Alan Dearle.

University of Edinburgh
Department of Computer Science

James Clerk Maxwell Building
The King's Buildings, Edinburgh

University of St Andrews
Department of Computational Science

North Haugh
St Andrews, Fife

PS-Algol Papers

PPR-2-83
May, 1983

PS-ALGOL PAPERS

M.P. Atkinson[+], P.J. Bailey[*], K.J. Chisholm[+], W.P. Cockshott[+], R. Morrison[*]

[+] Department of Computer Science
University of Edinburgh,
Mayfield Road,
Edinburgh. EH9 3JZ

[*] Department of Computational Science
University of St Andrews,
North Haugh,
St Andrews. KY16 8SX

PERSISTENT PROGRAMMING
RESEARCH REPORT 2

Preface

This research report presents a compendium of recent papers on PS-algol, produced by the Data Curator research team working at the Universities of Edinburgh and St Andrews. The work is supported by a number of SERC grants and by ICL who are also collaborating in the research.

The papers presented are:

Part 1	An approach to Persistent Programming	To be published in the Computer Journal. To appear 1983.
Part 2	An approach to Persistent Programming	Presented at the Digital Workshop on Databases and Programming languages, Boston, 27-29 April, 1983.
Part 3	PS-algol : a language for persistent programming	Submitted to the Australian National Computer Conference 1983.
Part 4	Some thought about the problems with persistent programming	Presented at the workshop on Database Programming Language interfaces, organised by P. Buneman, University of Pennsylvania, October 1982 (To appear modified in the proceedings of that workshop.)
Part 5	PS-algol library routines: Temporary user specification.	

These papers are closely related and contain overlapping material. It is therefore recommended that the reader reads part 1 thoroughly first, and then selects from the others according to his tast and the following guide:

Part 2	Contains further elaboration of the potential effects and design issues of persistent programming.
Part 3	Contains further discussion of our data type PICTURE
Part 4	Is an earlier paper, which indicates how the language has evolved recently and identifies the major problems still facing researchers working in this area.
Part 5	Gives specific details relevant to programmers using the current version of PS-algol.

Those wishing to cite these papers are asked to refer to the published versions if possible, and to list the set of authors in full on each occasion.

A list of other publications, and pending publications by the Data Curator Group is inlucded at the back of this volume.

PART 1

An approach to persistent programming

Accepted for publication in the Computer Journal, 1983.

An approach to persistent programming

M.P.Atkinson[+], P.J.Bailey[*], K.J.Chisholm[+], W.P.Cockshott[+] and R.Morrison[*]

[+] Department of Computer Science, University of Edinburgh, Mayfield Rd., Edinburgh EH9 3JZ

[*] Department of Computational Science, University of St.Andrews North Haugh, St.Andrews KY16 8SX

Abstract

This paper presents the identification of a new programming language concept and reports our initial investigations of its utility. The concept is to identify persistence as an orthogonal property of data, independent of data type and the way in which data is manipulated. This is expressed by the principle that all data objects independent of their data type should have the same rights to persistence or transience. We expect to achieve persistent independent programming, so that the same code is applicable to data of any persistence. We have designed a language PS-algol by utilising these ideas and constructed a number of implementations. The experience gained is reported here, as a step in the task of achieving languages with proper accommodation for persistent programming.

Introduction

The long term storage of data has been of concern to programming language designers for some time. Traditional programming languages provide facilities for the manipulation of data whose lifetime does not extend beyond the activation of the program. If data is required to survive a program activation then some file I/O or database management system interface is used. Two views of data evolve from this. Data can either be classed as short term data and would be manipulated by the programming language facilities or data would be long term data in which case it would be manipulated by the file system or the database management system (DBMS). The mapping between the two types of data is usually done in part by the file system or the DBMS and in part by explicit user translation code which has to be written and included in each program.

These different views of data are highlighted when the data structuring facilities of programming languages and database management systems are compared. Database systems have developed relational, hierarchical, network and functional data models [20,11,19,30,27] whereas programming languages may have arrays, records, sets, monitors [13] and abstract data types [18].

That there are two different views of data has certain disadvantages. Firstly in any program there is usually a considerable amount of code, typically 30% of the total [14], concerned with transferring data to and from files or a DBMS. Much space and time is taken up by code to perform translations between the program's form of data and the form used for the long term storage medium. This is unsatisfactory because of the time taken in writing and executing this mapping code and also because the quality of the application programs may be impaired by the mapping. Frequently the programmer is distracted from his task by the difficulties of understanding and managing the mapping. The translation merely to gain access to long term data should be differentiated from translations from a form appropriate to one use of the data to a form suitable for some other algorithms. Such translations are justified when the two forms cannot coexist and there is a substantial use of both forms. The second major disadvantage is that the data type protection offered by the programming language on its data is often lost across the mapping. The structure that might have been exploited in a program to aid comprehension is neither apparent nor protected and is soon lost.

We seek to eliminate the differences between the DBMS and programming language models of data. This can be done by separating the issue of what data structures are best from the issue of identifying and managing a property of data we call persistence. This is the period of time for which the data exists and is useable. A spectrum of persistence exists and is categorised by

1. transient results in expression evaluation.
2. local variables in procedure activations.
3. own variables, global variables and heap items whose extent is different from their scope.
4. data that exists between executions of a program.
5. data that exists between various versions of a program.
6. data that outlives the program.

The first three persistence categories are usually supported by programming languages and the second three categories by a DBMS, whereas filing systems are predominantly used for categories 4 and 5. We report here on the PS-algol system which is a step in the search for language and database constructs which meet the needs of persistent data and hence obviate the need for the programmer to cope with the problems described above. PS-algol uniformly supports programming for an increased range of persistence. The system is implemented and is being actively used in a number of projects. Here we will describe the language design decisions in implementing persistence, the underlying implementation problems, the results obtained and some thoughts for the future.

The language design method

As a first attempt at producing a system to support persistence we hypothesised that it should be possible to add persistence to an existing language with minimal change to the language. Thus the programmer would be faced with the normal task of mastering the programming language but would have the facility of persistence for little or no extra effort.

The language chosen for this was S-algol [21,12], a high level algol used for teaching at the University of St Andrews. This decision was made by the University of Edinburgh team after some trouble with attempts at Algol 68 [34] and Pascal [36] and resulted in the two teams collaborating on the project.

S-algol stands somewhere between Algol W [35] and Algol 68. It was designed using three principles first outlined by Strachey [29] and

Landin[17]. These are

1. The principle of correspondence
2. The principle of abstraction
3. The principle of data type completeness

The application of the three principles in designing S-algol is described elsewhere [22]. The result is an orthogonal language whose "power is gained from simplicity and its simplicity from generality" [33]. Here we are interested in data and the S-algol universe of discourse can be defined by

1. The scalar data types are integer, real, boolean, string, picture and file.
2. For any data type T, *T is the data type of a vector with elements of type T.
3. The data type pointer comprises a structure with any number of fields, and any data type in each field.

The world of data objects can be formed by the closure of rule a under the recursive application of rules b and c.

The unusual features of the S-algol universe of discourse are that it has strings as a simple data type [24], pictures as compound data objects [23] and run time checking of structure classes. The picture facility allows the user to produce line drawings in an infinite two dimensional space. It also offers a picture building facility in which the relationship between different sub-pictures is specified by mathematical transformations. A basic set of picture manipulating facilities along with a set of physical drawing attributes for each device is defined. A pointer may roam freely over the world of structures. That is a pointer is not bound to a structure class. However when a pointer is dereferenced using a structure field name, a run time check occurs to ensure the pointer is pointing at a structure with the appropriate field.

Together with our hypothesis of minimal change to the programming language we recognise certain principles for persistent data.

1. persistence independence : the persistence of a data object is independent of how the program manipulates that data object and conversely a fragment of program is expressed independently of the persistence of data it manipulates. For example, it should be possible to call a procedure with its parameters sometimes objects with long term persistence and at other times only transient.

2. persistence data type orthogonality : In line with the principle of data type completeness all data objects should be allowed the full range of persistence.
3. The choice of how to provide and identify persistence at a language level is independent of the choice of data objects in the language.

A number of methods were investigated to identify persistence of data. Some involved associating persistence with the variable name or the type in the declaration. Under the rule of persistence independence these were disallowed. S-algol itself helped to provide the solution. Its data structures already have some limited notion of persistence in that the scope and extent of these objects need not be the same. Such structures are of course heap items and their useability depends on the availability of legal names.

This limited persistence was extended to allow structures to persist beyond the activation of the program. Their use is protected by the fact that the structure accesses are already dynamically checked in S-algol. Thus we have achieved persistence and retained the protection mechanism. Of course, in implementing this we had to devise a method of storing and retrieving persistent data as well as a description of its type and a method of checking type equivalence when data is reused in different programs. This was not a trivial task.

The choice of which data items persist beyond the lifetime of a program was next. We argued, by preaching minimum change, that the system should decide automatically. Such decisions are already taken in a number of languages like S-algol when garbage collection is involved and we therefore felt that reachability, as in garbage collection, was a reasonable choice for identifying persistent objects. However a new origin for the transitive closure of references, under explicit user control, which differentiates persistent data and transient data is introduced. Thus when a transaction is committed we can identify a root object from which we can find all the persistent data in the program. This data we preserve for later use.

