  refs int1} x {pno:string(3)  refs int2}>  ->
{order:integer(1)  refs ord1}
 ==>
   <{S2:string(3)  refs Suppliers} x {P1:string(3)  refs Parts}>  ->
{6:integer(1) 7:integer(1) 8:integer(1)  refs Orders}

   <{sno:string(3)  refs int1} x {pno:string(3)  refs int2}>  ->
{order:integer(1)  refs ord1}
 ==>
   <{S3:string(3)  refs Suppliers} x {P3:string(3)  refs Parts}>  ->
{4:integer(1) 5:integer(1)  refs Orders}

   <{sno:string(3)  refs int1} x {pno:string(3)  refs int2}>  ->
{order:integer(1)  refs ord1}
 ==>
   <{S4:string(3)  refs Suppliers} x {P3:string(3)  refs Parts}>  ->
{9:integer
(1)  refs Orders}

   <{sno:string(3)  refs int1} x {pno:string(3)  refs int2}>  ->
{order:integer(1)  refs ord1}
 ==>
   <{S5:string(3)  refs Suppliers} x {P5:string(3)  refs Parts}>  ->
{12:integer(1)  refs Orders}

   <{sno:string(3)  refs int1} x {pno:string(3)  refs int2}>  ->
{order:integer(1)  refs ord1}
 ==>
   <{S5:string(3)  refs Suppliers} x {P4:string(3)  refs Parts}>  ->
{9:integer(1)  refs Orders}
|: |:
```

5. Finally, we show our chosen part of the query. We choose to return the suppliers with a capitalization greater than one billion (in the complete query the value is one million, but this actually returns all tuples). Again, we have removed the parts functor from the output because it is irrelevant for the query. The result of the query can be found as the target category of either the *suppliers_funct_~* or the *intension_functor1_ext* functors given at the end of the output given below:

```
|: for each n in nat_trans perform_closed_nat_trans(n);

Correctly formed imperative.



Testing type integer in type range integer
Base types match
X = 150000000 | base type = integer
Testing type integer in type range integer
Base types match
Testing X = 996000000 : integer | X >= 1000000000
Testing type integer in type range integer
Base types match
X = 800000000 | base type = integer
Testing type integer in type range integer
Base types match
Testing X = 4480000000 : integer | X >= 1000000000
Testing type float in type range float
Base types match
X = 5.6 | base type = float
Testing type integer in type range integer
Base types match
Testing X = 4480000000 : integer | X >= 1000000000
Testing type integer in type range integer
Base types match
X = 125000000 | base type = integer
Testing type integer in type range integer
Base types match
Testing X = 656250000 : integer | X >= 1000000000
Testing type integer in type range integer
Base types match
X = 42000000 | base type = integer
Testing type integer in type range integer
Base types match
Testing X = 138600000 : integer | X >= 1000000000
Testing type integer in type range integer
Base types match
X = 56000000 | base type = integer
Testing type integer in type range integer
```

```
Base types match
Testing X = 910000000 : integer | X >= 1000000000
|: |:
|: for each f in pm_functor print_functor(f);

Correctly formed imperative.
```

```
Functor suppliers_funct from Category int1 to Category Suppliers
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1a ::
S1:string(3) -> WH Smith:string(3)  ]
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1b ::
S2:string(3) -> British Gas:string(3)  ]
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1c ::
S3:string(3) -> Philips:string(3)  ]
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1d ::
S4:string(3) -> Osram:string(3)  ]
  [ f1 :: sno:string(3) -> sname:string(3)  ] ==> [ f1e ::
S5:string(3) -> Westinghouse:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2a ::
S1:string(3) -> Newcastle:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2b ::
S2:string(3) -> London:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2c ::
S3:string(3) -> Eindhoven:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2d ::
S4:string(3) -> Birmingham:string(3)  ]
  [ f2 :: sno:string(3) -> saddress:string(3)  ] ==> [ f2e ::
S5:string(3) -> Pennsylvania:string(3)  ]
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3a ::
S1:string(3) ->150000000:integer(1)  ]
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3b ::
S2:string(3) ->800000000:integer(1)  ]
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3c ::
S3:string(3) ->125000000:integer(1)  ]
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3d ::
S4:string(3) ->42000000:integer(1)  ]
  [ f3 :: sno:string(3) -> no_shares:integer(1)  ] ==> [ f3e ::
S5:string(3) ->56000000:integer(1)  ]
```

