University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ

University of St. Andrews
Department of Computational Science
North Haugh
St Andrews KY16 9SS

Constructing Database Systems in a Persistent Environment

Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D.

Persistent Programming
Research Report 34
June 1987

Al Dearle.

Constructing Database Systems In A Persistent Environment.

R.L. Cooper[1], M.P. Atkinson[1], A. Dearle[2] and D.Abderrahmane[1]

[1] Department of Computing Science,
University Of Glasgow,
Lilybank Gdns,
Glasgow,
G12 8QQ

[2] Department of Computational Science,
University Of St. Andrews,
North Haugh,
St. Andrews,
KY16 9SS

Abstract

The goal of the Persistent Programming Research Group is the provision of an environment which incorporates the principle of orthogonal persistence in order to facilitate the production of large and complex software. A database management system constitutes such software and in this paper we show how a persistent store assists in the construction of such a system. We show that a small number of features in a simple persistent programming language enable efficient implementations of various data models to be built quickly. The paper surveys three attempts to provide database programs using PS-algol. In the first, the implementation of a single interface system is greatly aided by persistence. The second shows how it is possible to provide software which includes a multiplicity of interfaces and a multiplicity of underlying data models. Finally we present a novel approach which makes use of run-time compilation to create efficient storage structures tailored to the application. These experiments represent the early development of a methodology for choosing an appropriate mixture of static and dynamic binding when using persistent programming languages.

Introduction.

When producing database systems in conventional programming environments, the programmer faces many kinds of problem. Some of these, such as organising data on backing store and linking to library modules, should not be the main concern. Instead, effort should be concentrated on ensuring that the most efficient storage structure is used and providing the interface best suited to the task in hand. It is also difficult in conventional environments to provide a flexible system. It is well known that different applications require different storage methods, while different interfaces suit different users' needs. However, providing more than one storage method or user interface will usually create a considerable increase in the complexity of the system.

The provision of a persistent environment [ATKI86a, ATKI86b] allows the programmer to concentrate on important issues and to ignore problems which should be handled automatically. *Persistence* is defined as the length of time for which an object exists. This may vary from short-lived local variables, which are created and deleted within a block, to data which are stored and intended to outlast the computer system on which they are created. We believe that the way in which the programmer refers to a data object within a program should not be related to its persistence. Essentially, this means that the programmer will not have to refer to any mechanisms extraneous to the programming language (such as file managers) to handle the storage of data. It should be possible to put a specified object into the backing store, in a way that ensures that every object reachable from it is also stored automatically. For instance, if the data is relational, to store all of the data in a relation the program only needs to enter a pointer to the relations's header into the backing store and all of the associated data (tuples, column names, etc.) will be stored automatically.

This paper describes three database systems which have been implemented using the persistent language, PS-algol: a version of Shipman's Functional Data Model; a relational system, which supports a number of user interfaces; and a relational system which utilises a run-time compile facility to create structures of greater efficiency. We will describe the benefits accrued from using PS-algol, although this is not an attempt to sell the language PS-algol, but rather to apprise the database and programming language researchers and practioners of the value of certain constructs which could be present in other languages.

Features of PS-algol.

PS-algol [ATKI83, ATKI85, PSAL86] is a block-structured persistent programming language. It incorporates the following features:

Orthogonal Persistence. All PS-algol data objects are manipulated in the same way, irrespective of their persistence. The PS-algol environment includes a Persistent Object Management System [COCK84, CAMP86], which handles all the details of data storage. Data to be stored is organised into 'databases' and any object reachable from the top level of a database will be dragged into backing store as part of that database, when a **commit** command is given. Data are copied to active memory incrementally as references to data objects are dereferenced.

The Universal Pointer Type. As well as a number of scalar base types, procedures and a constructor for vectors of objects of the same type, PS-algol contains a constructor for record-like objects. These may contain any number of fields, each of which may have any PS-algol type. The references to the union of objects that may be constructed in this way have a common type, **pntr**. This allows the programmer a degree of polymorphism, in that values of type **pntr** may be tokens for instances of any existing structure class and therefore objects of different types can be passed along the same route, or referenced from the same location. Type checking is still rigorous, although it does not occur until a **pntr** is dereferenced, prior to performing some operation on the referend. All other type checking is performed at compile time.

First-class procedures. Procedures are first-class objects, in that they may be manipulated like any other object. They may be: assigned to variables; used as the arguments or produced as the result of another procedure; and, most importantly, stored in a database just like any other data value [ATKI86b]. The implication of the latter is that, having been designed in a modular fashion, a program can be developed incrementally. Each module can be coded and tested separately and, as will be seen, different versions of a module can be simultaneously available. Experiments can be run which determine the most effective version and more than one version can be left in the system. This leads to flexibility. It also permits the development of system libraries of procedures and allows the access to data to be limited to a set of procedures, forming an abstract data type (ADT) and allows active data to be modelled [COOP87].

A Callable Compiler. PS-algol contains, as a library function, a call to the compiler. This means that a program, during its run, can construct a procedure as a string and then compile that string and apply the resulting procedure. This is extremely useful as, while the type system of PS-algol is strict (allowing early detection of data misuse), the callable compiler enables a procedure which is truly polymorphic to be written. The structure of such a procedure is: given an object of any type, examine its type, build a procedure which handles such a type, compile it and run it against the input object. The cost of compilation can be recovered if the procedure is stored and often re-used when objects of the same type are encountered.

Indexed Objects. There exists in PS-algol a data structure in the form of a table - a set of pairs of keys and associated structures, accessed through a universal pointer. This provides instances of adaptive index structures.

Graphics Facilities. The language has bit map, multifont text and line drawing graphics facilities. The implications of this for the production of good user interfaces will not be discussed in this paper, but machine independence is derived from having good tools for producing interfaces within the language. Furthermore, graphical data can be modelled with the same ease as textual and numerical data. [MORR86] describes the graphics facilities in more detail.

A Uniform Portable System. PS-algol aims to provide a uniform environment within a number of systems. At present, implementations exist for the UNIX systems on VAX, ICL PERQ and SUN computers, as well as for the Apple Macintosh and within VME on the ICL 3900 series machines. In each of these, the program developer needs only to know PS-algol and has no need to understand the details of the underlying system.

Binding in PS Algol

To sum up, PS-algol gives the programmer a uniform view of data objects. Long-term and short-term objects are handled in the same way, as are numerical, textual and graphical data and program modules. On the other hand, the availability of the universal pointer type and the callable compiler lets the programmer choose when binding should take place. Arguments for the desirability of a range of options on binding time are given in [ATKI87] - here we show how that range of options may be exploited.

