University of Edinburgh
Department of Computer Science

James Clerk Maxwell Building
The King's Buildings, Edinburgh

University of St Andrews
Department of Computational Science

North Haugh
St Andrews, Fife

Experimenting with the Functional Data Model

Alan Dearle.

PPR-5-83
September, 1983

8-1-84.

EXPERIMENTING WITH THE FUNCTIONAL DATA MODEL

M. P. Atkinson and K. G. Kulkarni

Department of Computer Science
University Of Edinburgh,
Mayfield Road,
Edinburgh EH9 3JZ

PERSISTENT PROGRAMMING
RESEARCH REPORT 5

SEPTEMBER 1983

1. Introduction

The idea of viewing an information system as a collection of functions appears in many proposals of data models [5, 15, 19, 29], each of them termed as the **functional data model**. As early as 1974, Folinus et. al [10] had suggested adopting the functional approach to information systems because *requests for answers, whether made to a processing program or a stored data base, are essentially requests for a value of a function, given various argument values* and went on to observe that this approach makes it possible to

- provide various kinds of *virtual information* i.e., the information that is not physically stored in the database, but is nonetheless accessible through a combination of algorithms and stored data.

- accomodate complex, procedural interrelationships among data items in addition to static and grouping relationships that are accomodated by the conventional data models.

- incorporate some kind of *inferential ability* in the system, i.e., the ability to infer a certain fact from some existing facts.

Kent [17] also recognises the fact that the functional approach makes it possible to *equate data in the system with what can be extracted, rather than what is physically stored*. It is the functional data model proposed by Shipman [29] which exploits these benefits of the functional approach to a significant extent and this is the model we took as our starting point.

Shipman's proposals also contain a high level, integrated data definition and data manipulation language, DAPLEX. DAPLEX supports query formulation based on set and function operators as well as update operations in which the abstract entities and not a group of values act as units of update. DAPLEX is the subject of the ADAPLEX experiment [34] which aims at embedding a derivative of DAPLEX in ADA [16]. DAPLEX is also the common high level query language in a heterogeneous distributed database system MULTIBASE proposed by Smith et. al. [32].

Rather than depend on embbedding DAPLEX in a compile-and-run procedural language for a working system, we chose to build a self-contained system with an interactive user interface based on the DAPLEX language. This interface allows for interactive creation, retrieval, and modification of both meta-data and data in the databases. Based on the preliminary experience gained, a number of modifications to both the data model and the database language have been made. We present proposals for continuing this evolution to accomodate versions, experiments, control of persistence, access control and federations using one consistent linguistic construct.

We first present a brief description of the model in Section 2 and discuss our reasons for choosing to work with this model in Section 3. Section 4 discusses the logical design of our variant of the functional data model with emphasis on the management of operations on meta-data and the control of name spaces as an authorisation and view mechanism. Section 5 speculates on the effects of allowing views to store data within themselves. Section 6 discusses language issues while Section 7 describes our implementation strategy based on a persistent algorithmic language *PS-algol* [2].

2. Functional Data Model

A good exposition of the Functional Data Model (FDM) is given by Shipman [29]. Kulkarni [20] provides a detailed discussion of this and other proposals of FDM. Briefly the data consists of a set of entities, and of functions relating the entities. The entities have types which are arranged in a type hierarchy, so that they are all subtypes of the type entity. Certain types are provided initially, distinguished by having a lexicographic representation. For example string. A function may be single-valued or multi-valued, i.e., the result of a function is a set. The model also allows multi-argument functions and these provide a convenient means to establish relationships involving entities belonging to more than two entity types without introducing artificial entities.

An example to represent a class of students and associated entities is given in fig. 1. Note **declare** or **define** introduces a new function, which if it has no arguments is a new entity type with the result type as its supertype. Those with arguments model properties of their arguments, by defining a result entity which is the value of that property. Subtypes inherit properties of their supertypes. Single arrow -> implies the function is single-valued, and double arrow ->> implies it is multivalued. A function introduced by **declare** is called a **base function** and is represented by physically storing a table of arguments and results. For example, a function relating persons and their names. A function introduced by **define** is called a **derived function** and is represented by an algorithm to compute its result.

```
declare person()                 ->> entity
declare student()                ->> person
declare staff()                  ->> person
declare course()                 ->> entity
declare tutorial()               ->> entity

declare name(person)             ->  string
declare sex(person)              ->  string

declare course(student)          ->> course
declare tutorial(student)        ->  tutorial
declare grade(student, course)   ->  string

declare staff(course)            ->  staff

declare staff(tutorial)          ->  staff

define  lecturer(student)        ->> staff(course(student))
define  tutor(student)           ->  staff(tutorial(student))
```

Fig. 1. Data Description of Student Database

3. Why Choose the Functional Data Model?

We had essentially two reasons for implementing the functional data model. The first reason was to evaluate the model itself and the second was to establish a flexible database design environment for which we considered FDM the best candiadate.