PS-algol

Given the constraint of minimal change to S-algol the simplest way to extend the facilities of the language is by adding standard functions and PS-algol is implemented as a number of functional extensions to S-algol.

In this way the language itself does not change to accommodate persistence. Thus the population of S-algol programmers could now use PS-algol with very little change to their programming style. In fact an S-algol program will run correctly with little diminution of speed in the PS-algol run time environment.

The functions added to support persistence are

procedure open.database(**string** database.name,password,mode -> **pntr**)

This procedure attempts to open the database specified by database.name (which in general is a path down a tree of directories, terminating in the database's name) in the mode (read or write) specified by mode, quoting password to establish this right. If the open fails the result is a pointer to an error record. If it succeeds the result is a pointer to a table (see below) which contains a set of name-value pairs and is the root from which preserved data is identified by transitive closure of reachability. A table is chosen so as to permit programs which use one route to the data to be independent of programs using other routes. Many databases may be open for reading, but only one may be open for writing so that the destination for new persistent data may be deduced. If this is the first successful open.database then a transaction is started.

procedure commit

This procedure commits the changes made so far to the databases open for writing. The program may continue to make further changes. Only changes made before the last commit will be recorded, so that not preforming a commit is equivalent to aborting the transaction.

procedure close.database(**string** database.name)

This procedure closes the specified database making it available for other users who wish to write to it (multiple readers are allowed). It takes effect immediately. An implicit close database is applied to every database open at the end of the program. When a database has been closed references to objects not yet imported may remain accessible as may any imported data. An error is detected if the programmer tries to access any data that has not been imported from the closed database. A program may reopen a database it has closed but to avoid inconsistency between data that was held in the program and the database, it may only do so if there has not been an intervening write to that database.

These three procedures are those concerned with managing persistence, but there are also a set of procedures to manipulate tables, a mechanism for associative lookup, implemented as B-trees[8] or equivalent algorithms. A table is an ordered set of pairs. Each pair consists of a key and a value. At present the key may be an integer or string value, and the value is a pointer to a structure instance or table. The basic library for tables is:

procedure table(-> **pntr**)

This procedure creates an empty table represented by an instance of the structure class Table.

procedure s.lookup(**string** key ; **pntr** table -> **pntr**)
procedure i.lookup(**int** key ; **pntr** table -> **pntr**)

These procedures take a given key and considering only the pairs within the table return the last value stored by a call of s.enter or i.enter for this key, or **nil** if there is no such pair.

procedure s.enter(**string** key ; **pntr** table,value)
procedure i.enter(**int** key ; **pntr** table,value)

These procedures store in the given table a pair or if the supplied value is **nil** delete the previously stored value for the given key.

```
procedure s.scan( pntr table,environment ;
                ( string,pntr,pntr -> bool )user  -> int  )
procedure i.scan( pntr table,environment ;
                 ( int,pntr,pntr -> bool )user -> int  )
```

These procedures provide iteration over tables. The function user is applied to every pair with a key of the appropriate type, in ascending order of integers or lexical order of strings, until either it yields **false** or the whole table has been scanned. The first parameter supplied to the function user is the key, the second is the corresponding value and the third parameter the value given as environment. The result of these scan procedures is the number of times user is applied. An example of their use is given in figure 3.

procedure cardinality(**pntr** table -> **int**)

This procedure returns the current number of entries in the given table.

The table facilities were envisaged as a way of packaging index constructions. For example singly and multiple indexed relations can readily be constructed using them. They have also proved very popular as a dynamic data structure constructor. We present as an example of their use three PS-algol programs to maintain a simple address list. The program in figure 1 adds an new person, that in figure 2 looks up the telephone number of a person and that in figure 3 finds the longest telephone number.

```
structure person( string name,phone.no ; pntr addr,other )
structure address( int no ; string street,town ; pntr next.addr )
let db = open.database( "Address.list","Morwenna","write" )
if db is error.record then write "Can't open database" else
begin
     write "Name : "               ; let this.name  = read.a.line
     write "Phone number : "       ; let this.phone = read.a.line
     write "House number : "       ; let this.house = readi
     write "Street : "             ; let this.street= read.a.line
     write "Town : "               ; let this.town  = read.a.line
     let p = person( this.name,this.phone,address( this.house,this.street,
                     this.town,nil ),nil )
     let addr.list = s.lookup( "addr.list.by.name",db )
     s.enter( this.name,addr.list,p )
     commit
end
```

A program to add one new person to a database containing an address list

figure 1

```
structure person( string name,phone.no ; pntr addr,other )
let db = open.database( "Address.list","Kernow","read" )
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup( "addr.list.by.name",db )
write "Name : "                ; let this.name = read.a.line
let this.person = s.lookup( this.name,addr.list )
if this.person = nil then write "Person not known" else
   write "Phone number is : ",this.person( phone.no )
```

A program to look up the telephone number of one person from the address list

figure 2

```
structure person( string name,phone.no ; pntr addr,other )
let db = open.database( "Address.list", "Kernow","read" )
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup( "addr.list.by.name",db )
structure env( int max ; string longest )
let el = env( 0,"" )
procedure phone( string name ; pntr val,el -> bool )
begin
     let number = val( phone.no )
     let len = length( number )
     if len > el( max ) do { el( max ) := len ; el( longest ) := number }
     true
end
let count = s.scan( addr.list,el,phone )
if count = 0 then write "Nobody in the list yet" else
```

```
write "Longest telephone number is : ",el( longest )
```

A program to find the longest telephone number in the address list

figure 3

Note that in these examples the declared data structures are different. This identifies which parts of the data each program may touch and so behaves like a simple database view mechanism. Only those quoted will be matched against the stored set of data descriptions when they are first used. Paths to other data, eg alternative addresses via next.addr and extra information about people via other have not been used. They are there to give access to data pertinent to other programs or to future needs and they do not clutter programs which do not use them. Similarly the standard table is used to name access paths and only those used need be considered in a particular program. Figure 4 shows a program to introduce a new access path to the address list. With these disciplines, which we find useful, data design and programs can grow together. These examples illustrate the ease with which small programs operating on persistent data may be composed to provide a complete system. In this example one would imagine other programs: to start an address list, to delete it, to remove entries, amend addresses, print lists etc. We believe this method of constructing software tools to be an attractive way of building such systems.

As well as showing these features of persistence the programs illustrate features of the parent language S-algol. For example, the type of a name is deduced from the initialising expression, this leads to conciseness. The freedom to place declarations where they are needed means that constant names are common and programs may be read without searching far for declarations.

```
structure person( string name,phone.no ; pntr addr,other )
let db = open.database( "Address.list","Morwenna","write" )
if db is error.record do { write "Can't open database" ; abort  }
let addr.list = s.lookup( "addr.list.by.name",db )
let phone.number.table = table      !create a new empty table
procedure put.it.in.phone.number.table( string name;pntr val,env -> bool )
   { s.enter( val( phone.no ),phone.number.table,val ) ; true }
let count = s.scan( addr.list,nil,put.it.in.phone.number.table )
s.enter( "addr.list.by.phone.number",db,phone.number.table )
commit
```

A program to construct a new index onto the address list,by phone number

figure 4

The persistent object management system

The persistent object management system is concerned with the movement of data between main store and backing store. This is controlled by the run time support system where transactions are implemented by Challis' algorithm [10]. The movement of data is achieved as follows.

a. **Movement of data on to the heap**

Data may be created on the heap during a transaction or it may migrate there as a copy of some persistent data object. The second mechanism is invoked when a pointer being dereferenced is a persistent identifier (PID). The persistent object manager is called to locate the object and place it on the heap, possibly carrying out minor translations. The initial pointer which is a PID is that yielded by open.database and subsequent pointers will be found in the fields of structures reached from that reference.

b. **Movement from the heap**

When a transaction is committed, all the data on the heap that is reachable from the persistent objects used during the transaction are transferred back to the disk. Some anticipation of these transfers may be necessary if the heap space is insufficient for the whole transaction.

The algorithms and data structures used to implement this data movement are described elsewhere [3,4]. Since the database may be shared by many programs the binding of the names of persistent data must by dynamic and symbolic. In PS-algol this binding is performed automatically when the object is accessed for the first time. There is no overhead in accessing local objects.

Type checking is also performed by the system. Remember that a pointer may roam over the domain of structure classes. Thus when a pointer is dereferenced to yield a value the system must check that the pointer points to a structure with the correct field name. The field name and the structure incarnation must carry around type information to enable this checking to be performed.

To extend this type checking to persistent structures it is sufficient to ensure that the type information migrates with the structure itself. This is accomplished by making the type information an implicit field of the structure thus guaranteeing that the type information will persist.

Design issues in persistent object management

There are a number of tradeoffs to consider when designing the persistent object manager used to implement the data migration. Some of these are illustrated here. We may choose between making all references suitable as disc references (as in Smalltalk [16]), translating every time or we may (and usually do) economise by storing a mapping and translating less often. Then a choice exists as to when to build up the map. Do we make an entry when a reference is first introduced or when it is first dereferenced? Similarly do we make all references go via this map? There are problems of how to store the map so that it grows on demand rather than causing a high initial overhead. We require access to the map to be fast in both directions. These decisions interact with store management. We wish to avoid putting data on disc if we can determine it is not reachable. We may use these transfer mechanisms to avoid space exhaustion after garbage collection. When virtual memory [31] is available it interacts with these algorithms and may be exploited if the operating system permits. On the other hand we would like a run time system which is portable.