In languages such as Poly and Galileo, the program is completely and statically bound at compile time. In PS-algol, there are three times when binding could take place:

- The program can be written so that everything is bound statically at compile-time.

- Using the universal pointer, the binding may be deferred until an object is actually dereferenced. The program will pass an object about and check its type when fields of that object are manipulated. Thus the program is still strictly type-checked, but the type-checking occurs at run-time. In this case, the binding will occur every time a field of the object is dereferenced.

- Using the callable compiler, the binding of data to program may be made at any time between the receipt by the program of a description of the type of a data object and the first use of such an object. It will then be bound once and for all to structures which are specific to data of this type. For instance, a database mangement system could organise the binding ar any time between receiving the database schema and the first attempt to populate the database. This opens up the attractive alternative of supplying the schema one day and having the compilation of efficient storage and retrieval modules performed automatically overnight by a daemon process which checks the persistent store to find any object types waiting to be bound to the program.

The choice made between these alternatives will depend upon the application. In some cases, it is necessary to choose to defer the binding and so use the universal pointer. Usually, however the preferred method would be to factor out the binding process by binding as soon as possible, using the callable compiler.

EFDM: The Extended Functional Data Model.

EFDM is an implementation of the Functional Data Model (FDM) as described by Shipman[SHIP81] constructed by Krishna Kulkarni ([KULK83], [KULK86], [KULK87]). The FDM models data as sets of entities and functions relating the entities. Kulkarni's initial attempt at implementation used the PASCAL language. However, this required interfacing the system to a low-level data management system and when PS-algol became available, he re-implemented EFDM entirely in PS-algol. There was a reduction in the amount of source code to about a third compared with the earlier PASCAL version. Appendix A describes the implementation in some detail.

Among the benefits identified by Kulkarni were:

- the organisation of data movement being handled by the system;
- the reduction in data misuse due to type security;
- the ability to organise the data in a uniform way through PS-algol's universal pointer type;

and
- an increase in speed of access to database items due to efficient heap management

The construction of the system is much simplified by having user data and meta-data stored in the same way, thus allowing the functions of the database handler to be used for both. There is a universal PS-algol structure for storing the information about each function and another universal structure for the data for each entity. The base function data are explicitly stored in container structures, which are referred to via pointer fields in the entity structures. This simple mechanism permits a degree of polymorphism, in that the result of an EFDM function may be referenced in a uniform way whatever its type. If the function is a single valued function whose result is a string, the pointer will point to a string container. If the function is multi-valued, the pointer will point to a list of values.

There is also a saving in storage space since there is no need to store a key with each object in PS-algol. The pointer to the the object is unique and consistent and therefore may be used as the internal identifier for the entity. Wherever the data resides, it will always be referred to by the same pointer value. All objects and sub-objects of the system are referred to via PS-algol pointers. Preservation of all of the data for an object merely requires that a pointer to the object be placed in a database - all the sub-objects follow it into the database automatically.

Derived functions are stored in the form of the tree returned from the Syntax Analyser. The Interpreter then uses this tree any time the function is called. Queries and programs may also be saved in the system and these are also stored in a tree form.

Kulkarni could have made yet more gains by using two more facilities offered by the PS-algol system. Firstly, the program as it stands is a single unit of about 3000 lines of code. PS-algol offers the ability to break the program into small modules, compile them separately and store them in the database. This means that the program could be developed incrementally, with consequent savings in compilation time and debugging time. Secondly, the code for queries, programs and derived functions is stored as a parsed tree and is then executed by the interpreter. This is an example of deferred binding. The speed of the system is reduced by this. Using the callable compiler, EFDM could factor out the binding by compiling the code instead. It could transform the tree into a PS-algol program and then compile it and store it in a form which would have a much faster execution speed.

A Database Architecture With Several Query Languages and Data Models.

Another database system was implemented at the University of Edinburgh by Pedro Hepp [HEPP83a, HEPP83b, NORR85]. The goal of this research was the creation of a system which provided a multiplicity of user interfaces to a uniform internal data model. In the system produced by Hepp, the Query Languages provided were: TABLES, a screen oriented query and update language for a relational database; RAQUEL, a relational algebra language, also for querying and updating a relational database; and FQL [BUNE82]. There is also a Report Generator - a document producer, which takes in commands to specify page layout, headings, etc. Appendix B describes the implementation of the system in greater detail.

In his arguments for using PS-algol, Hepp puts forward many of the same reasons as Kulkarni, but his main benefit from using PS-algol is not stated directly, but is implicit in every section of his thesis: the ability to create a program incrementally. He made use of this in four ways (apart from the reduction in compilation time obtained by breaking down source code into small modules). Firstly, he built his system incrementally. At first a very small system was implemented, with crude versions of the modules. Later, he replaced these with more sophisticated versions, using the persistent store to hold the most recent. This enabled him to develop each module separately. As the database access implicitly provided by PS-algol is based on lazy fetching from disc and strict type checking, program construction is performed as necessary by an incremental type-checked linker - the persistent system itself. It is possible for the programmer to arrange to use permanently one particular implementation of the module, or to use the latest version, or one chosen by any other algorithm.

Secondly, once the internal model was put into the persistent store, as many user interfaces as were required could be added, one at a time. In fact, having got the RAQUEL interface working (with all of the modification and debugging of the internal system implied by this), Hepp got the TABLES interface working "in less than a week" and the FQL interface "in approximately one week of work".

Thirdly, in making the decision on which underlying storage structures to use, he could try independently a number of different options before selecting the best one. This was done by replacing the storage handler with a number of variants and testing the resulting system for speed of access, storage requirements and ease of programming. He tested whether to represent a relation by lists or vectors and whether to represent tuples as strings, vectors of strings, vectors of pointers or as a list of pointers. His analysis led him to a different choice than Kulkarni: he represented his tuples as a vector of strings. This alternative common requires a set of procedures to translate between strings and other types. The application of these translation procedures is equivalent to using the fields of a structure and so is another technique for deferring binding in that most of the program can manipulate the data without knowing what type it is.

Finally, he used the persistent store to record patterns of usage of the various interfaces and modified them to overcome user problems. For instance, certain inelegancies in the syntax of RAQUEL queries were ironed out after examining the pattern of user errors. Furthermore, an analysis of the frequency of usage of objects in the system revealed that "a small set of columns and relations are used more frequently

in query composition than the rest." Clearly this fact could be used to provide more efficient storage and retrieval methods.

As discussed above with EFDM, the availability of the compiler as a system procedure in PS-algol would permit the system to be improved in two ways. Firstly, the storage structures for the data, at present chosen by analysis to be a static structure, could be created dynamically, according to the nature of the data. The next section carries this proposal a good deal further.

