Constructing Database Systems in a Persistent Environment

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

[1] - Dept of Computer Science, University of Glasgow, Lilybank Gdns, Glasgow, G12 8QQ

[2] - Dept 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 use the structure used by the program to organise the data in the backing store. 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 the Functional Data Model; a relational system, supporting a number of user interfaces; and a relational system utilising 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. The PS-algol type system 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 implementations, 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 a number of 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 may pass an object about and check its type only 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 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 management system could organise the binding at 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 dæmon 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 by use of 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: 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.

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 PS-algol structure for storing the information about each function and another universal structure for the data for each entity and these are used for system and user-defined functions alike. 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, queries and programs 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.

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, but 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 greater 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; FQL [BUNE82]; and a Report Generator.

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 started with a very small system consisting of crude versions of the modules and 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 independently try 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, which requires a set of procedures to translate between strings and other types. The application of these translation procedures is equivalent to dereferencing

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 its type.

Finally, he used the persistent store to record patterns of usage of the various interfaces and modified them to overcome users' problems with the system. 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.

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, currently 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 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 change to a more appropriate structure for the pattern of usage found. The user would not notice the change in underlying 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, using 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. In the next section we will 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.

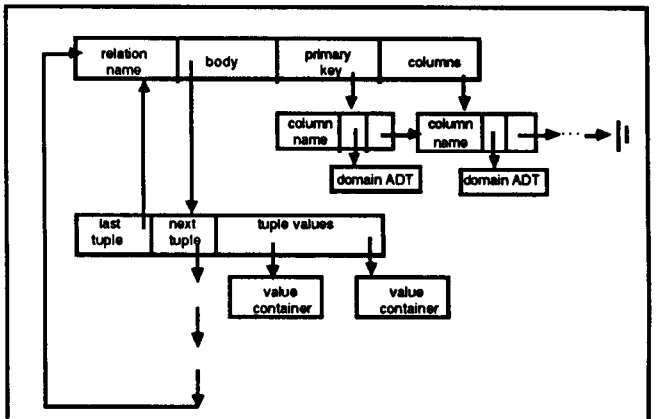

Figure 1 - Storage Structure for a Relation

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)
```

which 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

ADDR(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

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 empty 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 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 of 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 PkADT= s.lookup( PkeyType , DomainTable)
    let ColTypes = ....
    let ColADTs =....
    let PkeyComp = PkADT ( compDomVal )

    let TheseCols := nil
    for i = 1 to upb( ColNames ) do
      These.cols := ColHead ( ColNames(i ),
                              ColADTs( i ),
                              TheseCols )
    let ThisPkey := ColHead ( PkeyName,
                    PkADT, 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 (
        prc(pntr,*pntr) addTuple;
          .... ) ! other procedure holders
    relationADT ( AddTuple, .... )
  end
```

Figure 2 The First 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

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

and de-reference the 73 by

```
RC( house )
```

but to do this, we must bind the *AddrTuple* 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 which use the tuple structure, like *AddTuple*, cannot be specified in advance. Thus we rewrite the parts of *MakeRel* which are concerned with these procedures, as shown in Figure 3.

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

```
let Tuple.Class = ...   ! get these from
let Field.types  = ...  ! the description
let MakeAdTup =
    " 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, KeyVal( "
MakeAdTup:= MakeAdTup ++ FieldType (1) ++ " Val)"
for i = 1 to upb( FieldName ) do
   MakeAdTup :=MakeAdTup ++ ",ColVals( " ++
                    FieldType (i+1) ++ " Val)"
MakeAdTup:= MakeAdTup ++ ")
            before( next ) := NewTuple
            after( last ) := NewTuple
          end
        NewAddTuple
   end"
structure ProcBox(
            proc( pntr-> proc(pntr,*pntr) ) Makeproc)
let EmptyBox :=ProcBox(
            proc( pntr-> proc( pntr, *pntr) ); nullproc
let CompiledForm= compile( MakeAdTup,EmptyBox)
let AddTuple = CompiledForm( Makeproc )( ThisRel )
```

Figure 3. *MakeRel* Using the Callable Compiler.

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 *MakeAdTup* 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 *MakeAdTup* simply. In the case of the address structure above, *MakeAdTup* would be as shown in Figure 4.

MakeAdTup 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*.

```
proc( pntrTheRel -> proc( pntr, *pntr) )
```

```
begin
  structure RelHead ( ....      ! as above
  structure tuple ( pntr last ,next; string name;
                                  int house; int value )
  let NewAddTuple = proc( pntr KeyVal ;
                                  *pntrColVals )
    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 ADDR structure.

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 PA-algol browser, which 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 *MakeAdTup* above, would be:

structure *tuple* **(string** *str* 1; **int** *int2* ; **string** *str3*)

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 *MakeAdTup* 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** *sno*; **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 it 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 *Persistent Programming Report 34*, Universities of Glasgow and St. Andrews, 1987.

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 Garbage 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 Realtional Model of Data to Capture More Meaning",*ACM TODS*, 4, 4, Dec 1979.

COOP87 Cooper, RL - "Applications Programming inPS-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 Designand 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 ExtendedFunctional 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.