The design of the address structure for the persistent data allows interaction between addressability and the cost of disc garbage collection. This influences attempts to make the address structure extensible, to make it space efficient and to make it address sufficient data. Placement strategies, data compression techniquee and variations on the method of implementing transactions again interact and have significant effect on performance. The choice of a particular implementation is therefore a choice of a particular point in an extensive and many dimensional space. Our present choice described below is empirical and pragmatic rather than being based on systematic evaluation of this space or the optimisation of some abstract model.

Thus venturing into this approach to providing persistence has provoked many research issues concerning implementation which we have only just begun to explore and which will be the topics of other papers, the first of which is in preparation [5].

An implementation of persistence

Our earliest implementation of persistence used a linear table of PIDS and local addresses, with entries for every pointer in imported objects (objects that had been brought in from the database). Hashing was used to accelerate lookups of a PID (needed when a object is being imported to translate pointers to objects already imported). In the next implementation the position in this table was referred to as the local object number LON, and all references in all objects were represented by their LON. This cost an indirection in every reference (using an addressing mode on the VAX11) but meant that the local address to PID translation, needed on transferring data to the database (export) was merely indexing the table. Hash coding accelerated PID to LON translation and Bloom filters [9] were used to reduce the cost of discovering a PID had not yet been encountered. The main disadvantage of this was that an entry in the PIDLAM was needed for every active object so this table grew large. That implementation achieved good performance by using the virtual memory mechanisms of VAX, implementing Challis' algorithm as page tables stored at the start of the file and arranging that altered pages were paged to a new site.

Our present implementation, POMS (persistent object management system), is designed to overcome many of the deficiencies of the earlier systems, it is described in detail elsewhere [7]. Briefly, the design depends on adaptive structures for all the addressing mechanisms, to avoid the problems of a high fixed overhead, or a small limit to the maximum volume of data that can be accommodated. The PIDLAM is now two hashing structures which grow as necessary, one from PIDs to local addresses, the other from local addresses to PIDs, for only the pointers which have been both imported and dereferenced. Translation is done when a reference is first used to minimise the size of these tables.

To avoid a large number of transfers on startup, the logical to physical disc address mapping is itself mapped, using its own mapping and a bootstrap. So this map may grow as the database grows and is loaded incrementally. To avoid dependence on particular operating system features and the problem found when implementing transactions that file operations and database operations can get out of step, we include our own directory mechanism for databases. Depending on this strategy and writing most of the code for POMS in PS-algol itself, we believe we have achieved reasonably portability.

The objects are clustered by type in the database pages, so that the type description information may be factored out. The order of transfers during import is not under POMs control since it depends on the order the

program accesses structures. During the routines for commit, when data is exported, the total set of data which cannot become reachable from the root is identified from local information and discarded. The imported objects that have been changed and the newly created objets reachable from them, then have to be exported. This list of exports is sorted to minimise transfers and head movement, before write-back. The combination of these tactics, which we continue to refine, gives reasonable performance. Better performance may be achievable by exploiting segmentation hardware within the operating system.

Experience using PS-algol

At first sight the set of facilities provided by PS-algol may look fairly primitive. Notice however that the programmer never explicitly organises data movement but that it occurs automatically when data is used. Notice also that the language type rules are strictly enforced and that the programmer uses a method already familiar to him to preserve data. That is by the usual naming convention where the preservation of data is a consequence of arranging that there is a way of using the data. Thus we have achieved persistence by minimal change allowing the programmer to use all his familiar techniques of problem solving.

The effect on programs written in PS-algol has been quite dramatic. We have some early results of tests comparing programs written in PS-algol with programs written in Pascal with explicit database calls. These programs implemented a DAPLEX [27] look-alike, a relational algebra and various CAD and demonstration programs. We have found that there is a reduction by a factor of about three in the length of the source code. These are of course early results and will need further confirmation. However it is the sort of result we expected.

We have also found that the coding time for these programs is reduced by at least the same factor. We suspect that the maintenance of the programs will be easier. We avoid the layering costs associated with calls to successive levels of a DBMS as data is brought into the normal program from the heap. Consequently we have observed a reduction in CPU requirements for equivalent programs even though we are still interpreting. Whether there is an overall increase in speed depends on the data structures and the algorithms over them and we hope to investigate performance further.

Evaluation of persistence as an abstraction

The abstraction of persistence has been so successful that we would recommend that other language designers consider it for the languages they design. It identifies significant aspects of common programming tasks consequently reducing the effort required by the programmer to accomplish those tasks. It abstracts away much detail commonly visible to the programmer so that programs produced using it are simpler to understand and to transport. It is feasible to implement.

Persistence has also appeared as an orthogonal property of data in the work of Albano et al[1]. In an attempt to accommodate longer term persistence they make contexts proper objects which can be manipulated, as we suggest for an identified subset of contexts in our language proposals NEPAL[2]. In another group of languages PASCAL/R[26], RIGEL[25] & PLAIN [32] the designers have chosen not to adopt the principle of data type completeness, and have only allowed instances of type relation to have longer term persistence. It is interesting to note that in PASCAL/R the construct DATABASE has a form like a PASCAL record, and if its fields were allowed to take any of the PASCAL data types then that language would be consistent in allowing data structures of any type to have any persistence. ADAPLEX[28] is constructed by merging a given database model, DAPLEX[27] and an existing language ADA[15], with the inevitable consequence of restrictions on which data types can have which persistence. When casting program examples to assess languages [6] we have found it particularly irksome if data types which can persist cannot also have temporary instances. We conclude, therefore, that the consistency of our form of persistence is of advantage to the programmer.

Conclusions

The abstraction achieved by treating persistence as an orthogonal property of data has been shown to have many interesting properties. It is clear it favourably affects program length, program development time and program maintainability. Demonstration implementations of a particular flavour of this idea have shown it to be practicable for reasonable amounts of data. This particular flavour does not have concurrency other than at database level. This may be a limitation which prevents its application to simple transactions on large database systems but this leaves the very substantial number of programs which run against conventional files which implement complex interactions as in CAD, or use personal databases as uses of this language.

Acknowledgements

The work at Edinburgh was supported in part by SERC grant GRA 86541. It is now supported at Edinburgh by SERC grants GRC 21977 and GRC 21960 and at St Andrews by SERC grant GRC 15907. The work is also supported at both Universities by grants from ICL.

References

1. Albano, A., Cardelli, L. & Orsini, R. (1982). Galileo : a strongly typed interactive conceptual language.
2. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. (1982). NEPAL - The New Edinburgh Persistent Algorithmic Language. DATABASE Infotech State of the Art Report 9,8 299-318.
3. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. (1983). CMS : A chunk management system. accepted for publication Software, Practice & Experience
4. Atkinson, M.P., Chisholm, K.J., Cockshott, W.P. & Marshall, R.M. (1983). Algorithms for a persistent heap. accepted for publication Software, Practice & Experience
5. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. (1983). An exploration of various strategies for implementing persistence as an orthogonal property of data. in preparation.
6. Atkinson, M.P. & Buneman, P. (1983). Survey paper on persistent languages. in preparation.
7. Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. (1983). The persistent object management system. in preparation.
8. Bayer, B. & McCreight, A. (1972). Organisation and maintenance of large ordered indexes. Acta Informatica 1 173-189.
9. Bloom, B.H. (1970). Space/time tradeoffs in hash coding with allowable errors. Comm.ACM 13,7 422-426.
10. Challis, M.P. (1978). Data Consistency and integrity in a multi-user environment. In Databases : improving usability and responsiveness Academic Press 245-270.
11. Codd, E.F. (1970). A relational model for large shared databases. Comm.ACM 13,6 377-387.
12. Cole, A.J. & Morrison, R. (1982). An introduction to programming with

S-algol. Cambridge University Press.

13. Hoare, C.A.R. (1974). Monitors : an operating system structuring concept. Comm.ACM 17,10 549-557.
14. IBM report (1978). on the contents of a sample of programs surveyed. San Jose, California
15. Ichbiah et al (1979). Rationale of the design of the programming language Ada. ACM Sigplan Notices 14,6.
16. Kaehler, T. (1982). Virtual memory for an object-orientated language. Byte 378-387.
17. Landin, P.J. (1966). The next 700 programming languages. Comm.ACM 9,3 157-164.
18. Liskov, B.H. et al (1977). Abstraction mechanisms in CLU. Comm.ACM 20,8 564-576.
19. Lochovsky, F.H. & Tsichritizis, D.C. (1978). Hierarchical database management systems. ACM Computing Surveys 8,1 105-123.
20. Lochovsky, F.H. & Tsichritizis, D.C. (1982). Data Models.
21. Morrison, R. (1979). S-algol language reference manual. University of St Andrews CS/79/1.
22. Morrison R. (1982). Towards simpler programming languages : S-algol IUCC Bulletin 4,3 (October 1982)
23. Morrison, R. (1982). Low cost computer graphics for micro computers. Software, Practice & Experience 12 767-776.
24. Morrison, R. (1982). The string as a simple data type. ACM.Sigplan Notices Vol 17,3.
25. Rowe L.A. Reference manual for the programming language RIGEL.
26. Schmidt, J.W. (1977). Some high level language constructs for data of type relation . ACM.TODS 2,3 247-261.
27. Shipman, D.W. (1981). The functional data model and the data language DAPLEX. ACM.TODS 6,1 140-173.
28. Smith, J.M., Fox, S. & Landers, T. (1981). Reference manual for ADAPLEX. Computer Corporation of America, Cambridge, Massachusetts
29. Strachey, C. (1967). Fundamental concepts in programming languages. Oxford University Press.
30. Taylor, R.C & Frank, R.L. (1976). CODASYL database management systems. ACM Computing Surveys 8,1 67-103.
31. Traiger, I.L. (1982). Virtual memory management for database systems. ACM.Sogops 16,4 26-48.