3.1 Properties of the Model

As to the data model, we were attracted by the following features of the FDM:

(1) It is simple to understand and use.

It is well known that the existing data models fail to capture a substantial portion of the information associated with an application environment [13, 17, 18]. Because of this reason, data modelling has been one of the major themes of database research for more than ten years. As a result, a number of new *semantic* data models [1, 5, 7, 8, 12, 25, 27, 29, 31, 33]have been proposed. While some of these models suffer from excessive complexity, some others are incomplete in the sense that they do not describe the set of operations on the data structures they model. This was probably the reason why so few of them were subject to the test of implementation. Consequently reports of work on the tools for designing or maintaining the databases based on these models is almost non-existant. It is difficult to determine the efficacy of a data model without using it for a range of applications and to do this, the model must be implemented. Simplicity is attractive to the implementor in limiting the size of his task, and to the user in moderating the effort of learning and comprehension needed. FDM has this simplicity. In addition, it supports a well-defined set of operations.

(2) It is object based and not name based.

This aspect of FDM, supported by many other data models as well [12, 33], implies that the user is not required to be aware of the mapping from real world objects to values which act as tokens for them in the stored data. This also means that many referential integrity constraints [8, 9] are implied by the data model itself and it is possible to support varied external naming conventions.

(3) It provides a semantically rich modelling environment.

The concept of function augmented with multi-valued and multi-argument functions leads to uniform modelling of all types of inter-object relationships.

Because of this, creation of artificial entities to model non-binary or many-to-many relationships is avoided in this model. In addition, the accomodation of subtype-supertype hierarchy among entity types captures an important real-world semantics.

(4) It removes the sharp distinction between data and programs.

All functions irrespective of whether they are defined by stored data or by programs have equal rights. A large part of the application specific knowledge exists not as stored data but as a set of programs or *rules*. The concept of derived functions in this model captures such procedural knowledge as part of the schema itself. Because of this, FDM provides the ability to define relationships in a procedural fashion. For example, the relationship between a person and his age, as distinct from his date of birth. It also gives considerable flexibility in the variety of views that can be supported.

(5) It has a 'simple' data manipulation language.

The function application provides a natural mode of expressing user queries. Because of this, query formulations are closer to the natural language form.

(6) It has the potential to provide a data language in which data manipulation facilities are neatly integrated with the general-purpose computation facilities.

Existing data languages, both procedural and non-procedural, do not provide a complete programming environment. To overcome this deficiency, these languages are invariably coupled to existing procedural programming languages or alternatively, existing programming languages are extended with database notions. Both these approaches are found to be less than ideal [26]. A major reason for this is the lack of a common concept between the data models and the programming languages. FDM is well suited to overcome this deficiency as the concept of function to model stored data is also the basis of many general-purpose programming languages like LISP [22]. Buneman et al. show how this can be achieved in their functional query language, FQL [5].

(7) It has the potential to support all three major (hierarchical, CODASYL, and relational) data models [30, 29, 32, 11].

FDM is proposed as the global data model in the heterogeneous distributed database system, MULTIBASE [32].

3. 2 Fast Prototyping and Schema Modifications

The other reason for implementing this data model is to establish a flexible database design environment. One of the major problems with the traditional design methodologies is that they aim to *get it right the first time*. Since this is difficult to ensure in practice, there is an important need to accomodate user *experience* in the design process. Design approaches which employ *prototypes* are more appropriate in the database context.

There is also a necessity for economical mechanisms to effect changes. Such changes may be due to the improved insights gained during the experimentation on the prototype or because the requirements of the application the database is modelling has changed. Thus, continuous modifications to the schemas may be necessary to ensure that the system reflects the requirements as accurately as possible at all times.