```
  [ f4 :: sno:string(3) -> price:float(2)  ] ==> [ f4a ::
S1:string(3) -> 6.64:float(2)  ]
  [ f4 :: sno:string(3) -> price:float(2)  ] ==> [ f4b ::
S2:string(3) -> 5.6:float(2)  ]
  [ f4 :: sno:string(3).-> price:float(2)  ] ==> [ f4c ::
S3:string(3) -> 5.25:float(2)  ]
  [ f4 :: sno:string(3) -> price:float(2)  ] ==> [ f4d ::
S4:string(3) -> 3.3:float(2)  ]
  [ f4 :: sno:string(3) -> price:float(2)  ] ==> [ f4e ::
S5:string(3) -> 16.25:float(2)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5a :: 150000000:integer(1)
6.64:float(2) -> 996000000:integer(1)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5b :: 800000000:integer(1)
5.6:float(2) -> 4480000000:integer(1)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5c :: 125000000:integer(1)
5.25:float(2) -> 656250000:integer(1)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5d :: 42000000:integer(1)
3.3:float(2) -> 138600000:integer(1)  ]
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5e :: 56000000:integer(1)
16.25:float(2) -> 910000000:integer(1)  ]


Functor intension_functor1 from Category int1 to Category INT1p
  [ f5 :: no_shares:integer(1) price:float(2) ->
capitalization:integer(1)  ] ==> [ f5p :: no_shares:integer(1)
price:float(2) -> capitalization:integer(5)  ]

Functor suppliers_funct_~ from Category INT1p to Category Suppliers_~
  [ f5p :: no_shares:integer(1) price:float(2) ->
capitalization:integer(5)  ] ==> [ f5b_~ :: 800000000:integer(1)
5.6:float(2) -> 4480000000:integer(5)  ]

Functor intension_functor1_ext from Category Suppliers to Category
Suppliers_~
```

```
[ f5b :: 800000000:integer(1) 5.6:float(2) -> 4480000000:integer(1)
] ==> [ f5b_~ :: 800000000:integer(1) 5.6:float(2) ->
4480000000:integer(5)  ]
|: |: ;

