Suitability of Programming Languages for Categorical Databases

D A Nelson, B N Rossiter

Department of Computing Science,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU

January 1995

Abstract

Earlier work has defined a formalism for object-relational databases, based on category theory, using the concepts of multi-level mappings and products. A prototype implementation of this database model, known as the Product Model, is to be developed, but there are known problems in languages providing rich enough structures in which to produce a categorical model. This paper examines five well known programming languages, from a selection of programming paradigms, and the DAPLEX functional database language, in order to determine the most suitable language for implementing the prototype data model.

About the author

David Nelson is a Ph.D. student in the Department of Computing Science, sponsored by the EPSRC, with research interests in databases and category theory.

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

Suggested Keywords

Databases, functional model, object-relational model, category theory, product model, DAPLEX, P/FDM, Gofer, Lisp, C++, Pascal, Prolog, lambda calculus.

1. Introduction

This paper examines a selection of programming languages, from various paradigms, in an attempt to determine which is the most suitable for programming the Product Model [Rossiter95, Nelson94].

The product model is a framework for object-based databases, intended to provide a formal basis for the object-relational database model [Stonebraker94]. It uses standard constructs from category theory [Barr90], based on multi-level mappings and products.

The languages we are going to look at are: DAPLEX, Gofer (as one of many candidate functional programming languages), Prolog, Lisp (as a lambda calculus), C++ and Pascal. These languages form a relatively wide cross-section of the programming paradigms used, including functional and object-oriented, and the functional database model.

To decide on which would be the most suitable language, a set of criteria needs to be established. The criteria devised are as follows:

- Higher order functions : we need to be able to pass functions as first class objects [Rydeheard88], since the ability to handle mappings is one of the basic concepts of category theory;
- Loose typing : if the type system of the language is too strict, then it may prove difficult to define a general category. Ideally, a language which supports heterogeneity or dynamic binding may be best;
- Persistency : although most languages have some form of basic input/output, the structure of the categorical database would require storage of categories, pullbacks, and relationships. This requires the input/output system to be able to cope with complex data structures;
- High productivity : for a prototype application, a language which gives high productivity would be beneficial. Of course, if the language supported the above three criteria, then it should be productive.

2. Language Review

The main issue in categorical programming is in type systems. These have been investigated before, in Rydeheard [Rydeheard88], where the functional language ML [Wikstrom87] is used to produce most of the constructs of category theory. In ML, polymorphic types allow the generality of category theory to be expressed [Rydeheard88], although he suggests it is debatable whether a programming language with dependent types or a subtype mechanism is best.

We will look at the languages mentioned in the introduction, in order to ascertain the problems that may arise in prototyping a categorical database in any of those languages, so that a choice can be made as to which language will be most suitable.

2.1 DAPLEX

DAPLEX is the best known functional database model, developed by Shipman [Shipman81]. 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.

Although DAPLEX is not really a programming language, it was included in this review because it 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 what 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 [Kulkarni86], Extended Functional Data Model. A noticeable omission of DAPLEX was its inability to attach a 'general computation' function to one of the arrows, i.e. although attributes were defined as functions, they were really nothing more than relationships, as no general computation other than those supported by the DBMS could be made. EFDM added limited facilities for supporting general purpose computations, including PS-Algol [Kulkarni87] features in their self-contained EFDM system [Kulkarni86], as well as facilities for specifying integrity constraints and for handling named views of databases. Later extensions, by Dayal [Dayal89] and by Gray [Gray88], 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 was not very 'functional'. Gray also 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. The P/FDM system is embedded in SICStus Prolog [SICStus93], 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 types of return result. The result can be printed to the screen, another query can be nested within it, or the result may be an UPDATE statement, as defined in the language syntax [Gray88, Shipman81], or even 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 new entities can be defined, or entities already in the database can be updated, as the result of a query.

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, etc.

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 [Embury92], which can be accessed uniformly in a DAPLEX query (i.e. 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.

2.2 Gofer

Gofer [Jones94] is a version of Haskell [Hudak92]. Its main differences are that it is interpretative rather than compiled, and incorporates constructor classes, which allow a class to have more than one type constructor. Current category theory systems have usually been implemented in functional languages. Rydeheard et. al. [Rydeheard88, Dennis-Jones93] have looked at implementing categorical features as types in ML, and

Duponcheel [Duponcheel94] has produced an experimental prelude defining categories in Gofer.