32. Wasserman, A.I., Sheretz, D.D., Kersten,M.L., van de Reit, R.D. (1981). Revised report on the programming language PLAIN. ACM.Sigplan Notices 16,5.
33. van Wijngaarden, A. (1963). Generalised algol. Annual Review of automatic programming 3 17-26.
34. van Wijngaarden, A. et al (1969). Report on the algorithmic language Algol 68. Numerische Mathematik 14 79-218.
35. Wirth, N. & Hoare, C.A.R. (1966). A contribution to the development of algol. Comm.ACM 9,6 413-431.
36. Wirth, N. (1971). The programming language Pascal. Acta Informatica 1 35-63.

PART 2

An approach to persistent programming

Presented at the Digital WOrkshop on Databases and Programming Languages, Boston, 1983.

An approach to persistent programming

M.P.Atkinson[+], P.J.Bailey[*], K.J.Chisholm[+], W.P.Cockshott[+] and R.Morrison[*]

[+] Department of Computer Science, University of Edinburgh, Mayfield Rd., Edinburgh EH9 3JZ

[*] Department of Computational Science, University of St.Andrews North Haugh, St.Andrews KY16 8SX

Abstract

This paper presents the identification of a new programming language concept and reports our initial investigations of its utility. The concept is to identify persistence as an orthogonal property of data, independent of data type and the way in which data is manipulated. This is expressed by the principle that all data objects independent of their data type should have the same rights to persistence or transience. We expect to achieve persistent independent programming, so that the same code is applicable to data of any persistence. We have designed a language PS-algol by utilising these ideas and constructed a number of implementations. The experience gained is reported here, as a step in the task of achieving languages with proper accommodation for persistent programming.

Introduction

The long term storage of data has been of concern to programming language designers for some time. Traditional programming languages provide facilities for the manipulation of data whose lifetime does not extend beyond the activation of the program. If data is required to survive a program activation then some file I/O or database management system interface is used. Two views of data evolve from this. Data can either be classed as short term data and would be manipulated by the programming language facilities or data would be long term data in which case it would be manipulated by the file system or the database management system (DBMS). The mapping between the two types of data is usually done in part by the file system or the DBMS and in part by explicit user translation code which has to be written and included in each program.

These different views of data are highlighted when the data structuring facilities of programming languages and database management systems are compared. Database systems have developed relational, hierarchical, network and functional data models [39,1,2,3,4] whereas programming languages may have arrays, records, sets, monitors [5] and abstract data types [6].

That there are two different views of data has certain disadvantages. Firstly in any program there is usually a considerable amount of code, typically 30% of the total [37], concerned with transferring data to and from files or a DBMS. Much space and time is taken up by code to perform translations between the program's form of data and the form used for the long term storage medium. This is unsatisfactory because of the time taken in writing and executing this mapping code and also because the quality of the application programs may be impaired by the mapping. Frequently the programmer is distracted from his task by the difficulties of understanding and managing the mapping. The translation merely to gain access to long term data should be differentiated from translations from a form appropriate to one use of the data to a form suitable for some other algorithms. Such translations are justified when the two forms cannot coexist and there is a substantial use of both forms. The second major disadvantage is that the data type protection offered by the programming language on its data is often lost across the mapping. The structure that might have been exploited in a program to aid comprehension is neither apparent nor protected and is soon lost.

We seek to eliminate the differences between the DBMS and programming language models of data. This can be done by separating the issue of what data structures are best from the issue of identifying and managing a property of data we call persistence. This is the period of time for which the data exists and is useable. A spectrum of persistence exists and is categorised by

1. transient results in expression evaluation.
2. local variables in procedure activations.
3. own variables, global variables and heap items whose extent is different from their scope.
4. data that exists between executions of a program.
5. data that exists between various versions of a program.
6. data that outlives the program.

The first three persistence categories are usually supported by programming languages and the second three categories by a DBMS, whereas filing systems are predominantly used for categories 4 and 5. We report here on the PS-algol system which is a step in the search for language and database constructs which meet the needs of persistent data and hence obviate the need for the programmer to cope with the problems described above. PS-algol uniformly supports programming for an increased range of persistence. The system is implemented and is being actively used in a number of projects. Here we will describe the language design decisions in implementing persistence, the underlying implementation problems, the results obtained and some thoughts for the future.

The language design method

As a first attempt at producing a system to support persistence we hypothesised that it should be possible to add persistence to an existing language with minimal change to the language. Thus the programmer would be faced with the normal task of mastering the programming language but would have the facility of persistence for little or no extra effort.

The language chosen for this was S-algol [7,8], a high level algol used for teaching at the University of St Andrews. This decision was made by the University of Edinburgh team after some trouble with attempts at Algol 68 [9] and Pascal [10] and resulted in the two teams collaborating on the project.

S-algol stands somewhere between Algol W [11] and Algol 68. It was designed using three principles first outlined by Strachey [12] and

Landin[13]. These are

1. The principle of correspondence
2. The principle of abstraction
3. The principle of data type completeness

The application of the three principles in designing S-algol is described elsewhere [14]. The result is an orthogonal language whose "power is gained from simplicity and its simplicity from generality" [15]. Here we are interested in data and the S-algol universe of discourse can be defined by

1. The scalar data types are integer, real, boolean, string, picture and file.
2. For any data type T, *T is the data type of a vector with elements of type T.
3. The data type pointer comprises a structure with any number of fields, and any data type in each field.

The world of data objects can be formed by the closure of rule a under the recursive application of rules b and c.

The unusual features of the S-algol universe of discourse are that it has strings as a simple data type [38], pictures as compound data objects [34] and run time checking of structure classes. The picture facility allows the user to produce line drawings in an infinite two dimensional space. It also offers a picture building facility in which the relationship between different sub-pictures is specified by mathematical transformations. A basic set of picture manipulating facilities along with a set of physical drawing attributes for each device is defined. A pointer may roam freely over the world of structures. That is a pointer is not bound to a structure class. However when a pointer is dereferenced using a structure field name, a run time check occurs to ensure the pointer is pointing at a structure with the appropriate field.

Together with our hypothesis of minimal change to the programming language we recognise certain principles for persistent data.

1. persistence independence : the persistence of a data object is independent of how the program manipulates that data object and conversely a fragment of program is expressed independently of the persistence of data it manipulates. For example, it should be possible to call a procedure with its parameters sometimes objects with long term persistence and at other times only transient.

2. persistence data type orthogonality : In line with the principle of data type completeness all data objects should be allowed the full range of persistence.
3. The choice of how to provide and identify persistence at a language level is independent of the choice of data objects in the language.

A number of methods were investigated to identify persistence of data. Some involved associating persistence with the variable name or the type in the declaration. Under the rule of persistence independence these were disallowed. S-algol itself helped to provide the solution. Its data structures already have some limited notion of persistence in that the scope and extent of these objects need not be the same. Such structures are of course heap items and their useability depends on the availability of legal names.

This limited persistence was extended to allow structures to persist beyond the activation of the program. Their use is protected by the fact that the structure accesses are already dynamically checked in S-algol. Thus we have achieved persistence and retained the protection mechanism. Of course, in implementing this we had to devise a method of storing and retrieving persistent data as well as a description of its type and a method of checking type equivalence when data is reused in different programs. This was not a trivial task.

The choice of which data items persist beyond the lifetime of a program was next. We argued, by preaching minimum change, that the system should decide automatically. Such decisions are already taken in a number of languages like S-algol when garbage collection is involved and we therefore felt that reachability, as in garbage collection, was a reasonable choice for identifying persistent objects. However a new origin for the transitive closure of references, under explicit user control, which differentiates persistent data and transient data is introduced. Thus when a transaction is committed we can identify a root object from which we can find all the persistent data in the program. This data we preserve for later use.

PS-algol

Given the constraint of minimal change to S-algol the simplest way to extend the facilities of the language is by adding standard functions and PS-algol is implemented as a number of functional extensions to S-algol.

In this way the language itself does not change to accommodate persistence. Thus the population of S-algol programmers could now use PS-algol with very little change to their programming style. In fact an S-algol program will run correctly with little diminution of speed in the PS-algol run time environment.