yes
```

7.6 Conclusions

The sample queries show that the implementation provides satisfactory:

- representation of complex data;
- data definition facilities with categorical constructions;
- sound control of typing;
- representation of intension-extension functors in a natural manner;
- inheritance facilities at the intension level
- construction of a natural transformation;
- presentation and verification of data.

A number of problems have also arisen:

- the user interface is cumbersome for the purposes of data definition, data addition and query definition. In particular the creation of concepts such as the *os_position* entities could be automated, and input could be improved if some of the indirection in representation of the entities was removed. A graphical interface shall be the eventual aim for query construction, as previously discussed in section 6.4. Queries are constructed by arrow composition, and categories and their mappings are best viewed diagrammatically, so a graphical interface would allow the building of queries by composition of arrows and categories, which relates directly to categorical methods of diagram chasing for composing arrows. However, operations such as aggregation would be difficult to implement graphically, so some compromise would be needed;

- inheritance, while represented perfectly adequately in the intension, is not effective for extensional purposes;
- extensional values are not held quite as simply as the Product Model suggests, the identification of a collection of values for a particular instance requires further thought.

Our sample queries show that there is considerable scope for further work into a simple method of defining queries, but as this thesis concentrates on a prototype of the formal basis, we believe any further work is outside the scope of this thesis, and is something which we could consider more closely at a later date.

8. Discussion and Conclusions

The objective of this thesis was to develop a formal model in terms of category theory for the functionality of current object-relational databases, and to demonstrate the functionality of such a model by implementing a prototype.

We have demonstrated that it is possible to develop a feasible formal model for object-relational features, using the formal notation of category theory. This model has been able to represent in an universal manner the constructs required for object based database models. We have also demonstrated that it is possible to implement such a model, although we have highlighted a number of difficulties in such an implementation.

In particular, we believe that the main achievements of our work are:

1. representing, with category theory, in a universal formal manner, object-relational abstractions and intension-extension concepts;
2. implementing, with the functional data model database system P/FDM, categorical constructions.

We will now discuss these in more detail.

8.1 Modelling Aspects

We have shown that category theory is a very powerful tool for representing the important constructs of object based databases, and has significant advantages over set theoretical models for multi-level formalisms. In particular, we stress the ease with which the arrow can be used to model static and dynamic behaviour of constructs, how categories can model arrows, both intensionally and extensionally, and how functors provide a natural manner of producing a consistent database. Products are an excellent

way of defining relationships. Single inheritance at the intension level has been achieved with co-products. Finally, natural transformations are a powerful construct for modelling queries and views. A real practical use for category theory in the area of databases has been demonstrated in that queries with closure and views can only be effected with intension-extension natural transformations or constructions of equivalent power.

We have closure in a natural and consistent manner. We believe this has been implemented more clearly in the form of natural transformations than in functional models, as our closure applies to both the intensional and extensional levels. This form of closure has been difficult to achieve in standard object-oriented databases because it involves automatically redefining class definitions [cha94]. Our closure gives two important benefits, that of being able to perform queries and being able to provide views, where the resulting structures can be manipulated further as if they were part of the original database.

We have produced a formal model of an object-based database using the constructs of category theory, incorporating what we believe to be the best features of object-relational and functional database models to produce a database model that can cope with the demands of the future. In fact, it is interesting to note the similarity of our model with early extended relational systems, such as the nested-relational model [rot88], and in particular RM/T [cod79]. RM/T is based on entities which model both properties and relationships, which can be thought of as similar to our use of categories and products (where products are just advanced kinds of categories, that is they contain arrows as their primary structure). We believe this comparison gives an excellent backbone for object-relational databases, which may otherwise have had difficulties in coping with object concepts in an otherwise set-theoretic model.