The simplicity of FDM combined with the power of DAPLEX makes it is easy to construct prototypes in a short period of time. **declare** and **define** constructs of DAPLEX provide a natural way to describe increments to the schema. Using these commands, users can construct their schemas incrementally, i. e., they can add a component of the schema at any time. If modify and drop commands are provided, users can modify, or drop a component of the schema at any time. The meta-data can also be described as a set of functions which are updated automatically as the schema changes. This makes it possible for the users to examine the structure of the database at any time.

4. Extended Functional Data Model – EFDM

EFDM is our implementation of FDM. It provides an interactive user interface, using which the users can create, query, and modify databases. Both the model and the language closely follow the proposals of Shipman, though the language has been considerably extended to handle general purpose computation and views. The syntax for some of the constructs has also been modified. We briefly sketch here the salient points of EFDM; more detailed information can be found in the user manual [21]. A summary of the syntax appears in the Appendix A.

4. 1 Data Definition

We describe the data structure of an application in the form of function definitions using **declare** and **define** commands. Like DAPLEX (and unlike ADAPLEX), we allow the individual function declaration statements to occur at any time. To avoid inconsistencies, the users are helped to ensure that the same fact is not described by more than one base function. For example, assume that there exists a base function

student(course) ->> student.

Suppose now the user intends to add another function

course(student) ->> course.

If, in reality, this function is the inverse of the old function, adding it as a base function will mean that the same fact is represented by two independently updatable functions, and this will surely lead to inconsistencies in the database. On the other hand, the new function may be corresponding to a new fact, say, relating to the majors taken by a student. Since there is no way for the system to infer what is intended, whenever addition of a base function with a single argument is requested, EFDM displays all the existing base functions between the argument and result types and some "paths" that can be derived using inversion and composition of other base functions. The user is consulted to check if the function addition should proceed.

We also provide

drop <function>

facility to remove items from the schema. To avoid inconsistencies, **drop** may cause cascade deletion of functions that depend on the function being deleted. If there are implicit deletions, the user is consulted to check if the cascade, once identified, should proceed.

4.2 Manipulating Meta Data

The meta data of the schema corresponding to an application is held in a set of EFDM functions shown in fig. 2. These functions are automatically populated and modified when declare, define, or drop statements are processed. Only the document function may be explicitly updated by the user. The contents of these functions can be retrieved with the usual retrieval statements. So a user may use such queries to discover the form of a data base. To facilitate this the functions given in fig. 3 are defined.

```
function()             ->> entity -- contains a list of all
                                      functions existing in
                                      the database
name(function)         ->  string  -- name of the function
nargs(function)        ->  integer -- number of arguments the
                                      function has
arguments(function)    ->> function -- list of the argument
                                       functions
result(function)       ->  function -- the result function
type(function)         ->  string -- whether the function is
                                     single-valued or multivalued
status(function)       ->  string -- whether the function is
                                     base or derived
text(function)         ->  string -- the text of the function
                                     definition
document(function)     ->  string -- documentation associated
                                     with that function
```

Fig. 2. The functions to hold meta data of a schema

```
entitytype()               ->> f in function such that
                                nargs(f)=0
supertype(entitytype)      ->   result(entitytype)
supertypes(entitytype)     ->>  transitive of
                                supertype(entitytype)
subtype(entitytype)        ->> e in entitytype such that
                                result(e)= entitytype
subtypes(entitytype)       ->>  transitive of
                                subtype(entitytype)
fnsover(entitytype)        ->> f in function such that
                                nargs(f) ~= 0 and
                                some e in arguments(f) has
                                (e = entitytype or
                                some e1 in
                                supertypes(entitytype) has
                                e = e1 )
fnsyielding(entitytype)    ->> f in function such that
                                                     nargs(f) ~= 0 and
                                                     result(f) = entitytype
```

Fig. 3. The derived functions for querying meta data

4.3 Manipulating Extensional Data

The database query and update operations are again based on DAPLEX constructs. Additionally, we provide for explicit entity deletion operation. This is provided by

delete <entity>

which deletes the specified entity and causes a cascade deletion of all functions which reference this entity and all its subtype entities, again consulting the user before the cascade proceeds, unless there are no implicit deletions.

In addition to string, integer, and boolean, two additional primitive types of time and date are also supported. The usual arithmetic operations like addition, subtraction, multiplication, and division are provided for numeric-valued expressions. Concatenation, substring operations and various string matching tests are provided for string-valued expressions. Similarly, the boolean operations of and, or, and not are provided for boolean expressions. forward and difference operations as well as earlier and later tests are provided for date and time types.