The functions added to support persistence are

procedure open.database(**string** database.name,password,mode -> **pntr**)

This procedure attempts to open the database specified by database.name (which in general is a path down a tree of directories, terminating in the database's name) in the mode (read or write) specified by mode, quoting password to establish this right. If the open fails the result is a pointer to an error record. If it succeeds the result is a pointer to a table (see below) which contains a set of name-value pairs of roots from which preserved data is identified by transitive closure of reachability. A table is chosen so as to permit programs which use one route to the data to be independent of programs using other routes. Many databases may be open for reading, but only one may be open for writing so that the destination for new persistent data may be deduced. If this is the first successful open.database then a transaction is started.

procedure commit

This procedure commits the changes made so far to the databases open for writing. The program may continue to make further changes. Only changes made before the last commit will be recorded, so that not preforming a commit is equivalent to aborting the transaction.

procedure close.database(**string** database.name)

This procedure closes the specified database making it available for other users who wish to write to it (multiple readers are allowed). It takes effect immediately. An implicit close database is applied to every database open at the end of the program. When a database has been closed references to objects not yet imported may remain accessible as may any imported data. An error is detected if the programmer tries to access any data that has not been imported from the closed database. A program may reopen a database it has closed but to avoid inconsistency between data that was held in the program and the database, it may only do so if there has not been an intervening write to that database.

These three procedures are those concerned with managing persistence, but there are also a set of procedures to manipulate tables, a mechanism for associative lookup, implemented as B-trees[36] or equivalent algorithms. A table is an ordered set of pairs. Each pair consists of a key and a value. At present the key may be an integer or string value, and the value is a pointer to a structure instance or table. The basic library for tables is:

procedure table(-> **pntr**)

This procedure creates an empty table represented by an instance of the structure class Table.

procedure s.lookup(**string** key ; **pntr** table -> **pntr**)
procedure i.lookup(**int** key ; **pntr** table -> **pntr**)

These procedures take a given key and considering only the pairs within the table return the last value stored by a call of s.enter or i.enter for this key, or **nil** if there is no such pair.

procedure s.enter(**string** key ; **pntr** table,value)
procedure i.enter(**int** key ; **pntr** table,value)

These procedures store in the given table a pair or if the supplied value is **nil** delete the previously stored value for the given key.

procedure s.scan(**pntr** table,environment ;
(**string,pntr,pntr** -> **bool**)user -> **int**)
procedure i.scan(**pntr** table,environment ;
(**int,pntr,pntr** -> **bool**)user -> **int**)

These procedures provide iteration over tables. The function user is applied to every pair with a key of the appropriate type, in ascending order of integers or lexical order of strings, until either it yields false or the whole table has been scanned. The first parameter supplied to the function user is the key, the second is the corresponding value and the third parameter the value given as environment. The result of these scan procedures is the number of times user is applied. An example of their use is given in figure 3.

procedure cardinality(**pntr** table -> **int**)

This procedure returns the current number of entries in the given table.

The table facilities were envisaged as a way of packaging index constructions. For example singly and multiple indexed relations can readily be constructed using them. They have also proved very popular as a dynamic data structure constructor. We present as an example of PS-algol programs using them, three programs to maintain a simple address list. The program in figure 1 adds an new person, that in figure 2 looks up the telephone number of a person and that in figure 3 finds the longest telephone number.

```
structure person( string name,phone.no ; pntr addr,other )
structure address( int no ; string street,town ; pntr next.addr )
let db = open.database( "Address.list","Morwenna","write" )
if db is error.record then write "Can't open database" else
begin
     write "Name : "               ; let this.name  = read.a.line
     write "Phone number : "       ; let this.phone = read.a.line
     write "House number : "       ; let this.house = readi
     write "Street : "             ; let this.street= read.a.line
     write "Town : "               ; let this.town  = read.a.line
     let p = person( this.name,this.phone,address( this.house,this.street,
                     this.town,nil ),nil )
     let addr.list = s.lookup( "addr.list.by.name",db )
     s.enter( this.name,addr.list,p )
     commit
end
```

A program to add one new person to a database containing an address list

figure 1

```
structure person( string name,phone.no ; pntr addr,other )
let db = open.database( "Address.list","Kernow","read" )
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup( "addr.list.by.name",db )
write "Name : "               ; let this.name = read.a.line
let this.person = s.lookup( this.name,addr.list )
if this.person = nil then write "Person not known" else
   write "Phone number is : ",this.person( phone.no )
```

A program to look up the telephone number of one person from the address list

figure 2

```
structure person( string name,phone.no ; pntr addr,other )
let db = open.database( "Address.list", "Kernow","read" )
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup( "addr.list.by.name",db )
structure env( int max ; string longest )
let el = env( 0,"" )
procedure phone( string name ; pntr val,el -> bool )
begin
     let number = val( phone.no )
     let len = length( number )
     if len > el( max ) do { el( max ) := len ; el( longest ) := number }
     true
end
let count = s.scan( addr.list,el,phone )
if count = 0 then write "Nobody in the list yet" else
```

```
   write "Longest telephone number is : ",el( longest )
```

A program to find the longest telephone number in the address list

figure 3

Note that in these examples the declared data structures are different. This identifies which parts of the data each program may touch and so behaves like a simple database view mechanism. Only those quoted will be matched against the stored set of data descriptions when they are first used. Paths to other data, eg alternative addresses via next.addr and extra information about people via other has not been used. It is there to give access to data pertinent to other programs or to future needs and does not clutter programs which do not use it. Similarly the standard table is used to name access paths and only those used need be considered in a particular program. Figure 4 shows a program to introduce a new access path to the address list. With these disciplines, which we find useful, data design and programs can grow together. These examples illustrate the ease with which small programs operating on persistent data may be composed to provide a complete system. In this example one would imagine other programs: to start an address list, to delete it, to remove entries, amend addresses, print lists etc. We believe this method of constructing software tools to be an attractive way of building such systems.

As well as showing these features of persistence the programs illustrate features of the parent language S-algol. For example, the type of a name is deduced from the initialising expression, this leads to conciseness. The freedom to place declarations where they are needed means that constant names are common and programs may be read without searching far for declarations.

```
structure person( string name,phone.no ; pntr addr,other )
let db = open.database( "Address.list","Morwenna","write" )
if db is error.record do { write "Can't open database" ; abort  }
let addr.list = s.lookup( "addr.list.by.name",db )
let phone.number.table = table      !create a new empty table
procedure put.it.in.phone.number.table( string name;pntr val,env -> bool )
   { s.enter( val( phone.no ),phone.number.table,val ) ; true }
let count = s.scan( addr.list,nil,put.it.in.phone.number.table )
s.enter( "addr.list.by.phone.number",db,phone.number.table )
commit
```

A program to construct a new index onto the address list,by phone number

figure 4

The persistent object management system

The main interest here is in the control of the movement of data between main store and backing store. This is controlled by the run time support system. Transactions are implemented by Challis' algorithm [16]. The movement of data is achieved as follows.

a. **Movement of data on to the heap**

Data may be created on the heap during a transaction or it may migrate there as a copy of some persistent data object. The second mechanism is invoked when a pointer being dereferenced is a **persistent identifier (PID)**. The persistent object manager is called to locate the object and place it on the heap, possibly carrying out minor translations. The initial pointer which is a **PID** is that yielded by open.database and subsequent pointers will be found in the fields of structures reached from that reference.

b. **Movement from the heap**

When a transaction is committed, all the data on the heap that is reachable from the persistent objects used during the transaction are transferred back to the disk. Some anticipation of these transfers may be necessary if the heap space is insufficient for the whole transaction.

The algorithms and data structures used to implement this data movement are described elsewhere [17,18]. Since the database may be shared by many programs the binding of the names of persistent data must by dynamic and symbolic. In PS-algol this binding is performed automatically when the object is accessed for the first time. There is no overhead in accessing local objects.

Type checking is also performed by the system. Remember that a pointer may roam over the domain of structure classes. Thus when a pointer is dereferenced to yield a value the system must check that the pointer points to a structure with the correct field name. The field name and the structure incarnation must carry around type information to enable this checking to be performed.

To extend this type checking to persistent structures it is sufficient to ensure that the type information migrates with the structure itself. This is accomplished by making the type information an implicit field of the structure thus guaranteeing that the type information will persist.

Design issues in persistent object management

There are a number of tradeoffs to consider when designing the persistent object manager used to implement the data migration. Some of these are illustrated here. We may choose between making all references suitable as disc references (as in Smalltalk [40]), translating every time or we may (and usually do) economise by storing a mapping and translating less often. Then a choice exists as to when to build up the map. Do we make an entry when a reference is first introduced or when it is first dereferenced? Similarly do we make all references go via this map? There are problems of how to store the map so that it grows on demand rather than causing a high initial overhead. We require access to the map to be fast in both directions. These decisions interact with store management. We wish to avoid putting data on disc if we can determine it is not reachable. We may use these transfer mechanisms to avoid space exhaustion after garbage collection. When virtual memory [41] is available it interacts with these algorithms and may be exploited if the operating system permits. On the other hand we would like a programming system which is portable.

The design of the address structure for the persistent data allows interaction between addressability and the cost of disc garbage collection. This influences attempts to make the address structure extensible, to make it space efficient and to make it address sufficient data. Placement strategies, data compression techniques and variations on the method of implementing transactions again interact and have significant effect on performance. The choice of a particular implementation is therefore a choice of a particular point in an extensive and many dimensional space. Our present choice described below is empirical and pragmatic rather than being based on systematic evaluation of this space or the optimisation of some abstract model.