8.2 Implementation Areas

The system that has been developed is a prototype, that is it is in no way complete or perfect. For instance its input and output system requires much improvement, the representation of inheritance needs more work, extensional categories need to be more clearly defined in the model, and the system does not cope completely with dynamic structures, for example methods. However, hopefully the prototype has provided enough insight into how a full categorical system could be implemented in the future. In particular, the previous chapter on testing has highlighted many difficulties with the prototype in how data is input to the system, how queries are defined and how results are displayed. Construction of a full system would require extensive work on a data definition and manipulation language, and we believe that success would be best achieved by basing it on DAPLEX. Our prototype was successful due to many of the benefits of P/FDM, which shows the strength of functional and semantic data models.

The tests also lead us to believe that graphical user interfaces would be an excellent method of representing categorical database. One of the most powerful tools of category theory is diagram chasing for making proofs, and many of our tests are based on diagram chasing and testing arrows, something which graphical applications can do excellently.

We also believe that we have produced a practical new formalism for object models, not just another implementation of functional models. Although our system uses a functional database model for the implementation base, this does not necessarily imply that our system is just a functional database, our model has only used functional databases as the physical database storage structure. We have suggested that the object-relational model is an excellent attempt at modelling the functionality required for future database needs, however current systems are cumbersome and lacking in a strong theoretical basis. We therefore believe that our work improves on this situation by using a more natural theoretical basis for these future needs.

We also believe that the implementation has been largely successful. The prototype demonstrated many of the problems that would arise in the development of a complete system, but our prototype has managed to model natural transformations with closure, one of the main aims of this thesis. The performance of our model should be comparable to that of any database developed in P/FDM. All of our concepts map down to P/FDM structures which ensure therefore that we use its hashing system. However some of the consistency tests would probably have poor performance due to the formality of the arrow tests. Therefore it is better to ensure, as we did in the prototype, that all objects added to the system conform to the strict typing constraints. We believe that having a successful implementation that very closely follows the model is also beneficial in the design of object-relational database systems, and the prototype shows the scope for use of object-based database systems.

8.3 Problem Areas and Future Thoughts

As already stated, we believe the implementation in this thesis and the tests show that there is great scope for future development of categorical database systems, and future research into using category theory for modelling database systems, in particular keeping up with the changes in newer generation object models, and in extending our work to handle heterogeneous systems.

The model could be brought to a higher level of abstraction using the categorical concepts of adjoints, allegories, topoi, and pushouts, which appear to be appropriate constructions for more complex data modelling. However, implementation of these is likely to be even more difficult than the categorical concepts which have been implemented in this model.

Specific problems of the model were:

- co-products can not adequately model multiple inheritance, pushouts seem to be a better choice in any future model. The co-product structure discussed in section

5.4.3 is only able to handle single inheritance. A pushout is a co-product restricted over some further object. In figure 8-1 below, the object **C** would be the new properties of the specialisation, **A** and **B** would be the two objects that are being inherited from, and the disjoint union **A** + c **B** would be the new specialised object. It is easy to see that these structures can again be composed to form inheritance hierarchies.

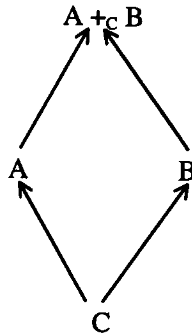


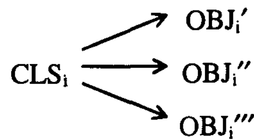