4.4 View Mechanism

EFDM also provides a *view* mechanism, which while providing different perspective of a global information space, also acts as authorisation mechanism. Views are defined using deduce statements. For example, for student database of fig. 1, we can define a view called *malestudents* as

```
view malestudents is

deduce male()          ->> entity using student
                                such that sex(student)="m"
deduce name(male)      ->  string using name(student)
end
```

All functions introduced by deduce are treated as derived functions. Notice that deduce is used to define view functions instead of define. This is because view function definitions involve change of name space: names before the using keyword define the namespace of the view being defined whereas names after the using keyword refer to those in the name space in which the view is being defined. In fact, every view definition automatically creates a different name space, which is completely independent of the global name space as well as of the name spaces of other views. The users of a particular view are not allowed to see the global name space or the name spaces of other views.

Once within a view, users can pose the usual FDM requests to carry out the database operations using the names available in their name space. Each query statement issued from an user view is translated into a corresponding query on the global name space by recursively applying view definition mappings and the global query is then executed and results displayed.

Since all the functions in a view are treated as derived functions, updates through view functions are allowed only if procedures for updating the corresponding global data is provided for each view function.

Currently, the operations provided to change the schema within a view are either defining derived functions from the set of functions they are allowed to have access to or dropping the derived functions they have created. This is in keeping with the assumption that the global view has all the information required by different users. This means that the individual users are not allowed to introduce any base functions or stored data of their own. Views can be dropped with

```
   drop <viewname>
```

command.

4.5 Operations on Views

As well as creating, using, and dropping a view, users can also establish a view as the context for subsequent operations. To do this three statements are required:

```
quote <string>
open  <viewname>
close <viewname>
```

At the start of a session the database holds no quoted passwords, each time quote is encountered the value of the string expression which follows is added to the set of quoted passwords.

open then makes the specified view as the current context for operations if there is no constraint on its use, or if one of the passwords which enable its use has been quoted. The only names then available are names provided by that view. Subsequent operations, define definitions, queries, updates and new view definitions are interpreted in, and modify the name space of that current view. The most global view is the schema.

The opening and closing of views is used to define transactions. A transaction starts when a view is opened, and is committed when the close statement is

encountered. If no close is encountered the operations will not be recorded.

4.6 Representing Views

The information about currently defined views is held in the EFDM structures given in figure 4.

view()	->>	entity
name(view)	->	string
context(view)	->	*view*
text(view)	->>	string
password(view)	->	string
document(view)	->	string

Fig. 4. The functions to hold the meta data for views

5. Deferred View Updates and Views with Memory

In section 4 the view mechanism for identifying smaller name spaces and constraining access to them was described. Provided a user gives a hierarchy of views all with the same password, they can also be used to subdivide and make more manageable the name space for a given user, who may then move freely about this tree of name spaces. This is similar to modules or block structures in programming languages, and is clearly useful.

What we have considered are the mechanisms for deferring view updates in order to conduct experiments of *what-if* nature and allowing a view to have memory so as to provide version management capabilities.

5.1 Deferred View Updates

Section 4 assumed that updates may be made through a view if update procedures are defined and that all the updates done from a view will be applied immediately, i.e., propogated up to the global view. However, in order to conduct experiments of *what-if* nature on databases appropriate in management and design, we need to have a mechanism which allows updates from a view to appear to have happened within that view but from outside that view they appear not to have happened. This can be achieved easily if we allow the update procedures for view functions to be modified by preceding with the word **defer**. We can interpret this as meaning that each time an update is requested, the system records enough data to permit it to be done later. Thus, a user can experiment with the data by making complex and extended updates without interfering with other people's work until he chooses to do so.

Two new operations are then introduced:

```
  apply <viewname>
and reset <viewname>
```

When a view is created the set of deferred updates is empty. As successive user sessions operate on this view a sequence of updates will be accumulated. The interaction of the operations in the sequence of updates is examined and an equivalent nett effect sequence is built. **apply** causes this accumulated sequence of updates to be applied in the view's context, and the stored sequence of updates to be set empty. **reset** explicitly abandons a sequence of updates.