Secondly, the analysis of usage, also at present performed off-line, could be performed regularly by the system itself. For example, a dæmon, activated at times of low system usage, would carry out some analysis of the usage of each data object, refer to some normative data on usage, and, if necessary, change the storage to be more appropriate for the pattern of usage found. The user would not notice the change in underlying storage structure, except in that his response times would be improved. These ideas are similar to those put forward by Stocker [STOC73], but the freedom to devise and manipulate any data structure would facilitate experiment and implementation.

A Polymorphic Architecture For Relations.

We present here a new internal model for a database engine, on top of which multiple user interfaces are provided. We take as our starting point a data storage model similar to that used by Hepp, which uses the universal pointer type to provide a polymorphic storage scheme for the tuples of a relation. We amend the interface to take advantage of PS-algol's facility for producing Abstract Data Types and then show how the storage of tuple structures may be tailored to the form of the relation using the compiler function. Thus we show how PS-algol permits polymorphic schemes by use of late or early binding.

After some investigation, we produced a storage scheme for a relation structured as shown in a simplified form in Figure 1. The header for the relation consists of four fields: the relation name; a pointer to the body, which is a doubly linked list of tuples; a pointer to the primary key header (here shown to be a single column, but in general a list of columns); and a pointer to the rest of the column headers of the relation (also pointed to by the primary key). The column headers are organised into a linked list of structures each containing the column's name and a pointer to an instance of an Abstract Data Type defined on domains. In our initial scheme, each tuple consists of a vector of pointers to value containers.

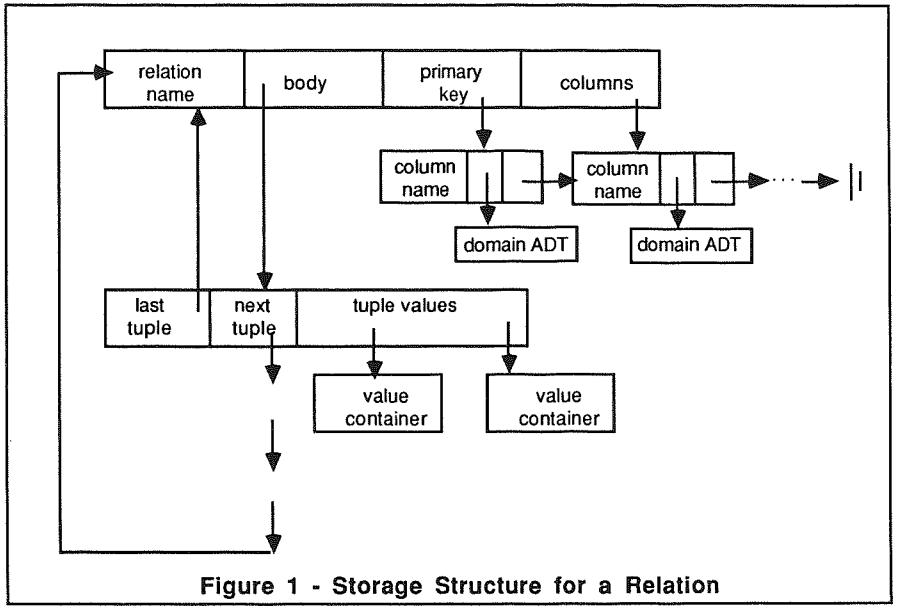

Figure 1 - Storage Structure for a Relation

The interfaces provided to both relations and domains are in the form of Abstract Data Types. Domains are represented by an ADT that contains at least the following operations:

proc(string -> pntr) *putDomVal*	! package a value
proc(pntr -> string) *getDomVal*	! unpack a value
proc(pntr, pntr -> bool) *compDomVal*	! compare two values

Domains are created by calls to a creation procedure by the user interface programs and stored in a table in the persistent store.

Relations are created similarly, using the following procedure -

MakeRel = **proc(string** *description* -> **pntr)**

This is given a description of the relation in the form of a string (containing attribute names, attribute domain types and which attributes are used as the key) and returns a packaged set of procedures, which contain all of the operations permitted on this relation, such as adding a tuple, looking up a tuple from the key, traversing the tuples, checking whether or not the relation is empty, etc. Each call of *MakeRel* binds the **same** code bodies to a new instance of data structures with the **same** definition.

Take as an example the relation

ADDRESS(**string** *name* | **int** *house*, **string** *street*)

in which the field *name* is to be used as the primary key. The construction of a simplified polymorphic representation in PS-algol (corresponding to Figure 1) of the tuple "R. Cooper, 73, Bow Rd." would be

```
structure tuple( pntr last, next; *pntr values )
structure StringContainer( string stringValue )
structure IntContainer( int intValue )
let RC = tuple( ..., ..., @ 1 of pntr [ StringContainer( "R. Cooper" ),
                                        IntContainer( 73 ),
                                        StringContainer( "Bow Rd." ) ] )
```

This creates an instance of the tuple structure, *RC*, consisting of pointers to the adjacent tuples in the list and a vector of pointers to the three field values. The 73 would be de-referenced by

```
RC( values )( 2 )( intValue )
```

which first takes the *values* field of *RC*, takes the second element of the vector and then unpacks it - thus the operation requires three levels of indirection.

The original version of *MakeRel* is shown simplified in Figure 2. The procedure constructs all the information it needs from *description* (looking up the domain information from the domain table). It then creates an instance of the relation structure as *ThisRel*. Then it defines operations on *ThisRel*, of which only the *AddTuple* operation, which adds a new tuple to the relation from values input from the calling program, is shown. Finally, it packages the operation procedures as an ADT for export to the calling program. *AddTuple* merely looks in the body of the relation to find where it should put the tuple, constructs the tuple from the values input and then inserts it. Note that *MakeRel* creates a new instance of the relation structure and then binds a copy of the operation procedures to it.