Both of these systems have problems though. Dennis-Jones and Rydeheard acknowledge that the polymorphic type system of ML is too restrictive to express general categorical structures [Dennis-Jones93], a problem that most functional languages would have. Their system also does not allow families of categories (i.e. a category of categories), because the ML type system will not allow recursive type definitions such as this.

The prelude produced by Duponcheel [Duponcheel94] defines categories as containing a homomorphism:

```
class Identity hom where
  id :: hom x x

class Composition hom where
  (.) :: hom y z -> hom x y -> hom x z

class (Identity hom, Composition hom) => Category hom
```

This appears to be a mathematically natural way of defining a category, based on homomorphisms, but the shortfall of their system is that it appears to only support general categories which contain one arrow, or a composable collection of arrows, with more complex categories, such as product categories, defined as special cases. It is difficult to see how a database object could be defined using their implementation of a category, unless a database category was defined as a collection of categories in their system.

The main problem with functional languages is that their type systems, although polymorphic, are too strict. It is difficult to define heterogeneous lists, a concept which would be useful for using a set structure to store the arrows in a category, unless the list was defined as a tuple. This is not a general solution though, because a specific tuple would have to be defined for every list used in the system, and redefined if the size of the list changed.

2.3 Prolog

Prolog [Bratko90] is a symbolic language based on clauses as relations between objects, used mainly in the field of Artificial Intelligence. It includes the concepts of backtracking, pattern matching and tree bound data structuring [Bratko90]. Prolog does not really support numeric computation[1] .

[1] Standard numerical operations of +, -, etc. are supported, but only on integers

The language is completely typeless, which has its advantages in that heterogeneous lists or structures are easy to handle, but an important disadvantage is that the language lacks any basic types other than characters, integers, and booleans.

Prolog has the ability to pass relations whose type is unknown, through the use of the functor concept [Bratko90], i.e. allowing a relation to be passed to another relation, where its name (known also as the functor) is only known at call time. This in effect allows relations to be classed as first-class objects. Prolog also allows clauses to be more than just normal relations, allowing some computation to be attached to them.

The following Prolog clauses illustrate one method of representing basic categorical constructs, namely the arrow and the category. Composition is defined by the composable clause, but this is not a constraint as in functional languages such as Gofer, and heterogeneous lists are used to represent a category. The categorical constructs can therefore be represented in Prolog in a relatively 'natural' manner compared to their representation in Gofer.

```
object(X).

arrow(X,X).
arrow(arrow(X,Y),arrow(Z,W)) :-
      !, composable(arrow(X,Y),arrow(Z,W)).
arrow(X,Y) :- object(X), object(Y).

composable(arrow(X,Y),arrow(Y,Z)).

category([arrow(X,Y)]).
category([arrow(X,Y)|Arrows]) :-
      category(Arrows).
```

So basically, the first clause defines any object to be definable as a categorical object. Arrows are then defined as a relationship between any two objects, or as a composition of two arrows, where the target of the first arrow is equivalent to the source of the second arrow, as tested by the composable relation. The composable relationship is associative because either arrow in the test can be a composed arrow. An identity arrow could simply be defined as an arrow where the target object is the same as the source object, i.e. arrow(X,X). Finally, the definition of a category is as a list of arrows, which, because of the definition of an arrow, may be composable, and may also contain identity arrows.

As with DAPLEX, it is possible to have inverses in Prolog. Because the language is based on pattern matching and backtracking, then any defined function should be able to have its inverse determined by supplying a clause with the result and using backtracking, i.e.

```
arrow(Source,4).
```

would return sources of arrows which have the integer value four as a target.

Persistence in Prolog can be handled in one of two ways. Prolog first of all has a basic input/output system, equivalent to input/output in most other languages, but again, this is too restrictive for storing categorical structures. But Prolog also allows clauses to

be stored in the database of terms currently being used, and therefore in the file which is currently storing terms. This allows Prolog clauses to be persistent, and therefore allows more complex structures than simple text to be used in a persistent manner.

A notable point is that most of the implementations of DAPLEX currently available, especially that of P/FDM, are directly implemented on top of a Prolog system. This implies that producing a categorical database, which is strongly influenced by functional database concepts, in Prolog would involve a task similar to producing DAPLEX in Prolog, which has already been done.