Figure 8-1 : Use of pushouts to model multiple inheritance

A pushout can be regarded as a sum in the context of other information while a coproduct is an unrestricted sum. The pushout is the more general concept as, if the object **C** is replaced by the universal object {*}, the pushout becomes in effect a coproduct;

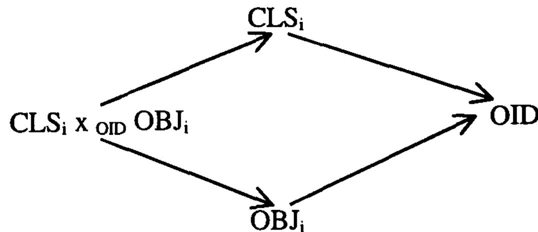
- The mapping between intensional and extensional categories to represent the abstraction of inheritance needs further thought. Inheritance has been represented in our model within the intension category **INT** as a collection of coproduct categories **UNI**. The functor D relates the coproduct categories to a collection of extensional categories held within the category **EXT**. Further work needs to be performed on exactly how D will relate the intensional coproduct and extensional categories. For instance will only the leaf-nodes, that is the terminal specialisations, in the inheritance hierarchy actually hold data? Or will every node in the inheritance hierarchy hold data?

Such further work should commence with a detailed investigation of all the types of inheritance from a semantic viewpoint. Various categorical constructions, representing the different types at the intensional level, could then be developed. The final stage would involve a study of the mappings required, between the intensional and extensional constructions, to provide effective storage and manipulation;

- the model needs to be revised to deal properly with extensional objects. The essential problem is that our model treats the extension as a single category instance whereas it is in fact a collection of category instances, with an initial object to link the instances. Referring back to section 5.6, the mapping *V* between classes and objects could be represented by the construction:



Thus associated with each class category will be a collection of object categories. CLS_i is the initial object in the construction as there is an arrow from it to every object instance. Such a structure can be viewed as a topos, in the form of a pullback:



where OID is the object identifier associated with each class/object pair.

Specific problems of the implementation included:

- the representation of extensional objects needs more careful examination;
- natural transformations need to be defined in a manner which makes them more universal, currently they do not cope with pullback categories;
- the implementation can currently only handle static concepts, the implementation of arrows could be revised to handle dynamic concepts such as methods;
- the interface needs improvement, both in terms of defining and entering data and in terms of querying, which may be best handled with some sort of graphical interface;
- the use of pullbacks needs to be extended to deal with *n*-ary relationships rather than just binary as currently supported;
- a more sophisticated type system with associated type tests is needed; the implementation can currently only handle simplistic type tests such as number ranges and lexicographic ordering of strings
- the current implementation does not allow views and update facilities, again the treatment of natural transformations may need revised;
- the representation of the internal properties in a category needs revised to match the definition in the model, rather than just storing a set of arrows.

However, we still believe that P/FDM was an excellent choice as an implementation base, although for development of a production system there may be other choices. The algebraic specification languages are appealing, but perhaps again they would not be ideal for developing a practical system. Changing the model so that categories made more extensive use of finite products would then ensure that the model could be implemented in functional languages. Categories would then be cartesian closed, which, while sufficient in theoretic terms, may be unduly restrictive for practical purposes such as modelling some complex objects with uncertain structures.

This thesis has demonstrated a comprehensive formal model which provides a practical definition of object-based database systems. The prototype shows the scope for

improving the appeal of object-based database systems, in particular the object-relational model.

9. References

[abr92] S. Abramsky, D. [Dov] Gabbay, T. S. E. Maibaum. *Handbook of Logic and Computer Science, Background: Mathematical Structures.* Clarendon Press, Oxford, 1992.

[ans94] *ISO-ANSI SQL 3 Working Draft.* Digital Equipment Corporation, Massachusetts, March 1994.

[atk83] M. Atkinson et al. An Approach to Persistent Programming. *The Computer Journal,* 26(4), 1983.

[atk90] M. Atkinson et al. The Object-Oriented Database System Manifesto. In *[ban92].*

[ban92] F. Bancilhon et al. *The Story of O_2 : Implementing an Object-Oriented Database System,* Morgan Kaufmann, 1992.

[bar84] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North Holland, 1984.