This version of *MakeRel* can be written once to handle any kind of relation since all the values are stored via pointers. It achieves polymorphism by using the **pntr** type to defer binding. The calling program handles all the packaging and dereferencing of the data, allowing *MakeRel* to be general purpose.


```
structure RelHead( string rname; pntr body, pkey, columns )
structure ColHead( string cname; pntr domType, nextCol )
structure tuple( pntr last, next; *pntr values )
let MakeRel = proc( string description -> pntr )
    begin
        let RelName =                   ! get these from
        let PkeyName =
        let PkeyType =
        let ColNames =                  ! the description
        let PkeyADT = s.lookup( PkeyType, DomainTable )
        let ColTypes = ....
        let ColADTs = ....
        let PkeyComp = PkeyADT ( compDomVal )

        let TheseCols := nil
        for i = 1 to upb( ColNames ) do
            These.cols := ColHead( ColNames( i ), ColADTs( i ), TheseCols )
        let ThisPkey := ColHead( PkeyName, PkeyADT, TheseCols )
        let ThisRel = RelHead( RelName; nil, ThisPkey, TheseCols )

        let AddTuple = proc( pntr PKVal; *pntr ColVals)
            begin
                let before:= ThisRel( body )
                while before ~= ThisRel and
                    PkeyComp (before( values )( 1 ), PKVal) do
                                        before := before( next )
                let after = before( next )
                let NewVals =           ! code to construct a vector of pointers to the
                                        ! input values
                let NewTuple := tuple( before, after, NewVals )
                before( next ) := NewTuple
                after( last ) := NewTuple
            end
        ......                          ! other operations of the ADT

        structure relationADT(          proc( pntr,*pntr ) addTuple;
                              .... )    ! other procedure holders
        relationADT( AddTuple, .... )
    end
```

Figure 2. The Simple Form of the *MakeRel* Procedure.

A New Architecture which Tailors Tuple Structures to Suit the Relation Type.

In the above model, the operation to dereference the "73" field of *RC* required three levels of indirection. The new model proposes to replace the tuple structure given above with one that is more appropriate to the particular relation. We would prefer to create *RC* by

```
AddressTuple( string name; int house; string street )
let RC = AddressTuple( "R.Cooper", 73, "Bow Rd." )
```

and de-reference the 73 by

RC(house)

but to do this, we must bind the *AddressTuple* structure into the program. When writing the system, we do not know that the user is going to create this relation and we certainly do not want to restrict the relations that can be created. A mechanism is needed which operates dynamically (as does our original structure) and produces the more efficient structure above. The *MakeRel* procedure therefore has to use a new strategy.

To use the more efficient second structure and still retain polymorphism, we use a technique introduced in the PS-algol Database Browser ([DEAR87]). This is to construct all those procedures which make use of the tuple structure **at run-time**. The browser allows the traversal of objects in the persistent store by following pointers. Each time a pointer is followed, the resulting structure is examined and, from it, code to display such a structure is constructed during the run of the program.

In the database system, procedures like the one which checks whether a relation is empty can be statically determined, as they only reference the relation header, which is the same for all relations. In contrast, procedures like *AddTuple* cannot be specified in advance as they make use of the tuple structure. Thus we rewrite the parts of *MakeRel* which are concerned with these procedures, as shown in Figure 3.

```
let Tuple.Class = ...                ! get these from
let Field.types = ...                ! the description
let MakeAddTuple =
    " proc( pntr TheRel -> proc( pntr, *pntr ) )
        begin
            structure RelHead( ....      ! as above
            structure " ++ TupleClass ++ "
            let NewAddTuple = proc( pntr PKVal; *pntr ColVals)
                begin
                    let before= ...              ! as before using TheRel( body )
                    let after = before( next )
                    let NewTuple  := tuple( before, after, PKVal( "
MakeAddTuple := MakeAddTuple ++ FieldType(1 ) ++ "Val) "
for i = 1 to upb( FieldName) do
    MakeAddTuple :=MakeAddTuple++",ColVals( "++FieldType(i+1)++"Val)"
MakeAddTuple := MakeAddTuple ++ ")
                    before( next ) := NewTuple
                    after( last ) := NewTuple
                end
            NewAddTuple
    end"
structure ProcBox(proc( pntr -> proc(pntr,*pntr) ) Makeproc)
let CompiledForm = compile( MakeAddTuple,
        ProcBox( proc( pntr -> proc( pntr, *pntr ) ); nullproc )
let AddTuple = CompiledForm( Makeproc )( ThisRel )
```

Figure 3. *MakeRel* Using the Run-time Compiler.

In this second version, *AddTuple* cannot be directly specified. Nor can it be directly specified as a string, since this would not permit the specific instance of the relation to be bound into the procedure. Just adding references to an object called *ThisRel* into the string defining *AddTuple* will not make them refer to the required object, since *AddTuple* must be compiled separately. Instead a procedure-generating procedure, *MakeAdTuple*, itself constructed as a string, takes in a pointer to *ThisRel* and produces a version of*AddTuple* which operates on *ThisRel*.

MakeRel takes in the current relation and generates the string containing the tuple structure, *TupleClass*, and the vector of field types, *FieldType*, from the input description. Then it constructs the *MakeAddTuple* procedure as a string which varies only in the tuple structure and the line of code constructing the tuple. In this line, the values of the fields are unpacked from their containers by dereferencing the field of the container. If the field is an integer field, for instance, it is contained in an *IntContainer*, whose field name is *intVal*. Conventionally the fields of a container structure are always of the form *type* ++"Val", and so can be created by *MakeAddTuple* simply. In the case of the address structure above, *MakeAddTuple* would be as shown in Figure 4.

```
proc( pntr TheRel -> proc( pntr, *pntr ) )
    begin
        structure RelHead( ....      ! as above
        structure tuple( pntr last,next; string name; int house; int value )
        let NewAddTuple = proc( pntr KeyVal; *pntr ColVals)
            begin
                let before= ....
                let after = before( next )
                let NewTuple  := tuple( before, after, KeyVal( StringVal),
                        ColVals( 1 )( IntVal ), ColVals( 2 )( StringVal ) )
                before( next ) := NewTuple
                after( last ) := NewTuple
            end
        NewAddTuple
    end
```

Figure 4. *AddTuple* generated for the address structure.

MakeAddTuple is then compiled and run with *ThisRel* as its argument. It returns the appropriate *AddTuple* procedure as its result. It is at this point that the relation structure is bound to the *AddTuple* code to return a procedure which adds a tuple to **this** relation. This procedure is then packaged as part of the ADT returned by *MakeRel*.

Further Speeding By Memo-ising.

There are some overheads when using this method. Relation creation is a more expensive operation as it involves compilation. Although this should be offset by more efficient access to the relation once it has been created, we can do something to cut down on the need to compile every time a relation is created. Again, we utilise a

technique introduced in the PS-algol Browser. This is to transform the tuple structure definition into a canonical form involving only the types of the columns. Thus the address structure would be referred to as a **string.int.string** structure and the structure defined in *MakeAddTuple* above, would be:

structure *tuple*(**string** *string1*; **int** *int2*; **string** *string3*)

When the address structure is encountered, *MakeRel* refers to a table in the database to find if it has already encountered a structure keyed by "string.int.string". If it has, compiled forms of the procedure generating procedures, like *MakeAddTuple* in the example above, are retrieved from the database and re-used. Otherwise, it will compile new versions and enter them into the table, ready for any other structure, for instance:

structure *student*(**string** *sname*; **int** *snumber*; **string** *class*)

which will be mapped onto the same canonical form and will look up and use the same procedures. Further savings still, are achieved by permuting the column types into a canonical order. This method of "memo-ising" a structure is supported by PS-algol tables.