We anticipate using a differential file mechanism [28] to implement this. In fact, a slightly more subtle implementation is necessary in this context. The reason being that a deferred update may no longer be applicable due to changes in the database since it was recorded. At present we see that as a user problem. People will normally avoid such situations by agreeing the territory each works on, as they do now when teams are involved in design and management. But such division of tasks is never perfect, and must in any case be checked. When it is checked, the apply is aborted with the sequence left unchanged, and the user informed of the problem. (Note it is difficult to describe the problem so as to be meaningful to the user.) When the user has understood the *real world* problem of two people's work interfering, he will decide how to resolve it, and updates his view. Subsequent application of the view should not meet the same problem.

5.2 Views with Memory

As they stand in section 4, views are memoryless. **defines** and **deduces** may extend their name space, **drop** may reduce it, but no other updates on the structure of a view are possible. However, the different views might also want different versions of the meta-data, i.e., base relations, and there might be no intention of merging the data. This can be easily achieved by allowing **declare** to be used within a view. This would then associate base functions (i.e., explicitly stored) with the view. These would only be useable from this view and its descendents, but could be manipulated in the usual way.

This allows more general variants. It also allows people to *possess* private data, optionally *related* to the original data, without any commitment to pass on, or to allow access to others, protected by the view's password. This seems to give an equivalent to the proposed federated architectures [14, 23], except that the recursive definition means there may be many layers of federation.

If every use of a view starts with an explicit **open** command and ends with an explicit **close** command, we then have a mechanism for providing transient data. **declares** occuring in a view in the current session, which is never closed will store data during the session, but will vanish at the end of it. Thus we have a mechanism for differentiating persistent, and private data and possibly federations.

Its one drawback seems to be the laboriousness of defining views so that all the external names wanted are imported. The alternative is to introduce another concept for access control which seems less attractive. We believe the problem is best overcome by having a good cut and paste screen editor at the user interface.

Note that with the one concept of view, and the avoidance of exceptions as to where decalre, define etc. may be written, we have in one uniform and simple language achieved a number of effects, which have hitherto required different notations and treatment and have hence either been omitted or led to a more complex language. These effects are: i) views, ii) meta-data edits, iii) 'what-if' experimental grouping of operations on the data, iv) protected data spaces, v) federations, vi) identification of persistence. Although it looks feasible, the implementation of this approach has yet to be researched.

6. The Search for a Uniform Language

We would like a language which in one continuous spectrum without major discontinuity would provide simple general queries, packaged queries, and the power of a general purpose programming language.

Starting from DAPLEX one can take two paths. One is to integrate DAPLEX in a procedural language. This is the approach taken by the ADAPLEX team. While this makes the full power of ADA available for database application programming, there are three criticisms of this approach from our viewpoint:

- Since normal programming language types differ too much from DAPLEX entity types, it is impossible to adhere to the principle of data type completeness [35]. This results in a complex language.

- DAPLEX programs are predominantly applicative or functional in style. Embedding it in a procedural language leads to two different styles of programming in the same language.

- It produces a language which does not even approach the end of the spectrum concerned with doing simple things, simply and interactively.

The other, which we have only begun to explore, is to extend the language retaining the predominantly functional style. Buneman has already demonstrated how powerful this can be in formulating queries [5] and our language has comparable capabilities. The main challenge arises from adequately modelling and packaging update and I/O operations without disharmony. Essentially the idea of explicitly controlling the implementation or propagation of updates is antithetical to pure functional programming. Yet organising the storage of data and hence updates is a dominant computing activity central to databases. We are therefore expanding the set of provided functions to include I/O operations and building on the functional parameter handling mechanisms to establish whether a convenient and consistent language can be developed to satisfy the range of programming needs. Early user experience suggests this is likely to be the case. We are considering whether to retain the present syntactic form of the language or to use a form closer to familiar programming styles.

7. Implementation

Our work adopts a novel approach to building database management systems. The data model is implemented using PS-algol [2, 3, 4] which avoids us having to organise any explicit data transfers, allowing the implementor to concentrate on data structure and algorithm design. Before we discuss the specific implementation details, we will briefly describe this language and the impact it had on our work.

7.1 Brief Description of the PS-algol Language

PS-algol is an algol-like language derived from the strongly typed programming language S-algol [24] which supports a properly managed heap. The principal feature of PS-algol language is that it supports a persistent heap on which a data structure built in one run of a program may be preserved to be used in other runs of the same or other programs. Data reachability serves as the principal means of identifying data persistence. Preservation of data is a consequence of arranging that there is a way of using that data. This feature makes it possible to provide persistence as an orthogonal, variable property of data and further simplifies the provision of access to data which persists longer than the execution times of individual programs.