Thus venturing into this approach to providing persistence has provoked many research issues concerning implementation which we have only just begun to explore and which will be the topics of other papers, the first of which is in preparation [19].

An implementation of persistence

Our earliest implementation of persistence used a linear table of PIDS and local addresses, with entries for every pointer in imported objects (objects that had been brought in from the database). Hashing was used to accelerate lookups of a PID (needed when a object is being imported to translate pointers to objects already imported). In the next implementation the position in this table was referred to as the local object number LON, and all references in all objects were represented by their LON. This cost an indirection in every reference (using an addressing mode on the VAX11) but meant that the local address to PID translation, needed on transferring data to the database (export) was merely indexing the table. Hash coding accelerated PID to LON translation and Bloom fitters [44] were used to reduce the cost of discovering a PID had not yet been encountered. The main disadvantage of this was that an entry in the PIDLAM was needed for every object so this table grew large. That implementation achieved good performance by using the virtual memory mechanisms of VAX, implementing Challis' algorithm as page tables stored at the start of the file and arranging that altered pages were paged to a new site.

Our present implementation, POMS (persistent object management system), is designed to overcome many of the deficiencies of the earlier systems, it is described in detail elsewhere [40]. Briefly, the design depends on adaptive structures for all the addressing mechanisms, to void the problems of a high fixed overhead, or a small limit to the maximum volume of data that can be accommodated. The PIDLAM is now two hashing structures which grow as necessary, one from PIDs to local addresses, the other from local addresses to PIDs, for only the pointer which have been both imported and dereferenced. Translation is done when a reference is first used to minimise the size of these tables.

To avoid a large number of transfers on startup, the logical to physical disc address mapping necessary for Challis' algorithm is itself mapped, using its own mapping and a bootstrap. So this map may grow as the database grows and is loaded incrementally. To avoid dependence on particular operating system features and the problem found when implementing transactions that file operations and database operations can get out of step, we include our own directory mechanism for databases. Depending on this strategy and writing most of the code for POMS in PS-algol itself, we believe we have achieved reasonably portability.

The objects are clustered by type in the database pages, so that the type description information may be factored out. The order of transfers during import is not under POMs control since it depends on the order the

program accesses structures. During the routines for commit, when data is exported, the total set of data which cannot become reachable from the root is identified from local information and discarded. The imported objects that have been changed and the newly created objets reachable from them, then have to be exported. This list of exports is sorted to minimise transfers and head movement, before write-back. The combination of these tactics, which we continue to refine, gives reasonable performance. Better performance may be achievable by exploiting segmentation hardware within the operating system.

Experience using PS-algol

At first sight the set of facilities provided by PS-algol may look fairly primitive. Notice however that the programmer never explicitly organises data movement but that it occurs automatically when data is used. Notice also that the language type rules are strictly enforced and that the programmer uses a method already familiar to him to preserve data. That is by the usual naming convention where the preservation of data is a consequence of arranging that there is a way of using the data. Thus we have achieved persistence by minimal change allowing the programmer to use all his familiar techniques of problem solving.

The effect on programs written in PS-algol has been quite dramatic. We have some early results of tests comparing programs written in PS-algol with programs written in Pascal with explicit database calls. These programs implemented a DAPLEX [4] look-alike, a relational algebra and various CAD and demonstration programs. We have found that there is a reduction by a factor of about three in the length of the source code. These are of course early results and will need further confirmation. However it is the sort of result we expected.

We have also found that the coding time for these programs is reduced by at least the same factor. We suspect that the maintenance of the programs will be easier since it is some function of the number of lines of code being maintained. We avoid the layering costs associated with calls to successive levels of a DBMS as data is brought into the normal program from the heap. Consequently we have observed a reduction in CPU requirements for equivalent programs even though we are still interpreting. Whether there is an overall increase in speed depends on the data structures and the algorithms over them and we hope to investigate performance further.

Evaluation of persistence as an abstraction

It is apparent from our experiments that it is useful to introduce a new abstraction into programming languages. Persistence is the abstraction of long term data storage and transient data storage into one simpler concept.

The abstraction has been so successful that we would recommend that other language designers include it in the languages they design. It is successful in three important respects: it identifies significant aspects of common programming tasks consequently reducing the effort required by the programmer to accomplish those tasks, it abstracts away much detail commonly visible to the programmer so that programs produced using it are simpler to understand and to transport and the abstraction leaves the language system implementor with a feasible task.

Although the abstraction itself is successful, we note that there are various flavours to be explored. These potential flavours lead us to consideration of new research, and to a comparison with other research. Persistence has appeared as an orthogonal property of data in one other language in the work of Albano et al[20]. In an attempt to accommodate longer term persistence they make contexts proper objects in the language which can be manipulated, as we suggest for an identified subset of contexts in our language proposals NEPAL[22]. In another group of languages PASCAL/R[23], RIGEL[24] & PLAIN [25] the designers have chosen not to adopt the principle of data type completeness, and have only allowed instances of type relation to have longer term persistence. It is interesting to note that in PASCAL/R the construct DATABASE has a form like a PASCAL record, and if its fields were allowed to take any of the PASCAL data types then that language would be consistent in allowing data structures of any type to have any persistence. But PASCAL/R does not attempt data type completeness in other situations. ADAPLEX[26] is another example of a language constructed by merging a given database model, DAPLEX[4] and an existing language ADA[27], with the inevitable consequence of restrictions on which data types can have which persistence. When casting program examples to assess languages [42] we have found it particularly irksome if data types which can persist cannot also have temporary instances, perhaps more often than we have found difficulty with being unable to construct directly a persistent equivalent of some temporary data structure. We conclude, therefore, that the consistency of our form of persistence is of advantage to the programmer. So far it has met our programming needs well. If it were not well suited to a particular category of applications, this would not invalidate the idea of incorporating an abstraction of persistence into a programming language, but would suggest a different flavour is needed. For example the base types and base constructors might need to be different.

Prospects for persistence

It is early yet to assess the impact of persistence as a language concept. Given an embodiment of it in a language it appears to have a profound effect on programming style. The first reaction is to notice that programs can be split into separate programs. Each builds or modifies a data structure which persists. Consequently, the nature of the data structures which are practicable is changed. For example, in an assembler system there are separate programs for setting up an empty table of machine function mnemonics, adding entries to that table, deleting entries, changing the syntax and semantics associated with an entry, listing the entries, interrogating an individual entry. All these are built independently of the assembler itself. Most of the programs are less than a page long and are easily built and maintained. The program development environment can exploit persistent data structures but needs new features to support them. As an example, we have an FQL [43] interface to allow high level queries against the persistent data for debugging and program planning.

We seek the ideal of a persistent interactive language which is both simple and yet supports appropriate data types. We particularly hope to allow high level bulk operations, such as those achieved by database query languages, to be one end of a spectrum of operations which without major discontinuity spans to the intimate manipulation of detailed structure which some tasks demand.

The consequences of introducing the persistence abstraction into language design are likely to influence other aspects of languages. As well as changing the base types and constructions, we expect to see changes in the ideas of constancy. At present one can assume that certain things are totally constant. For example in the declaration:

structure pie(**string** name ; **real** weight ; **bool** meat)

there is the idea that the extent of pie may vary, the values of a variable within an instance of pie may vary, but the template (types and component names) and the representation does not. Once we look for supporting an indefinite range of persistence then these aspects of structures must be seen as only relatively constant.

It is possible that the provision of persistence in languages will modify the way we build systems. Consider database systems. At present implementing these systems is a substantial software task. DBMS are difficult to build because it is not easy to impose structure on them, interfaces are many and complex. It is possible that a persistent language may be an appropriate decomposition. The interface and modelling software is easily written in it, without the distraction of integrity, recovery, data transfers, placement etc. If it is possible to support it with sufficient efficiency, which we believe it may be, then this is a promising approach.

Conclusions

The abstraction achieved by treating persistence as an orthogonal property of data has been shown to have many interesting properties. It is clear it favourably affects program length, program development time and program maintainability. Demonstration implementations of a particular flavour of this idea have shown it to be practicable for reasonable amounts of data. This particular flavour does not have concurrency other than at database level. This may be a limitation which prevents its application to simple transactions on large database systems but this leaves the very substantial number of programs which run against conventional files which implement complex interactions as in CAD, or use personal databases as uses of this language. It is expected that programming based on this abstraction will evolve and require consequential adaption of other language aspects. Certainly the idea has proved itself of sufficient interest to warrant pursing the large number of research possibilities it suggests.

Acknowledgements

The work at Edinburgh was supported in part by SERC grant GRA 86541. It is now supported at Edinburgh by SERC grants GRC 21977 and GRC 21960 and at St Andrews by SERC grant GRC 15907. The work is also supported at both Universities by a grant from ICL.