Conclusions.

We have examined three database systems programmed in PS-algol. EFDM is a single program providing an implementation of the Functional Data Model. The persistent environment frees the programmer from the chores involved in organising backing store. The development of EFDM shows how this speeds program development and coherence. Moreover, the provision of a universal pointer type allowed the bindings to data objects to be deferred and greatly simplified the storage structures involved.

An examination of Pedro Hepp's work showed how he used the persistent store, to develop his system incrementally. The program was divided into manageable modules, each of which was implemented separately. Not only did this make program development faster by reducing compilation time, but allowed him to experiment on the internal model of the data by trying different versions. It also allowed him to provide a number of user interfaces which operate independently of each other. He used the persistent store to record information about system usage, an analysis of which enabled him to make improvements to it. He transformed all of his data types to strings to defer data binding.

Our own work has centred around attempts to increase system efficiency by using a callable version of the compiler to factor out these bindings. We have shown how the "database engine" could be programmed to provide a relation as an Abstract Data Type. Our motive for this was an enforced and formal definition of module boundaries, guaranteeing that module replacement was feasible. We have shown how access to a compiler at run-time has enabled us to generate the ADT, using a more efficient representation as its internal model. Finally, we have shown how the cost of creating a relation can be reduced by a canonical representation of relations, which enable those with the same types to share code.

In summary, we have shown that programming a DBMS in a persistent environment frees the programmer from the time consuming issues involved in organising backing store and allows concentration on more important problems, such as a more efficient access to data and a more ergonomic user interface. We have also shown that the programmer should be provided with a range of options on when the binding of data to the program occurs. In particular, we have shown how the availability of run-time compilation within the implementation language permits storage schemes which are both efficient and type-secure.

Acknowledgements.

The work reported here was funded partly by an ICL URC grant, partly by Alvey/SERC grant GR/D43259 and partly by sponsorship from the Algerian Official Authority. We would also like to thank Professor Tim Merritt and Ms Rosemary McLeish for advice and helpful comments on earlier drafts of this paper and for the useful comments of our colleagues in the Persistent Programming Research Group and of Dr. John Jeacocke during the initial stages of this work.

Bibliography.

ATKI83 Atkinson, MP, Bailey PJ, Chisholm, KJ, Cockshott, WP and Morrison R - "An Approach to Persistent Programming", *The Computer Journal*, 26, 4, (1983), 360-365.

ATKI85 Atkinson, MP and Morrison R - "Procedures as Persistent Data Objects", *ACM TOPLAS* 7, 4, 539-559, (Oct 1985).

ATKI86a Atkinson, MP and Morrison R - "Integrated Persistent Programming Systems", *Proc 19th Annual Hawaii Conference on System Sciences, Jan 7-10, 1986* (ed. B.D. Shriver), Vol IIA, Software, 842-854.

ATKI86b Atkinson, MP, Morrison R and Pratten, GD - "Designing a Persistent Information Space Architecture", *Proc Information Processing 1986*, North Holland Press (Sept 1986) 115-119.

ATKI87 Atkinson, MP and Morrison R - "Binding and Type Checking in Database Programming Languages", *in preparation - see first half of this report.*

BUNE82 Buneman, OP, Frankel, RE and Nikhil, R - "An Implementation Technique for Database Query Languages", *ACM TODS*, 7, 2, June 1982.

CAMP86 Campin, J and Atkinson, MP - "A Persistent Store Grabage Collector with Statistical Facilities", *Persistent Programming Report 29*, Universities of Glasgow and St. Andrews, 1986.

COCK84 Cockshott, WP, Atkinson, MP, Chisholm, KJ, Bailey, PJ and Morrison, R - "POMS: A Persistent Object Mangement System", *Software Practice and Experience*, 14, 1, 49-71, Jan 1984.

CODD79 Codd, EF - "Extending the Relational Model of Data to Capture More Meaning", *ACM TODS*, 4, 4, Dec 1979.

COOP87 Cooper, RL - "Applications Programming in PS-Algol", *Persistent Programming Report 25*, Universities of Glasgow and St. Andrews, 1987.

DEAR87 Dearle, A. and Brown, A.L. - "Safe Browsing in a Strongly Typed Persistent Environment", *Persistent Programming Report* 33 , Universities of Glasgow and St. Andrews, 1987 - to appear in *The Computer Journal*, 1988.

HEPP83a Hepp PE - "A DBS Architecture Supporting Coexisting Query Languages and Data Models", *Ph. D. Thesis*, University of Edinburgh, 1983.

HEPP83b Hepp PE - "A DBS Architecture Supporting Coexisting User Interfaces: Description and Examples", *Persistent Programming Report 6*, Universities of Glasgow and St. Andrews, 1983.

KULK83 Kulkarni, KG - "Evaluation of Functional Data Models for Database Design and Use", *Ph. D. Thesis*, University of Edinburgh, 1983.

KULK86 Kulkarni, KG and Atkinson, MP - "EFDM: Extended Functional Data Model", *The Computer Journal*, 29, 1, (1986) 38-45.

KULK87 Kulkarni, KG and Atkinson, MP - "Implementing an Extended Functional Data Model Using PS-algol", *Software Practice and Experience*, 17, 3, March 1987, 171-185.

MORR86 Morrison R, Dearle, A, Brown, AL, and Atkinson, MP - "An Integrated Graphics Programming Environment", *Computer Graphics Forum*, 5, 2, June 1986, 147-157.

NORR85 Norrie, M - "The Edinburgh Node of the Proteus Distributed Database System", *University of Edinburgh Internal Report*, CSR-191-85.

PSAL86 "The PS-algol Reference Manual - Third Edition", *Persistent Programming Report 12*, Universities of Glasgow and St. Andrews, 1987.

SHIP81 Shipman, DW - "The Functional Data Model and the Data Language DAPLEX", *ACM TODS*, 6, 1, 140-173, March, 1981.

STOC73 Stocker P. and Dearnley, PA - "Self Organising Data Management Systems", *The Computer Journal*, 16, 2, (1973), 100-105.

Appendix A. Implementation Details of EFDM.

Overview.

The Functional Data Model treats data as sets of entities and functions relating the entities. Meta-data and user data are stored as functions in a uniform way. Entities are created by the application of zero-argument functions. Thus

declare person( ) ->> **entity**

specifies a function which creates *person* entities.

Entities may be arranged in type hierachies as in

declare student( ) ->> person

which defines a second type of entity creating function. Entities created by this function will inherit all of the properties of *person* and may have more properties of their own defined on it.