2.4 Lisp and the Lambda Calculus

Lisp is a language based on the lambda calculus [Barandregt84]. As with Prolog, its main use is in the field of Artificial Intelligence, although it has an advantage in that it supports numeric computation and symbolic algebra. There are many versions of Lisp, the best known being Common Lisp [Shapiro92, Winston89], CLOS (The Common Lisp Object System) [Shapiro92, Winston89], which is an extension of Common Lisp to include some object-oriented features, and Scheme [Hanson91a, 91b].

The expressiveness of the lambda calculus should enable any language such as Lisp to support categorical constructs, although this would usually involve the rewriting of any formalism into unnatural lambda calculus concepts. In Barr and Wells [Barr90], they briefly show a formalism for mapping between cartesian closed categories and the typed lambda calculus, although our categorical model would require more than a lambda calculus representation for cartesian closed categories.

Most dialects of Lisp, including Common Lisp, support heterogeneity. Lisp's main construct is the s-expression, a list of Lisp expressions, where the list can be heterogeneous. As with Prolog, the language is loosely typed, although in this case the list of basic types it actually supports is much more extensive than in Prolog. Lisp, for example, does support floating point numbers.

Persistency in Lisp is of interest. Input/Output is handled through streams, where a file containing Lisp expressions can be processed, in much the same way as in Prolog. This allows any Lisp s-expression to be an input or output value in Lisp.

Finally, the eval [Hanson91b] function allows evaluation of expressions which have been created 'on-the-fly', and allows selective evaluation of members of an s-expression. This would allow, for example, part of a list to be a Lisp expression, which could be evaluated at run-time, e.g. a category which is defined and contains a method arrow could be evaluated whenever required.

2.5 C++

C++ [Lippman90, Stroustrup91] is the most widely known object-oriented programming language, whereas Prolog, Lisp, and in some sense DAPLEX, are based on the functional programming paradigm.

C++ can use the class concept for representing categorical constructions, and there is support for heterogeneous structures, as it is pointer based, and therefore dynamically typed. C++ is strongly typed, although polymorphic, which could hinder implementation. It should also be possible for functions to be passed as first class objects in C++, although this may not work between classes because it breaks the encapsulation rule.

A possible C++ header class for defining an arrow is shown below. It shows basically the method of having to use pointers for storing the source and target of the arrow, and that the types of these arrows must be explicitly cast when used, because of the strongly typed nature of C++.

```
class Arrow()
{
      private:
            void *dom;
            void *cod;

      public:
            Arrow(void* source, void *function);  /* target is
                                                    a function */
            Arrow(void* source, void *target);  /* target is a
                                                  relation */
}
```

Productivity and extendibility in C++ would cause problems. Productivity is likely to be low because of the manner in which C++ handles pointers to objects. It would be difficult to extend the system because of the complexity of programming a categorical structure in an object-oriented manner, where structures are tightly coupled. C++ is also a strongly typed language, where any dynamic typing would have to be handled at run-time.

All these problems suggest that C++ would not be the ideal language for prototyping a categorical database system.

2.6 Pascal

Pascal [Holmes87, Welsh88] is a strongly typed, imperative language. Heterogeneity is not supported in Pascal, and it is difficult to treat functions as first class objects.

Wagner [Wagner85a, 85b, 89] has actually looked at category theory and the Pascal language. But, what he has investigated is a formal way of representing Pascal

programming constructs, mainly using the product and co-product constructs of category theory, and equating them to the record and union structures of Pascal. In this way he has devised a formal way of representing Pascal programs in categorical terms.

In Pascal, a category representing a specific object in the database could be defined as:

```
TYPE PersonCategory = RECORD
                        arrow1 : RECORD
                                   dom : INTEGER;
                                   cod : STRING
                                 END; {arrow1}
                        arrow2 : RECORD
                                   dom : BOOLEAN;
                                   cod : REAL
                                 END {arrow2}
                      END; {Category}
```

This, in effect, defines a category which contains two arrows. Arrow 1 has a source which is an integer, and a target which is a string, i.e. a mapping from an object in the domain of integers, to a target in the codomain of strings. Arrow 2 is a mapping from a boolean to a real number.

This is a completely non-general solution, where a different record would have to be defined for each categorical type, very much equivalent to defining tuple types in a functional language such as Gofer. There is also no inheritance in Pascal, so a general category type would be difficult to implement.

3. Conclusions

In conclusion, all languages have their own specific problems, and although category theory provides a clear manner of formulating programming aspects, it is a difficult concept to program. No language was specifically designed for direct support of categorical constructs, and no language supports all features of category theory. Functional languages, for example, support a subset of category theory, which causes problems when implementations of categorical data types in those functional languages are attempted.