References

1. Codd, E.F. (1970). A relational model for large shared databases Comm.ACM 13,6 377-387.
2. Lochovsky, F.H. & Tsichritizis, D.C. (1978). Hierarchical database management systems ACM Computing Surveys 8,1 105-123.
3. Taylor, R.C & Frank, R.L. (1976). CODASYL database management systems ACM Computing Surveys 8,1 67-103.
4. Shipman, D.W. (1981). The functional data model and the data language DAPLEX ACM.TODS 6,1 140-173.
5. Hoare, C.A.R. (1974). Monitors : an operating system structuring concept Comm.ACM 17,10 549-557.
6. Liskov, B.H. et al (1977). Abstraction mechanisms in CLU. Comm.ACM 20,8 564-576.
7. Morrison, R. (1979). S-algol language reference manual. University of St Andrews CS/79/1.
8. Cole, A.J. & Morrison, R. (1982). An introduction to programming with S-algol. Cambridge University Press.
9. van Wijngaarden, A. et al (1969). Report on the algorithmic language Algol 68. Numerische Mathematik 14 79-218.
10. Wirth, N. (1971). The programming language Pascal. Acta Informatica 1 35-63.
11. Wirth, N. & Hoare, C.A.R. (1966). A contribution to the development of algol. Comm.ACM 9,6 413-431.
12. Strachey, C. (1967). Fundamental concepts in programming languages. Oxford University Press.
13. Landin, P.J. (1966). The next 700 programming languages. Comm.ACM 9,3 157-164.
14. Morrison R. (1979). Towards simpler programming languages : S-algol IUCC Bulletin 4,3 (October 1982)
15. van Wijngaarden, A. (1963). Generalised algol. Annual Review of automatic programming 3 17-26.
16. Challis, M.P. (1978). Data Consistency and integrity in a multi-user environment. In Databases : improving usability and responsiveness Academic Press 245-270.
17. Atkinson, M.P., Cockshott, W.P. & Chisholm, K.J. (1982). CMS : A chunk management system. accepted for publication Software, Practice & Experience

18. Atkinson, M.P., Cockshott, W.P., Chisholm, K.J. & Marshall, R.M. (1982). Algorithms for a persistent heap. accepted for publication Software, Practice & Experience

19. Atkinson, M.P., Cockshott, W.P. & Chisholm, K.J. (1982). An exploration of various strategies for implementing persistence as an orthogonal property of data. in preparation.

20. Albano, A., Cardelli, L. & Orsini, R. (1982). Galileo : a strongly typed interactive conceptual language.

21. Gordon, M.J.C., Milner, A.J.R.G., Morris, L., Newey, M. & Wadsworth, C. (1978). A metalanguage for interactive proof in LCF. ACM.POPL

22. Atkinson, M.P., Cockshott, W.P. & Chisholm, K.J. (1982). NEPAL - The New Edinburgh Persistent Algorithmic Language. DATABASE Infotech State of the Art Report 9,8 299-318.

23. Schmidt, J.W. (1977). Some high level language constructs for data of type relation . ACM.TODS 2,3 247-261.

24. Rowe L.A. Reference manual for the programming language RIGEL.

25. Wasserman, A.I., Sheretz, D.D., Kersten,M.L., van de Reit, R.D. (1981). Revised report on the programming language PLAIN. ACM SIGPLAN Notices 16,5.

26. Smith, J.M., Fox, S. & Landers, T. (1981). Reference manual for ADAPLEX. Computer Corporation of America, Cambridge, Massachusetts

27. Ichbiah et al (1979). Rationale of the design of the programming language Ada. ACM Sigplan Notices 14,6.

28. Cardelli, L. (1982). The semantics of a two dimensional language. University of Edinburgh, Department of Computer Science Report CSR-115-82.

29. Rowe, L. & Skeans Screen RIGEL. University of California at Berkeley, Department of Electrical Engineering.

30. Shu, N.C. & Lum, V.Y. (1982). Specification of Forms Processing and Business procedures for Office Automation. IEEE transactions on Software Engineering, SE-8,5 499-512.

31. Tesler, L. (1982). The Smalltalk environment. Byte 90-147.

32. Kung, H.T. & Robinson, J.T. (1982). On Optimistic Methods for Concurrency Control. ACM.TODS 6,2 213-226.

33. Date, C.J. (1979). Locking and recovery in a shared database system : An application tutorial. in proceedings of the 5th Very Large Database Conference. 1-15.

34. Morrison, R. (1982). Low cost computer graphics for micro computers. Software, Practice & Experience 12 767-776.

35. Anderson, B. (1980). Programming in the home of the future. International Journal of Machine Studies 12 341-365.

36. Bayer, B. & McCreight (1972). Organisation and maintenance of large ordered indexes. Acta Informatica 1 173-189.

37. IBM report (1978). on the contents of a sample of programs surveyed. San Jose ??????

38. Morrison, R. (1982). The string as a simple data type. Sigplan Notices Vol 17,3.

40. Kaehler, T. (1982). Virtual memory for an object-orientated language. Byte 378-387.

41. Traiger, I.L. (1982). Virtual memory management for database systems. ACM.SIGOPS 16,4 26-48.

42. Atkinson, M.P. & Buneman, P. (1983). Survey paper on persistent languages. in preparation.

43. Owoso, G.O. (1983). The persistent programming development environment. in preparation.

44. Bloom, B.H. (1970). Space/time tradeoffs in hash coding with allowable errors. CACM 13,7 422-426.

PART 3

PS-algol : a language for persistent programming

Submitted to the Australian National
Computing Conference, 1983

PS-algol : a language for persistent programming

M.P.Atkinson+, P.J.Bailey*, K.J.Chisholm+, W.P.Cockshott+ and R.Morrison*
+ University of Edinburgh, Scotland
* University of St Andrews, Scotland

Introduction

The long term storage of data has been of concern to programming language designers for some time. Traditional programming languages provide facilities for the manipulation of data whose lifetime does not extend beyond the activation of the program. If data is required to survive a program activation then some file I/O or database management system interface is used. Two views of data evolve from this. Data can either be classed as short term data and would be manipulated by the programming language facilities or data would be long term data in which case it would be manipulated by the file system or the database management system (DBMS). The mapping between the two types of data is usually done by the file system or the DBMS.

These different views of data are highlighted when the data structuring facilities of programming languages and database management systems are compared. Database systems have developed relational, hierarchical, network and functional models of data [1,2,3,4] whereas programming languages may have arrays, records, sets, monitors [5] and abstract data types [6].

That there are two different views of data has certain disadvantages. Firstly in any program there is usually a considerable amount of code, typically 30% of the total [37], concerned with transferring data to and from files or a DBMS. In both cases much space and time is taken up by code to perform translations between the preferred form of data (the program's) and the form used by the storage medium. This is unsatisfactory because of the time taken in writing and executing this mapping code and also because the quality of the application programs may be impaired by the mapping. Frequently the programmer is distracted from his task by the difficulties of understanding and managing the mapping. The translation merely to gain access to long term data should be differentiated from translations from a form appropriate to one use of the data to a form suitable to some other algorithms. Such translations are justified when the two forms cannot coexist and there is a substantial use of both forms. The second major disadvantage is that the data type protection offered by the programming language on its data is often lost across the mapping. The structure that might have been exploited in a program to aid comprehension is neither apparent nor protected and is soon lost.

We feel that there is little to be gained by supporting the coexistence of these two views of data and consequently define a property of data, called persistence that is the period of time for which the data exists and is useable. A spectrum of persistence exists and is categorised by

1. transient results in expression evaluation.
2. local variables in procedure activations.
3. own variables, global variables and heap items whose extent is different from their scope.
4. data that exists between executions of a program.
5. data that exists between various versions of a program.
6. data that outlives the program.

It can be readily seen that the first three persistence categories are usually supported by programming languages and the second three categories by a DBMS. We report here on the PS-algol system which supports persistent programming for all the categories of data above. The system is implemented and is being actively used in a number of projects mentioned later. Here we will describe the language design decisions in implementing persistence, the underlying implementation problems, the results obtained and some thoughts for the future.

The design method

As a first attempt at producing a system to support persistence we hypothesised that it should be possible to add persistence to an existing language with minimal change to the language. Thus the programmer would be faced with the normal task of mastering the programming language but would have the facility of persistence at little or no extra cost.

The language chosen for this was S-algol [7,8], a high level algol used for teaching at the University of St Andrews. Indeed this decision was made by the Edinburgh University team after some trouble with attempts at Algol 68 [9] and Pascal [10] and resulted in the two teams collaborating on the project.

S-algol stands somewhere between Algol W [11] and Algol 68 in

taxonomic clustering. It was designed using three principles first outlined by Strachey [12] and Landin[13]. These are

1. The principle of correspondence
2. The principle of abstraction
3. The principle of data type completeness

The application of the three principles in designing S-algol is described elsewhere [14]. The result is an orthogonal language whose "power is gained from simplicity and its simplicity from generality" [15]. Here we are interested in data and the S-algol universe of discourse can be defined by

1. The scalar data types are int, real, bool, string, pic and file.
2. For any data type T, *T is the data type of a vector with elements of type T.
3. The data type pntr comprises a structure with any number of fields, and any data type in each field.

The world of data objects can be formed by the closure of rule a under the recursive application of rules b and c.

The unusual features of the S-algol universe of discourse are that it has strings as a simple data type [38], pictures as compound data objects and run time checking of structure classes. The picture facility allows the user to produce line drawings in an infinite two dimensional space. It also offers a picture building facility in which the relationship between different sub-pictures is specified by mathematical transformations. A basic set of picture manipulating facilities along with a set of physical drawing attributes for each device is defined [34]. A pntr may roam freely over the world of structures. That is a pntr is not bound to a structure class. However when a pntr is dereferenced using a structure field name, a run time check occurs to ensure the pntr is pointing at a structure with the appropriate field.