Properties of entities are also defined as functions. These take the entity type(s) as arguments and return the property value as a result. For instance:

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

which defines a single-valued function on person entities.

As this function has been specified with a **declare** command, it is a *base function*, that is the data associated with it is explicitly stored. Further *derived functions* may also be specified as follows:

define tutor(student) -> staff(tutorial(student))

where the **define** command indicates that values of the function may be derived from other functions by applying the functions on the right hand side of the arrow.

Either type of function may take more than one argument and either type of function may be single- or multi-valued. Functions are specified as being single- or multi-valued by putting either one arrow (as in *name* above) or two arrows (as in *person*) in the line defining them.

Further constructs are provided to permit the definition of constraints on entities and of views on the data. There are some aggregate functions supplied, such as **count** and **maximum**. There is a simple retrieval language, illustrated by the query:

for each s **in** student **print** cname(s), sname(s);

and updating commands such as

for a new s **in** student
let cname(s) = 'Moyana'
let sname(s) = 'Johns';

and **delete the** s **in** student **such that** cnames(s) = 'Moyana' **and** sname(s) = 'Johns';

Queries and updating programs may be stored by name and re-used as required.

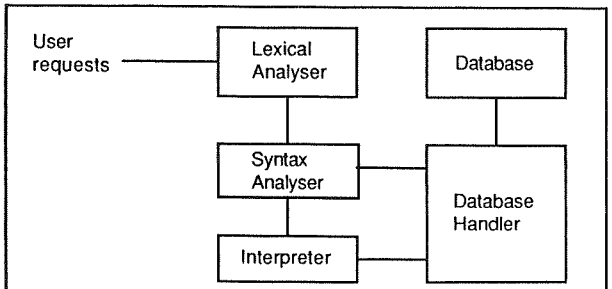

Figure A - Block Diagram of EFDM.

The Implementation.

The EFDM system consists of a single PS-algol program of some 3000 lines. The layout of the program is shown in Figure A. A user request passes through the Lexical Analyser and the Syntax Analyser, emerging as a syntax tree. Schema modification requests are handled by direct calls to the Database Handler, while data update and retrieval requests pass through the Interpreter, which in turn calls the Database Handler as required. The construction of the system is much simplified by having user data and meta-data stored in the same way thus allowing the functions of the Database Handler to be used for both.

Three kinds of data object are stored in EFDM:

- **entities**;
- **functions**;

and

- **programs**, which include updates, queries and the code for the body of derived functions.

Programs are stored as binary parse trees, which consist of triples at each node including the operation at this node and pointers to left- and right-subtrees.

Entities are stored in lists with the other objects produced by the same entity creating function. Each entity is a PS-algol structure with four fields:

- a pointer to the function which generated it;
- a pointer to the next entity in the list;
- a pointer to a super-type entity;

and

- a vector of pointers to attribute values.

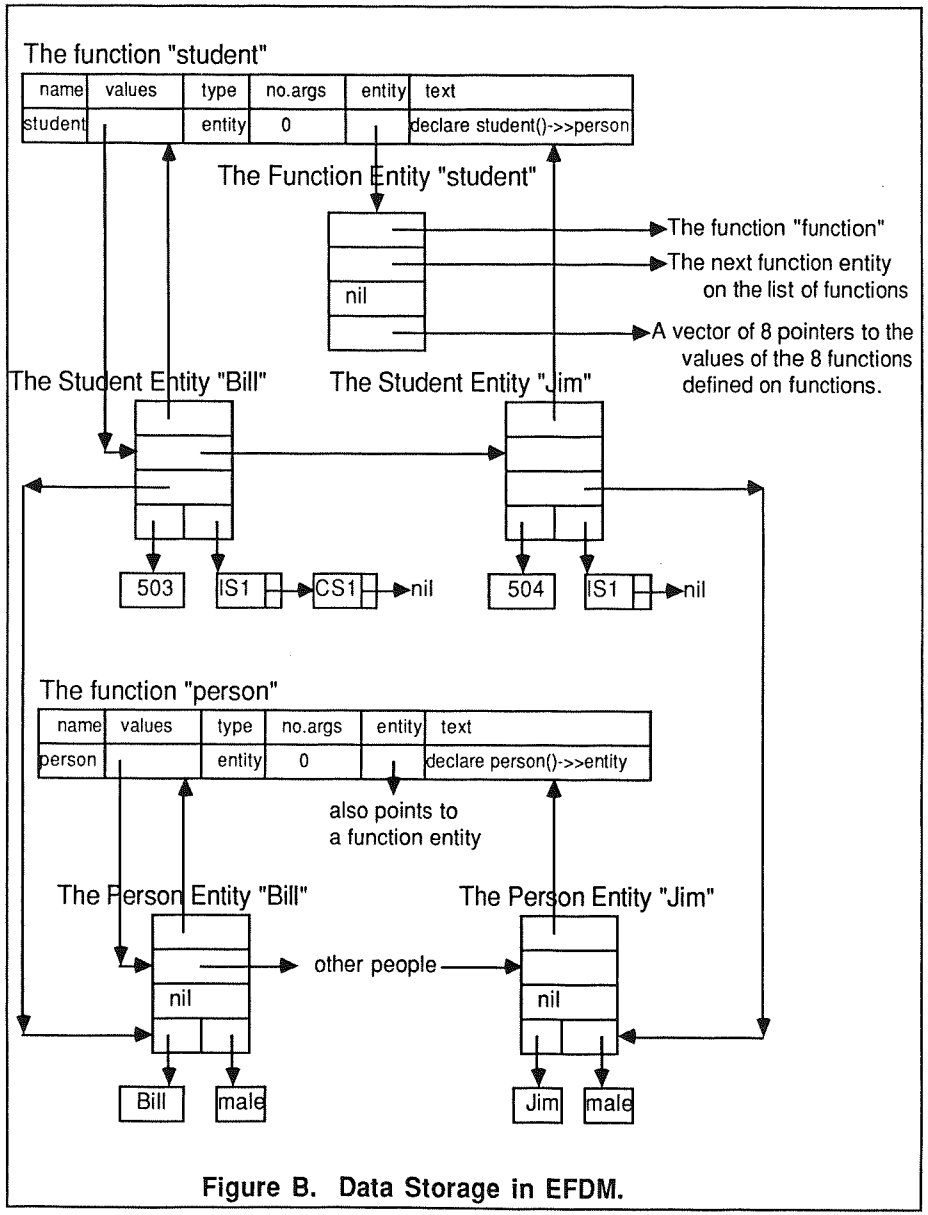

Figure B. Data Storage in EFDM.