The use of PS-algol language for our implementation has led to many important benefits:

- Source code volumes are much reduced – for example, some portions of the implementations were done earlier using PASCAL [36]. The source code in PS-algol was about one third the length of the PASCAL source.
- Coding times are much reduced due to reduced code volumes and this has a better than proportional effect on the debugging times.
- Execution times are reduced as a consequence of avoiding layering costs. In a traditional system, every access to database items passes through many layers of subroutine calls and mappings. In PS-algol the first use of a data item automatically brings it to the heap in the right form for the current program. Thereafter, access is direct. Disk transfers may also be reduced as the collection of data on the heap has high relevance to the present computation, unlike statistically based disk-buffer caches. As user behaviour usually has only a slowly shifting focus of interest this can be very effective.
- Type rules can be strictly enforced.

Use of the PS-algol language has made it possible to concentrate exclusively on

the design of the logical aspects of the user interface and has greatly helped in conducting experiments to determine suitable high-level data structures and algorithms.

7.2 Data Structures

The data structures to implement the data model essentially fall into one of the following four categories:

1. Representation of individual entities.
2. Representation of sets of entities.
3. Representation of entity type hierarchy.
4. Representation of functions mapping entities to entities.

It is the choice of data structures for representing functions which is most important. The efficiency of the implementation critically depends on how the function values are stored and how the function values for a given entity are evaluated. In addition, it is imperative that entities are internally identifiable irrespective of how they are identified externally. Therefore, the implementation must have a mechanism which assigns a unique entity identifier to each entity upon creation.

In our implementation, the values of all one-argument non-inherited functions applicable to an entity are stored at one place in a record-like structure shown in figure 5. To access the values of functions defined on the supertype, a pointer to its immediate super-entity is included in each entity structure. As the language itself ensures that a pointer to each structure is unique, we have chosen the pointer to each entity structure as its entity identifier. This avoids us having to invent and maintain unique entity identifiers. All multi-argument functions are then implemented separately as linked list of rows of argument and result entity identifiers. It may be noted that the designers of ADAPLEX DBMS have also advocated a similar implementation strategy [6].

link to supertype object	value for function 1	value for function 2		value for function n

Fig. 5. Object Structure

Sets of entities are currently implemented as linked lists of such structures. Creating a new entity results in creation of such a structure and adding it to the corresponding linked list. Deletion of an entity results in removing all references to the corresponding structure and finally the entity structure itself from the corresponding linked list.

The above data structures refer to the base functions only. The data for a derived functions is calculated every time it is accessed. To facilitate this, the pointer to the syntax tree corresponding to its definition is stored along with each derived function. The user queries involving derived functions are processed by replacing the references to those functions by the corresponding definition tree. The resulting syntax tree is then executed against the base functions of the database.

To handle the incremental schema changes, the implementation adopts different techniques depending on the number of arguments the function has and the nature of the function. The addition or deletion of a one-argument base function results in creating the new instances of the modified entity structure and copying corresponding values from the old instances to the new instances. On the other hand, the addition or deletion of a multi-argument base function results in the addition or deletion of a table without affecting the existing entity structures. The addition or deletion of derived functions has no effect on the stored entity structures.

Compared to a relational-like implementation where each entity is assigned a unique integer to stand as its entity identifier and the values for each function are stored as binary tables, this scheme has many advantages. Firstly, in the schema we have chosen, the entity identifier is not stored in each function representation and this results in a considerable saving in storage. Secondly, the function values for a given object are provided simply by field dereference instead of table lookup. Thirdly, function composition is achieved by following the pointers instead of join-like operation in the relation-like implementation. Both these factors result in fast execution times. We have found that the above structure provides a reasonable response times for moderate volumes of data, typically found in personal data bases.

8. Conclusions

In this paper, we have tried to demonstrate that the functional data model has a great potential for solving many of the problems associated with the conventional data models without introducing undue complexity. We have done this by means of a working implementation, the novelty of which is that it is implemented using a persistent algorithmic language. The result is a powerful working system providing interactive functional programming suitable as a prototyping tool when doing database design and as a personal DBMS for moderate volumes of data. A simple device to control the visible name space promises simple provision of experiments, version management, federations and nested transactions. The language appears to be cleanly extensible to full programming power.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the efforts of Ken Chisholm, Paul Cockshott, and George Ross in providing the persistent programming environment. We appreciate many useful discussions with them and with Pedro Hepp, Segun Owoso, and Ron Morrison. We thank Roy Dyckhoff for suggesting the syntax summary given in the Appendix A.