[bar90] M. Barr, C. Wells. *Category Theory for Computing Science.* Prentice-Hall International Series in Computer Science, 1990.

[bee92] C. Beeri. New Data Models and Languages: The Challenge. *Proceedings of the 11th ACM Symposium on Principles of Database Systems (PODS),* 1-15, 1992.

[ber93] E. Bertino, L. Martino. *Object-Oriented Database Systems: Concepts and Architectures.* Addison-Wesley, 1993.

[boo94] G. Booch. *Object-Oriented Analysis and Design with Applications, 2nd Edition.* Benjamin Cummings, 1994.

[bra93] K. S. Brathwaite. *Object-Oriented Database Design: Concepts and Applications.* Academic Press, 1993.

[bra90] I. Bratko. *PROLOG Programming for Artificial Intelligence, 2nd Edition.* Addison-Wesley, 1990.

[bur80] R. M. Burstall, J. A. Goguen. The Semantics of Clear: A

Specification Language. *Lecture Notes in Computer Science*, 86:292-332, 1980.

[cad96] B. Cadish, Z. Diskin. Algebraic Graph-Oriented = Category Theory Based: Manifesto of Categorizing Database Theory. Frame Inform System, Database Design Laboratory, Latvija, update of *DBDL Research Report FIS/DBDL-94-02,* December 1996.

[car84] L. Cardelli. A Semantics of Multiple Inheritance. *LNCS,* 173:51-67, 1984.

[car85] L. Cardelli, P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys,* 17(4), December 1985.

[car85b] J. Cartmell. Formalising the Network and Hierarchical Data Models: An Application of Categorical Logic. *Lecture Notes in Computer Science*, 240:466-492, 1985.

[cat94] R. G. G. Cattell. *The Object Database Standard: ODMG-93, Release 1.1.* Morgan Kaufmann, 1994.

[cha94] D. K. C. Chan, P. W. Trindler, R. C. Welland. Evaluating Object-Oriented Query Languages. *The Computer Journal,* 37(10):858-872, October 1994.

[che76] P. P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM TODS,* 1(1):9-36, March 1976.

[coa91] P. Coad, E Yourdon. *Object-Oriented Analysis, 2nd Edition.* Yourdon Press, 1991.

[cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* 13(6):377-387, June 1970.

[cod79] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems,* 4(4), December 1979.

[cos87] G. Cosineau, P. L. Curien, M. Mauny. The Categorical Abstract Machine. *Science of Computer Programming,* 203-211, 1987.

[dat95] C. J. Date. *An Introduction to Database Systems, Sixth Edition.* Addison-Wesley, 1995.

[day89] U. Dayal. Queries and Views in an Object-Oriented Data Model. *Second International Workshop on Database Programming Languages,* 1989.

[dem92] 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.

[den93] E. Dennis-Jones, D. E. Rydeheard. Categorical ML - Category Theoretic Modular Programming. *Formal Aspects of Computing,* 5(4):337-366, 1993.

[dis93] Z. Diskin. Abstract Queries, Schema Transformations and Algebraic Theories: An Application of Categorical Algebra to Database Theory. Frame Inform System, Database Design Laboratory, Latvija, *DBDL Research Report FIS/DBDL-93-02*, November 1993.

[dup94] L. Duponcheel. *Gofer Experimental Prelude.* Alcatel, Belgium, 1994.

[ehr87] H. D. Ehrich, A. Semadas, C. Semadas. Objects, Object Types and Object Identification. In: Categorical Methods in Computer Science, ed. G Ehrig, H. Herrlich, H. J. Kreowski, Preuβ, *Lecture Notes in Computer Science*, 393:142-156, 1987.

[elm94] R. Elmasri, S. B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley World Student Series, 1994.

[emb92] S. M. Embury, J. Jiao, P. M. D. Gray. Using Prolog to Provide Access to Metadata in an Object-Oriented Database. *International Conference on the Practical Application of Prolog,* April 1992.

[emb94] S. M. Embury. *Constraint-Based Updates in a Functional Data Model Database.* University of Aberdeen, PhD Thesis, April 1994..

[emb95] S. M. Embury, et al. *User Manual for P/FDM Version 9.0.* University of Aberdeen, Technical Report AUCS/TR9501, January 1995.

[fis89] D. Fishman, et al. Overview of the Iris DBMS. In *[kim89].*

[fre90] P. J. Freyd, A. Scedrov. *Categories, Allegories.* North-Holland

Mathematical Library 39, 1990.

[ghe90] G. Ghelli. Modelling Features of Object-Oriented Languages in Second Order Functional Languages with Subtypes. *FIDE Technical Report Series FIDE/90/3, Basic Research Action 3070*, 1990.

[gog86] J. A. Goguen. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. *ACM SIGPLAN Notices*, 21(10):153-162, October 1986.

[gog89] J. A. Goguen. A Categorical Manifesto. SRI International, Computer Science Laboratory, *SRI-CSL-89-08*, July 1989.

[gog92] J. A. Goguen et al. Introducing OBJ. SRI International, Computer Science Laboratory, *SRI-CLS-92-03*, March 1992.

[gol83] A. Goldberg, D. Robson. *Smalltalk-80 : The Language and its Implementation*. Addison-Wesley, 1983.

[gor95] A. D. Gordon, K. Hammond. *Monadic I/O in Haskell 1.3*. Proceedings of the Haskell Workshop, La Jolla, California, 50-68, June 1995. Also, Yale University Research Report YALEU/DCS/RR-1075, 1995.

[gra88] P. M. D. Gray, D. S. Moffat, N. W. Paton. A Prolog Interface to a Functional Data Model Database. *Lecture Notes in Computer Science*, 1988.