To illustrate this consider Figure B which shows part of the database after the following information has been entered:

declare person() ->> **entity**
declare name(person) -> **string**
declare sex(person) -> **string**
declare student() -> person
declare matric(student) -> **integer**
declare courses(student) ->> **string**
and then two students Bill, male, numbered 503, taking courses "IS1" and "CS2" and Jim, male, 504, taking only "IS1" are introduced.

The figure shows four entity objects, two for each student - one being his student entity and one being his associated person entity. Thus the four pieces of information about each student may be found either from his student entity or by chasing the pointer to the super-entity to his person entity.

The values are packaged in various structures - there is one for instance for packaging integers, another for strings. The universal pointer type of PS-algol allows references to any of these to be held within a structure of the same type (a vector of pointers) and further allows values of multi-valued functions to be held in the same way. For instance, *name* is a single-valued function whose result is a string, so the pointer to a name will point to a string container. The function, *courses*, on the other hand is multi-valued, and so the pointer to the courses points to a list of values. Thus a totally polymorphic structure has been imposed on the data and the bulk of the program can handle entities without needing to know which type they are.

Notice that, unlike many DBMS structures, no identifier is stored with the entity. The pointer to the entity is unique and consistent and so may be used as the internal identifier for the entity. Wherever the data actually resides, it will always be referred to by the same pointer value.

The values of multi-argument functions are also stored as a list of values, each value having associated with a list of its arguments and a pointer to the result value. Thus if we declare the following base function:

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

then the data that Moyanna Johns got an "A" for course "IS1" is stored as shown in Figure C.

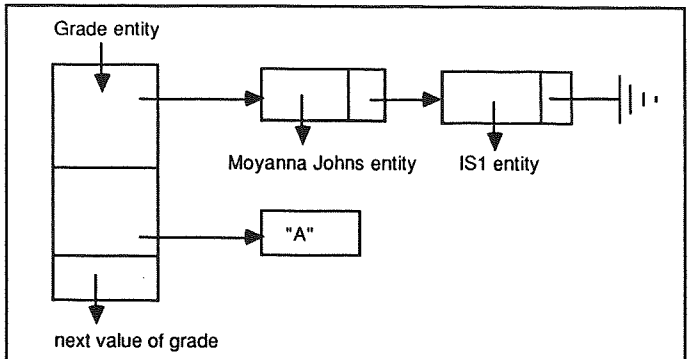

Figure C - Multi Argument Function Storage

The data stored about **functions** (as distinct from the values of the function) are stored in two separate structures, a function structure and an entity structure. The function structure has a number of fields, including:

- a string containing the name;
- a string containing the type of the function;
- a string containing the text specifying the function;
- an integer containing the number of arguments;
- a pointer to its associated entity structure;

and
- a pointer to the values of the function, which is either:
  - a program, in the case of derived functions;

or
  - a list of entities, in the case of base functions.

The user has no access to this structure. However, each function also has an entity structure associated with it and this allows functions to be declared with functions as arguments, such as

declare name(function) -> **string**

which returns the name of a function. Eight such functions are automatically inserted into the database when the system is started up. The data for these functions are stored in exactly the same way as for the base functions defined on students, that is the values are associated with the function entity. For instance, there will an entity associated with the function *grade* and one of the entries in its function value vector will be a pointer to the string "grade".

Appendix B. Implementation Details of RAQUEL.

Overview.

The model produced by Hepp contains three components: an Internal Conceptual Schema (ICS), containing meta-data; an Internal Data Manipulation Language (IDML); and an Internal Query Language (IQL). A modified subset of the extended relational mode, RM/T [CODD79] with the following architecture was proposed, illustrated in Figure D.

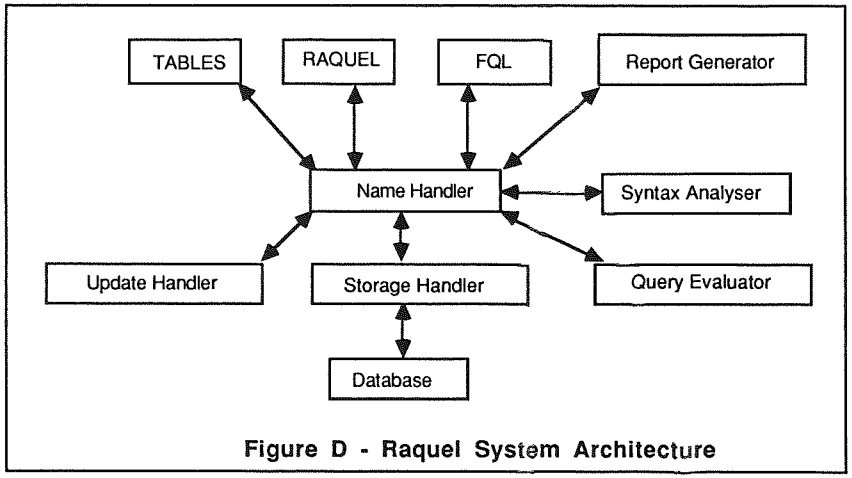

Figure D - Raquel System Architecture

The Query Language interfaces interact with the database only through the Name Handler, which translates names into internal identifiers. It uses the ICS to make all its checks and this itself has been organised as a set of relations, so that as with EFDM, the same procedures can be used to access meta-data and user data. The ICS starts off with two relations, one containing a list of all the relations in the system and one containing a list of attributes. User-defined relations and attributes are gradually added to these. The other components shown include the Storage Handler (SH), the Query Evaluator (QE) and the Update Handler (UH). Th SH controls the creation, maintainance and deletion of relational structures, such as relations and tuples. The QE processes queries specified in the IQL and the UH ensures database consistency by monitoring update requests to detect integrity violations.

The Storage Structures.

All data and meta-data are stored in the form of **relations**. The data for a relation is stored in a doubly linked list of **tuples**, with a **relation header** containing summary information at the head of the list. Included in the summary information is some information about each attribute or column of the relation, stored in a **column header**.

The data stored in a relation header includes:

- a pointer to the first tuple;
- a pointer to the last tuple;
- the order (number of attributes) of the relation;
- the cardinality (number of tuples) of the relation;
- a vector of pointers to a header for each column of the relation;

and
- a boolean indicating whether the relation is temporary or not.

Each column header contains the four fields:

- a string containing a default value for the columns;
- a string containg the type of the field;
- a pointer to a constraint structure;

and
- an integer defining the number of characters that the column will use when being printed.

Each tuple contains three fields:

- a pointer to the next tuple;
- a pointer to the last tuple;

and
- a vector of **strings** containing the attribute values.