The work at Edinburgh was supported by SERC grant 86541. It is now supported by SERC grants 21977 and 21960, and by a grant from ICL. K. G. Kulkarni was supported for the duration of this work by a Scholarship from the Association of Commonwealth Universities.

REFERENCES

1. Abrial, J. R. Data Semantics. In *Database Management*, Klimble, J. W and Koffman K. L., Ed., North-Holland, 1974.

2. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "PS-algol: an Algol with a Persistent Heap." *ACM SIGPLAN Notices 17*, 7 (July 1981).

3. Atkinson, M. P., Chisholm, K. J., Cockshott, W. P. and Marshall, R. M. "Algorithms for a Persistent Heap." *Software Practice and Experience 13*, 7 (March 1983).

4. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming." *Computer Journal 26* (1983).

5. Buneman, P. and Frankel, R. E. FQL - A Functional Query Language. Proceedings of International Conference on Management of Data, ACM-SIGMOD, 1979.

6. Chan, A., et. al. Storage and Access Structures to Support a Semantic Data Model. In *Proceedings of Eighth International Conference on Very Large Data Bases*. ACM, 1982.

7. Chen, P. P. S. "The Entity-Relationship Model: Towards a Unified View of Data." *ACM Transactions on Database Systems 11*, 1 (March 1976).

8. Codd, E. F. "Extending the Relational Model of Data to Capture More Meaning." *ACM Transactions on Database Systems 4*, 4 (December 1979).

9. Date, C. J. Referential Integrity. Proceedings of 7th International Conference on Very Large Data Bases, VLDB, 1981.

10. Folinus, J. J., Madnick, S. E., and Shutzmann, H. B. "Virtual information in Database Systems." *FDT, SIGFIDET 6* (1974).

11. Gray, P. M. D. The Functional Data Model Related to the CODASYL Model. In *Databases: Role and Structure*, Stocker, P., Ed., Cambridge University Press, 1983.

12. Hammer, M. and McLeod, D. The Semantic Data Model: a Modelling Mechanism for Database Applications. Proceedings of International Conference on the Management of Data, ACM-SIGMOD, 1978.

13. Hammer, M. and Mcleod, D. On Database Management System Architecture. In *Infotech State of the Art Report on Database*, Atkinson, M., Ed., Infotech, 1980.

14. Hiembigner, D. and Mcleod, D. Federated Information Bases (A Preliminary Report). Tech. Rept. TR-105, University of Southern California, October, 1981.

15. Housel, B. C., Waddle, V., and Yao, S. B. The Functional Dependency Model for Logical Database Design. Proceedings of 5th International Conference on Very Large Databases, VLDB, 1979.

16. Ichbiah et al. "Rationale of the Design of the Programming Language Ada." *ACM Sigplan Notices 14*, 6 (1979).

17. Kent, W.. *Data and Reality*. North-Holland, 1978.

18. Kent, W. "Limitations of Record-based Information Models." *ACM Transactions on Database Systems 4*, 1 (1979).

19. Kerschberg, L., Klug, A., and Tsichritzis, D. A Taxonomy of Data Models. Tech. Rept. CSRG-70, University of Toronto, 1976.

20. Kulkarni, K. G. *Evaluation of Functional Data Models for Database Design and Use*. Ph. D. Th., University of Edinburgh, 1983.

21. Kulkarni, K. G. Extended Functional Data Model - User Manual. University of Edinburgh, September, 1983.

22. McCarthy, J., et al. *LISP 1.5 Programmer's Manual*. 1962. MIT Press.

23. Mcleod, D. and Heimbigner, D. A Federated Architecture for Database Systems. Proceedings of the National Computer Conference, AFIPS, 1980.

24. Morrison, R. S-algol Language Reference Manual. Tech. Rept. CS/79/1, University of St. Andrews, 1979.

25. Mylopoulos, J., Bernstein, P. A. and Wong, H. K. T. "A Language Facility for Designing Database Intensive Applications." *ACM Transactions on Database Systems 5*, 2 (June 1980).

26. Pirotte, A. and Lacroix, M. User Interfaces for Database Application Programming. In *Infotech State of the Art Conference on Database*, Infotech Limited, 1980.