It would seem that a typeless language would be best, giving the choice of Prolog or Lisp. But DAPLEX also warrants consideration, in that its basic constructs can be thought of as very similar to those of category theory, i.e. objects and arrows.

For prototyping the product model, P/FDM seems the most natural choice. It already provides a rich function structure, along with persistency and many other normal database concepts, which allow the model to be developed without having to worry about normal database implementation problems. In this sense, our implementation can be viewed as an enhancement of a functional model rather than a completely new model (as nested relational is to the relational model). Also, if any categorical

constructs are difficult to implement in DAPLEX terms, then its close integration to the SICStus Prolog system would allow Prolog concepts to be used as well.

4. Acknowledgements

Many thanks must go to Professor Peter Gray and his associates at Aberdeen University, who are currently in the process of making the P/FDM system available for our use.

5. References

[Barandregt84]
Barandregt84. *The Lambda Calculus, Its Syntax and Semantics.* North Holland, 1984.

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

[Bratko90]
Bratko I. *PROLOG Programming for Artificial Intelligence, 2nd Edition.* Addison Wesley, 1990.

[Dayal89]
Dayal U. *Queries and Views in an Object-Oriented Data Model.* Second International Workshop on Database Programming Languages, 1989.

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

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

[Embury92]
Embury S M, Jiao Z, Gray P M D. *Using Prolog to Provide Access to Metadata in an Object-Oriented Database*. International Conference on the Practical Application of Prolog, April 1992.

[Gray88]
Gray P M D, Moffat D S, Paton N W. *A Prolog Interface to a Functional Data Model Database.* Lecture Notes in Computer Science, 1988.

[Hanson91a]
Hanson C. *MIT Scheme User's Manual, Draft Edition 0.9.* Massachusetts Institute of Technology, January 1991.

[Hanson91b]
Hanson C, et. al. *MIT Scheme Reference Manual, Edition 1.1.* Massachusetts Institute of Technology, November 1991.

[Holmes87]
Holmes B J. *Pascal Programming.* D P Publications, Ltd, 1987.

[Hudak92]
Hudak P, Fasel H J. *A Gentle Introduction to Haskell.* SIGPLAN Notices, Vol. 27, No. 5, 1992.

[Jones94]
Jones M P. *An Introduction to Gofer, Draft Report.* Yale University, July 1994.

[Kulkarni86]
Kulkarni K G, Atkinson M P. *EFDM : Extended Functional Data Model.* The Computer Journal, Vol. 29, No. 1, 1986.

[Kulkarni87]
Kulkarni K G, Atkinson M P. *Implementing an Extended Functional Data Model Using PS-Algol.* Software Practice and Experience, Vol. 17, No. 3, March 1987.

[Lippman90]
Lippman S B. *C++ Primer.* Addison Wesley, March 1990.

[Nelson94]
Nelson D A, Rossiter B N, Heather M A. *The Functorial Data Model : An Extension to Functional Databases.* University of Newcastle upon Tyne, Technical Report Series, No. 488, May 1994.

[Rossiter95]
Rossiter B N, Nelson D A, Heather M A. *The Categorical Product Data Model as a Formalism for Object-Relational Databases.* University of Newcastle upon Tyne, Technical Report Series, in print, 1995.

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

[Shapiro92]
Shapiro S C. *Common LISP : An Interactive Approach.* Computer Science Press. 1992.

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

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

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

[Stroustrup91]
Stroustrup B. *The C++ Programming Language, 2nd Edition.* Addison Wesley, 1991.

[Wagner85a]
Wagner E G. *Algebraic Theories, Data Types, and Control Constructs.* IBM Research Report, RC 11343, August 1985.

[Wagner85b]
Wagner E G. *Categorical Semantics, or Extending Data Types to Include Memory.* IBM Research Report, RC 11456, October 1985.

[Wagner89]
Wagner E G. *Categories, Data Types, and Imperative Languages.* Lecture Notes in Computer Science, Vol. 240, 1989.

[Welsh88]
Welsh J, Elder J. *Introduction to Pascal, 3rd Edition.* Prentice Hall, 1988.

[Wikstrom87]
Wikstrom A. *Functional Programming using Standard ML.* Prentice Hall International Series in Computer Science, 1987.

[Winston89]
Winston P H, Horn B K P. *Lisp, 3rd Edition.* Addison-Wesley, 1989.