[gra92] P. M. D. Gray, K. G. Kulkarni, N. W. Paton. *Object-Oriented Databases: A Semantic Data Model Approach*. Prentice-Hall International Series in Computer Science, 1992.

[ham81] M. Hammer, D. McLeod. Database Description with SDM : A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351-386, September 1981.

[han91a] C. Hanson, et al. *MIT Scheme User's Manual, Draft Edition 0.9*. Massachusetts Institute of Technology, January 1991.

[han91b] C. Hanson, et al. *MIT Scheme Reference Manual, Edition 1.1*. Massachusetts Institute of Technology, November 1991.

[hew94] Hewlett Packard.

http://www.hp.com/sesd/3rd.party.1/object.database.html, 1994.

[hol87] B. J. Holmes. *Pascal Programming*. D. P. Publications, Ltd., 1987.

[hol94] S. D. Holford. *Object-Based Data Models as a Platform for Spatial Data Definition and Query Processing*. University of Newcastle upon Tyne, MSc Dissertation D605, 1994.

[hud92] P. Hudak, H. J. Fasel. A Gentle Introduction to Haskell. *SIGPLAN Notices,* 27(5), 1992.

[ing94] *Ingres Reference Manual.* Relational Technology Inc., 1994.

[jae82] G. Jaeschke, H. Schek. Remarks on the Algebra of Non First Normal Form Relations. *Proceedings of the ADM SIGACT-SIGMON Symposium on Principles of Database Systems.* Los Angeles, California, March 1982.

[jon94] P. Jones. *An Introduction to Gofer, Draft Report.* Yale University, July 1994.

[kap92] G. Kappel, A. Min Tjoa. State of Art and Open Issues on Graphical Interfaces for Object-Oriented Database Systems. *Information and Software Technology*, 34(11), November 1992.

[ker76] L. Kerschberg, J. E. S. Pacheco. *A Functional Data Base Model.* Monographs in Computer Science and Computer Applications, No. 2/76, Pontificia Universidade Catolico do Rio de Janeiro, February 1976.

[kim89] W. Kim, F. Lochovsky. *Object-Oriented Concepts, Databases and Applications.* Addison-Wesley, 1989.

[kim90] W. Kim. *Introduction of Object-Oriented Database Systems.* MIT Press, 1990.

[kim94] W. Kim. *On Object-Oriented Database Technology.* ADB Inc., 1994.

[kim94b] M. J. Kim, D. A. Nelson, B. N. Rossiter. *Evaluation of the Object-Relational DBMS Postgres I. Administrative Data.* University of Newcastle Upon Tyne, Technical Report Series, no. 500, November 1994.

[kul86] K. G. Kulkarni, M. P. Atkinson. EFDM: Extended Functional Data Model. *The Computer Journal,* 29(1), 1986.

[kul87] K. G. Kulkarni, M. P. Atkinson. Implementing an Extended Functional Data Model using PS_Algol. *Software Practice and Experience,* 17(3):171-185, 1987.

[kup93] K. M. Kuper, M. Y. Vardi. The Logical Data Model. *ACM TODS,* 18(3):379-413, 1993.

[lam91] C. Lamb et al. The ObjectStore Database System. *Communications of the ACM,* 34(10), October 1991.

[lay88] P. Layzell, P. Loucopoulos. *Systems Analysis and Development, Third Edition.* Chartwell-Bratt Studentlitteratur, 1988.

[leh91] S. K. Lehalli, N. Spyratos. Towards a Categorical Model Supporting Structured Objects and Inheritance. *FIDE Technical Report, FIDE/91/8,* 1991.

[leh92] S. K. Lehalli, N. Spyratos. Categorical Modelling of Database Concepts. *FIDE Technical Report FIDE/91/8,* 1992.

[lev91] M. Levene, A. Poulovassilis. An Object-Oriented Data Model Formalised Through Hypergraphs. *Data and Knowledge Engineering,* 6:205-224, 1991.

[lip90] S. B. Lippman. *C++ Primer.* Addison Wesley, March 1990.

[mac91] S. MacLane, I. Moerdijk. *Sheaves in Geometry and Logic, A First Introduction to Topos Theory.* Springer-Verlag, 1991.

[mat94] Matisse Sales Literature, Matisse, 1994.

[mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall International Series in Computer Science, 1988.

[mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall International Series in Computer Science, 1992.

[nel93] D. A. Nelson. *To Formalise and Implement a Categorical Object-Oriented DBMS : A Graphical Perspective.* University of Newcastle upon Tyne, PhD Thesis Proposal, August 1993.

[nel94] D. A. Nelson, B. N. Rossiter, M. A. Heather. *The Functorial Data*

Model: An Extension to Functional Databases. University of Newcastle upon Tyne, Technical Report Series, No. 488, 1994.

[nel95] D. A. Nelson, B. N. Rossiter. Prototyping a Categorical Database in P/FDM. *Proceedings of the Second International Workshop on Advances in Databases and Information Systems (ADBIS'95)*, Moscow, June 1995.

[ols94] M. A. Olson. *A Comparison of University Postgres and Montage.* Montage Software Inc., 1994.

[osb79] S. L. Osborn. Testing for Existence of a Covering Boyce-Code Normal Form. *Information Processing Letters,* 8(1):11-14, January 1979.

[pey93] S. L. Peyton-Jones, P. Wadler. Imperative Functional Programming. *ACM Symposium on Principles of Programming Languages (POPL)*, 71-84, Charleston, January 1993.

[poi92] A. Poigné. Basic Category Theory. In *[abr92].*

[pou90] A. Poulovassilis, P. King. Extending the Functional Data Model to Computational Completeness. *Lecture Notes in Computer Science,* 416:75-91, 1990.