Note that Hepp has used a different strategy for getting a kind of polymorphism into his program - every value is represented as a string. His program relies on there being a few types of columns (integer, real, boolean, string, date and time) and he provides, for each type, a pair of translation procedures - string.to.type and type.to.string. This is a fairly inefficient method of storing data, both in terms of space and in terms of time to search and to dereference data.

As Kulkarni did, Hepp stores meta-data in the same way as ordinary data. In this case, he provides two relations: "tables", which holds one tuple for each relation; and "columns" which holds one tuple for each column of each relation.

The "tables" relation has three attributes:

- **tabname** - the relation's name;
- **protection** - a string specifying disallowed operations, such as deleting a column, adding a row;
- **sortorder** - which is one of:
  - "n" - no order;
  - attribute name = "a" - ascending order on the given attribute;
  - attribute name = "d" - descending order on the given attribute.

The "columns" relation has five attributes:

- **colname** - the name of the column;
- **tabname** - the name of the relation of which it is a column;
- **type** - one of "i", "r", "s", "b", "d" or "t";
- **default** - a default value for the attribute;
- **constraint** - one of:
  "unique"
  "isin relname.colname" - must already be present in another relation;
  a predicate;
  or "n" - no constraint.

At start up time, the system contains just these relations, as shown in Figure E. As can be seen, tables has 2 tuples, while columns has 8.

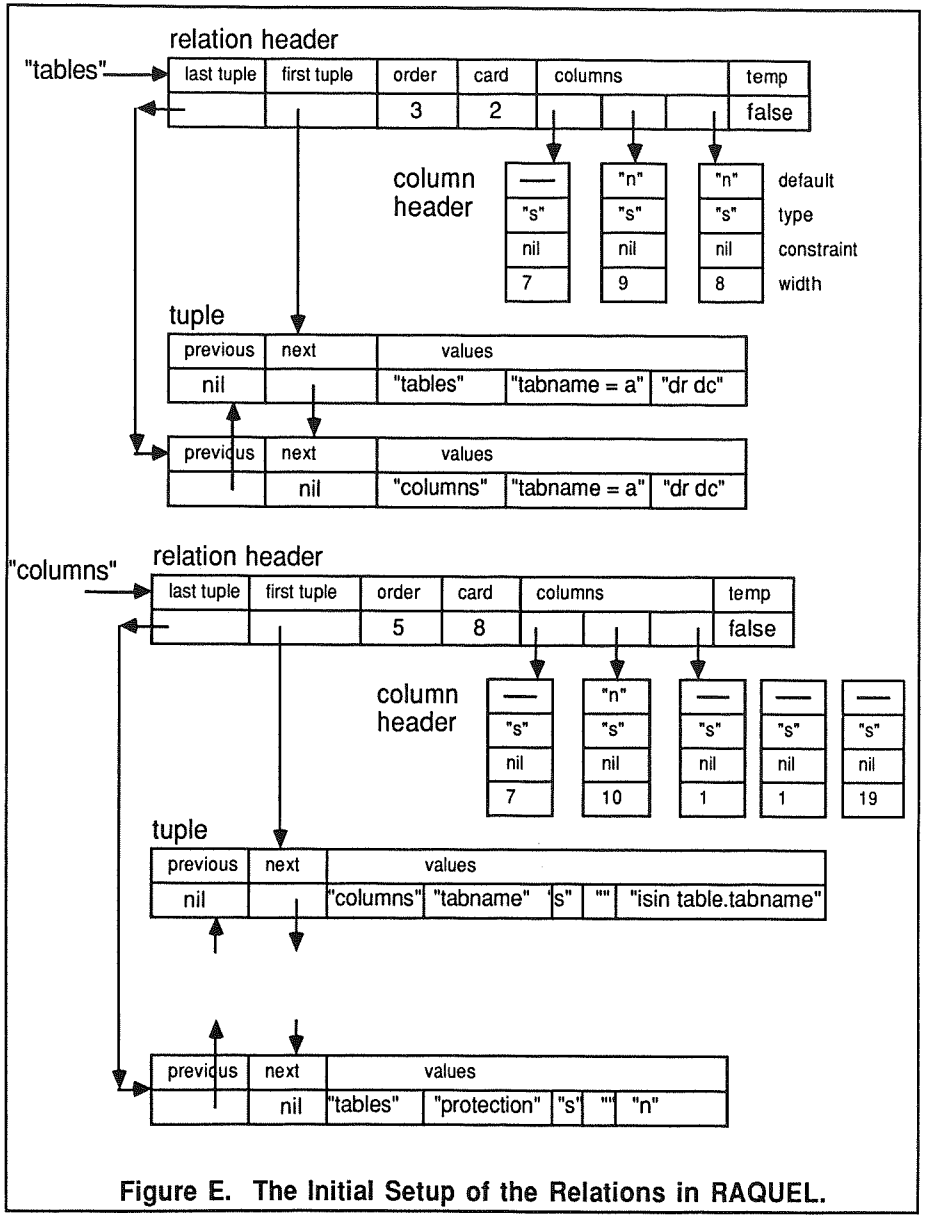

Figure E. The Initial Setup of the Relations in RAQUEL.

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, N0. 2, (Summer 1986) - see PPRR-23-86.

Kulkarni, K.G. & Atkinson, M. P.
"Implementing an Extended Functional Data Model Using PS-algol", Software - Practise and Experience, Vol. 17(3), 171-185 (March 1987)

Cooper, R.L. & Atkinson, M.P.
"The Advantages of a Unified Treatment of Data", Software Tool 87: Improving Tools, Advance Computing Series, 8, 89-96, Online Publications, June 1987.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following theses, for the degree of Ph. D. unless otherwise stated, have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

J. Livingstone
Graphical Manipulation in Programming Languages: Some Experiments, M.Sc., University of Glasgow, 1987

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those which have been produced at 9th July 1986. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R	Presently no longer available
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-87	PS-algol Reference Manual - fourth edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-25-87	Applications Programming in PS-algol - Cooper, R.L.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-86	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00

PPRR-34-87	Constructing Database Systems in a Persistent Environment - Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00
PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00
PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00
PPRR-38-87	Flexible Incremental Bindings in a Persistent Object Store - Morrison, R., Atkinson, M.P. and Dearle, A.	£1.00
PPRR-39-87	Polymorphic Persistent Processes - Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J.	£1.00
PPRR-40-87	Andrew, Unix and Educational Computing - Hansen, W. J.	£1.00
PPRR-41-87	Factors that Affect Reading and Writing with Personal Computers and Workstations - Hansen, W. J. and Haas, C.	£1.00
PPRR-42-87	A Practical Algebra for Substring Expressions - Hansen, W. J.	£1.00
PPRR-43-87	The NESS Reference Manual - Hansen, W. J.	£1.00