27. Schmid, H. A., and Swenson, J. R. On the Semantics of the Relational Model. Proceedings of International Conference on the Management of Data, ACM-SIGMOD, 1975.

28. Severance, D. and Lohman, G. "Differential files: Their Application to the Maintenance of Large Databases." *ACM Transactions on Database Systems 1* (September 1976), 256-267.

29. Shipman, D. W. "The Functional Data Model and the Data Language DAPLEX." *ACM Transactions on Database Systems 6*, 1 (March 1981), 140-173.

30. Sibley, E. H., Kerschberg, L. Data Architecture and Data Model Considerations. In *Proceedings of AFIPS National Computer Conference*, AFIPS, 1977, pp. 85-96.

31. Smith, J. M. and Smith, D. C. P. "Database Abstractions - Aggregation and Generalisation." *ACM Transactions on Database Systems 2*, 2 (June 1977).

32. Smith, J. M., Bernstein, P. A., et. al. Basic Architecture of Multibase. Computer Corporationof America, November, 1980.

33. Smith, J. M. and Smith, D. C. P. Conceptual Database Design. In *Infotech State of the Art Report on Database*, Atkinson, M., Ed., Infotech, 1980.

34. Smith, J. M., Fox, S. and Landers, T. Reference Manual for ADAPLEX. Computer Corporation of America, January, 1981.

35. Strachey, C.. *Fundamental Concepts in Programming Languages*. Oxford University, 1967.

36. Wirth, N. "The Programming Language PASCAL." *ACTA Informatica 1* (1971).

Appendix A: Syntax Specification of EFDM Implementation

```
command = imperative|
   declare funspec ("->" | "->>") typeid|
   define funspec ("->" | "->>") fundef|
   constraint identifier on funlist ->
   (total | fixed | unique | disjoint | singleton) |
   program programid is imperative|
   output programid fileid|
   view viewid is
   (deduce funcspec ("->"|"->>") typeid using
    fundef) end.
   drop (funspec|programid|viewid)|
   load|
   programid.
imperative = for each set imperative|
   for singleton imperative|
   update|print stuple.
set = vblid in set1
   [such that predicate] [as typeid]
set1 = mvfuncall|typeid|"("stuple")"|
   "(" set {(union|intersection|difference) set} ")".
singleton = exp1 {or exp1}
exp1 = exp2 {and exp2}
exp2 = [not] exp3
exp3 = exp4 [compop exp4]
exp4 = [prefix] exp5 {addop exp5}
exp5 = exp6 {mulop exp6}
exp6 = exp7 [as typeid]
exp7 = constant|vblid|svfuncall|aggcall
   the set|a new typeid|
   quant set (has|have) predicate|
   "(" singleton ")".
svfuncall = funcid "(" stuple ")".
mvfuncall = funcid "(" mtuple ")".
stuple = singleton {",")singleton}.
mtuple = expr {","expr}.
expr = set|singleton.
aggcall = (count|max|min) "(" set ")"|
   (total|average) "(" over mtuple singleton ")".
update = let funcall "=" expr|
    include (funcall|typeid) "=" set|
    exclude (funcall|typeid) "=" set|
    delete singleton.
funcall = funcid "(" stuple ")".
fundef =
    (expr |
      inverse of funcspec |
      transitive of expr |
      compound of tuple.
     ).
funcspec = funcid "(" [arglist] ")".
arglist = typeid {","typeid}.
funlist = (typeid | funcid "(" arglist ")") {","
                 (typeid | funcid "(" arglist ")") }
compop = ">"|"<"|"="|">="|"<="|"^".
quant = some|all|no|
   (at (least|most)|exactly) integer.
integer = singleton.
predicate = singleton.
constant = int|str|bool.
int = digit {digit}.
str = """" character {character}"""".
bool = true|false.
vblid = identifier.
typeid = identifier.
funcid = identifier.
programid = identifier.
viewid = identifier.
identifier = letter {(letter|digit|"_")}.
```

```
prefix = "+"|"-".
addop = "+"|"-"|"++".
mulop = "*"|"/"|rem.
```

Note: Bold-face words, and non-alphanumeric symbols enclosed in quote marks represent terminals. Lower case italic words represent syntactic categories. Curly brackets denote repetetion. Square brackets denote optionality. Grouping is expressed by parantheses. i.e., (a|b)c stands for ac|bc.