[pou93] A. Poulovassilis, C. Small. A Domain-Theoretic Approach to Integrating Functional and Logic Database Languages. *Proceedings of 19th International Conference on Very Large Databases (VLDB 93),* 416-428, Dublin, August 1993.

[pou94] A. Poulovassilis, M. Levene. A Nested Graph Model for the Representation and Manipulation of Complex Objects. *ACM Transactions on Information Systems,* 12:35-68, 1994.

[pou96] A. Poulovassilis, S. Reddi, C. Small. A Formal Semantics for an Active Functional DBPL. *Journal of Intelligent Information Systems,* 7:151-172, 1996.

[rat98] Rational Software Corporation. Http://www.rational.com

[rhe90a] J. Rhein et al. *The Postgres Reference Manual.* University of California, Berkeley, 1990.

[rhe90b] J. Rhein et al. *The Postgres User Manual.* University of California, Berkeley, 1990.

[ros92] 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 University of Newcastle upon Tyne, Technical Report Series, No. 407.

[ros93] 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.

[ros95] B. N. Rossiter, D. A. Nelson, M. A. Heather. *The Categorical Product Data Model as a Formalism for Object-Relational Databases.* University of Newcastle upon Tyne, Technical Report Series, No. 505, February 1995.

[rot88] M. A. Roth, H. F. Korth, A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems,* 13(4), December 1988.

[row87] L. A. Rowe, M. R. Stonebraker. The Postgres Data Model. *VLDB,* 1988.

[rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

[ryd88] D. E. Rydeheard, R. M. Burstall. *Computational Category Theory.* Prentice-Hall International Series in Computer Science, 1988.

[sha92] S. C. Shapiro. *Common LISP: An Interactive Approach.* Computer Science Press, 1992.

[shi81] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM TODS,* 6(1):140-173, March 1981.

[sib77] E. H. Sibley, L. Kershberg. Data Architecture and Data Model Considerations. *Proceedings of the AFIPS National Computer Conference,* Dallas, Texas, 85-96, June 1977.

[sic93] *SICStus Prolog User's Manual, Edition 2.1, Patch #7.* Swedish

Institute of Computer Science, January 1993.

[sim90] H. Simmonds. *Lecture Notes for SERC School On Logic for Information Technology.* University of Leeds, 1990.

[sma91] C. Small, A. Poulovassilis. An Overview of PFL. *Proceedings of the Third International Workshop on Database Programming Languages,* Nauplion, Greece, 96-110, 1991.

[smi77] J. Smith, D. Smith. Data Abstraction, Aggregation and Generalisation. *ACM TODS,* 2(2):105-133, 1977.

[smi81] J. M. Smith, S. Fox, T. Landers. *Reference Manual for ADAPLEX.* Computer Corporation of America, 1981.

[smi94] N. S. Smith. *Evaluation of Triggers in an Object-Relational Database System.* University of Newcastle upon Tyne, MSc Dissertation D601, 1994.

[sto86a] M. Stonebraker. Document Processing in a Relational Database System. *The INGRES Papers,* Addison Wesley 357-375, 1986.

[sto86b] M. Stonebraker, L. A. Rowe. The Design of Postgres. In *Proceedings ACM SIGMOD Conference*, pp. 340-355, 1986.

[sto94] M. Stonebraker. *Object-Relational Database Systems.* Montage Software Inc., 1994.

[str91] B. Stroustrup. *The C++ Programming Language, 2nd Edition.* Addison-Wesley, 1991.

[sut92] D. R. Sutton, P. J. H. King. Integration of Modal Logic and the Functional Database Model. *Lecture Notes in Computer Science,* 618, 1992.

[tan93] A. Tansel, et al. (eds). *Temporal Databases : Theory, Design and Implementation.* Benjamin Cummings, 1993.

[tod76] S. Todd. The Peterlee Relational Test Vehicle: A System Overview. *IBM Systems Journal,* 15(4), December 1976.

[tsi78] D. Tsichritzis. ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Database Management Systems. *Information Systems,* 3(3):173-192, 1978.

[ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems 1*. Computer Science Press, 1988.

[vic91] S. Vickers. Geometric Theories and Databases. In: Applications of Categories in Computer Science, ed. M. P. Fourman, P. T. Johnstone, A. M. Pitts, *London Mathematical Society Lecture Note Series*, 288-314, 1991.

[wag85a] E. G. Wagner. *Algebraic Theories, Data Types, and Control Constructs.* IBM Research Report, RC 11343, August 1985.

[wag85b] E. G. Wagner. *Categorical Semantics, or Extending Data Types to Include Memory.* IBM Research Report, RC 11456, October 1985.

[wag89] E. G. Wagner. Categories, Data Types, and Imperative Languages. *Lecture Notes in Computer Science*,240, 1989.

[wel88] J. Welsh, J. Elder. *Introduction to Pascal, 3rd Edition.* Prentice-Hall, 1988.

[wik87] A. Wikstrom. *Functional Programming using Standard ML.* Prentice-Hall International Series in Computer Science, 1987.

[win89] H. Winston, B. K. P. Horn. *Lisp, 3rd Edition.* Addison-Wesley, 1989.

[wol95] J. E. Wolfengangen. Object-Oriented Solutions. *Proceedings of the Second International Workshop on Advances in Databases and Information Systems (ADBIS'95)*, Moscow, June 27-30, 1995.

[woo88] J. Woodcock, M. Loomes. *Software Engineering Mathematics.* Pitman, 1988.

[zlo75] M. M. Zloof. *Query By Example*. Proceedings NCC **44**, Anaheim, California, AFIPS Press 1975.