Together with our hypothesis of minimal change to the programming language we recognise certain principles for persistent data.

1. persistence independence : the persistence of a data object is independent of how the program manipulates that data object.
2. persistence data type orthogonality : In line with the principle of data type completeness all data objects should be allowed the full range of persistence.
3. The choice of how to provide and identify persistence at a

language level is independent of the choice of data objects in the language.

A number of methods were investigated to identify persistence of data. Some involved associating persistence with the variable name or the type in the declaration. Under the rule of persistence independence these were disallowed. S-algol itself helped to provide the solution. Its data structures already have some limited notion of persistence in that the scope and extent of these objects need not be the same. Such structures are of course heap items and their useability depends on the availability of legal names.

This limited persistence was extended to allow structures to persist beyond the activation of the program. Their use is protected by the fact that the structure accesses are already dynamically checked in S-algol. Thus we have achieved persistence and retained the protection mechanism.

The choice of which data items persist beyond the lifetime of a program was next. We argued, by preaching minimum change, that the system should decide automatically. Such decisions are already taken in a number of languages like S-algol when garbage collection is involved and we therefore felt that reachability, as in garbage collection, was a reasonable choice for identifying persistent objects. However a new origin for the transitive closure of references which differentiates persistent data and transient data is introduced.

PS-algol

Given the constraint of minimal change to S-algol the simplest way to extend the facilities of the language is by adding standard functions and PS-algol is implemented as a number of functional extensions to S-algol. It was therefore not necessary to alter the compiler itself as all the work is done by the run time support. Thus the population of S-algol programmers could now use PS-algol with very little change to their programming style.

The functions added to support persistence were

procedure open.database(**string** database.name,mode,user,password -> **pntr**)
This procedure opens the database and returns a pointer to it. A transaction will be started if this is the first call of the function since the start of the program, the last commit or abandon.

procedure get.root(**pntr** database -> **pntr**)

procedure set.root(**pntr** old.root,new.root)
These procedures get and record the new value of the root for the given database. This root is the origin of the reachability closure defining and giving access to all the persistent data. The effect of set.root becomes visible and preserved when a transaction commits.

procedure commit
Commit the current transaction on all the open databases.

procedure abandon
Abandon the current transaction on all the open databases. Restore their state to the state at the last open or commit.

procedure close.database(**pntr** root.value)
Indicate the database is no longer required after the next commit or abandon.

These are the full set of routines concerned with persistence. However we have found it desirable to provide a set of functions for associative lookup which in fact give access to B-tree algorithms [36] written in terms of PS-algol. These are

procedure table(-> **pntr**)
Create a new empty table.

procedure lookup(**pntr** table ; **string** key -> **pntr**)
Lookup the structure in 'table' using 'key'.

procedure enter(**pntr** table,value ; **string** key -> **pntr**)
Enter the structure 'value' in 'table' using 'key'.

procedure scan(**pntr** table,environment ;
(**pntr**,**pntr**,**string** -> **pntr**) user -> **pntr**)
Apply the function 'user' to every entry in 'table'.

To give a flavour of the language an example of a PS-algol program is given in Appendix I.

Implementation

Since we have made very little change to the compiler we will not mention it here. The main interest is in the control of the movement of data between main store and backing store. This is all controlled by the run time support system. Transactions are implemented by Challis' algorithm [16]. The movement of data is achieved as follows.

a. **Movement of data on to the heap**

Data may be created on the heap during a transaction or it may migrate there as a copy of some persistent data object. The second mechanism is invoked when a pointer being dereferenced turns out to be a persistent identifier (PID). The persistent object manager is called to locate the object and place it on the heap, possibly carrying out minor translations. The initial pointer which is a PID is that yielded by get.root and subsequent pointers will be found in the fields of structures reached from that reference.

b. **Movement from the heap**

When a transaction is committed, all the data on the heap that is reachable from the persistent objects used during the transaction are transferred back to the disk. Some anticipation of these transfers may be necessary if the heap space is insufficient for the whole transaction.

The algorithms and data structures used to implement this data movement economically are described in [17,18]. Since the database may be shared by many programs the binding of the names of persistent data must by dynamic and symbolic. In PS-algol this binding is performed when the object is accessed for the first time and is automatic thereafter. There is no overhead in accessing local objects. A survey of the tradeoffs available in such algorithms is in preparation [19].

Experience using this strategy

At first sight the set of facilities provided by PS-algol may look fairly primitive. Notice however that the programmer never explicitly organises data movement but that it occurs automatically when data is used. Notice also that the language type rules are strictly enforced and that the programmer uses a method already familiar to him to preserve data. That is by the usual naming convention where the preservation of data is a consequence of arranging that there is a way of using the data. Thus we have achieved persistence by minimal change allowing the programmer to use all his familiar techniques of problem solving.

The effect on programs written in PS-algol has been quite dramatic.

In Appendix II we quote some early results of tests comparing programs written in PS-algol with programs written in Pascal with explicit database calls. These programs were to implement a DAPLEX [4] look alike and various CAD and demonstration programs. We have found that there is a reduction by a factor of about three in the size of the source code. These are of course early results and will need further confirmation. However it is the sort of result we expected.

We have also found that the coding time for these programs is reduced by at least the same factor. We suspect that the maintenance of the programs will be easier since it is some function of the number of lines of code being maintained. As would be expected from smaller programs they actually run faster as well but the increase in speed is data dependent and we would not wish to put a figure on it at this stage.

Conclusions

It is apparent from our experiments that it is useful to introduce two new abstractions into programming languages. One, persistence, is the abstraction of long term data storage and transient data storage into one simpler concept, the other, pictures, is an abstraction of all the details of graphical output devices.

Both abstractions have been so successful that we would recommend that other language designers include them in the languages they design. They are successful in three important respects: they both identify significant aspects of common programming tasks consequently reducing the effort required by the programmer to accomplish those tasks, they both abstract away much detail commonly visible to the programmer so that programs produced using them are simpler to understand and to transport and the abstractions leave the language system implementor with a feasible task.

Although the abstraction itself is successful, we note that there are various flavours to be explored. These potential flavours lead us to consideration of new research, and to a comparison with other research. Persistence has appeared as an orthogonal property of data in one other language in the work of Albano et al[20]. In their language ELLE, they extend ML[21] adding persistence. In an attempt to accommodate longer term persistence they make contexts proper objects in the language which can be manipulated, as we suggest for an identified subset of contexts in our language proposals NEPAL[22]. In another group of languages

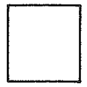
PASCAL/R[23], RIGEL[24] & PLAIN [25] the designers have chosen not to adopt the principle of data type completeness, and have only allowed instances of type relation to have longer term persistence. It is interesting to note that in PASCAL/R the construct DATABASE has a form like a PASCAL record, and if its fields were allowed to take any of the PASCAL data types then that language would be consistent in allowing data structures of any type to have any persistence. But PASCAL/R does not attempt data type completeness in other situations. ADAPLEX[26] is another example of a language constructed by merging a given database model, DAPLEX[4] and a existing language ADA[27], with the inevitable consequence of restrictions on which data types can have which persistence. When casting program examples to assess languages we have found it particularly irksome if data types which can persist cannot also have temporary instances, perhaps more often than we have found difficulty with being unable to construct directly an instance of some temporary data structure. We conclude, therefore, that the consistency of our form of persistence is of advantage to the programmer. So far it has met our programming needs well, but it is possible that there are applications for which it is not well suited. This would not invalidate the idea of incorporating an abstraction of persistence into a programming language but would suggest a different flavour is needed.

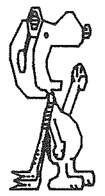
Where next with Pictures?

A similar consideration can be given to the idea of pictures. Other models of graphical output are possible, such as filled areas for bitmap graphics[28], forms[29,30] and default icons for browsing[31]. The set of operators used can vary, for example Cardelli[28] proposes automatic alignment as boxes are abutted vertically and horizontally, but does not permit simple superposition. A particular value of the introduction of pictures is the device independence they provide. But we need to consider abstractions for input as well as output, via the range of graphical and pointing devices, we need to accommodate area fill graphics and provide more readable constant denotations. For example

```
    let sq = [ 0,0 ] ^ [ 0,1 ] ^ [ 1,1 ] ^ [ 1,0 ]
```

might be a denotation for a square, but it is hard to see that at a glance, or indeed to notice the error, whereas

let sq = 

let snoopy = 

each have an obvious meaning, visible at a glance, but the second would be extremely tedious to include in present notations. We therefore conclude that an important adjunct to including a new type in a language is to provide a natural notation for it.

Where next with persistence?

Similarly we are not satisfied with our present form of persistence. Measurement, experimental use and exploration of methods of implementation are all continuing, but our main effort is directed at incorporating in our abstraction some of the other concerns of databases. For example, large scale data often implies sharing, which suggests the need for views, protection and concurrent access. The provision of views appears equivalent to the provision of abstract data types, and we postulate that adding persistence to a language with good abstract data facilities will meet this need. The provision of protection divides into : protection from users, protection from software and protection from hardware failure. The first two can be enforced by restricting access to the abstract data types, by relying on the program development system thus enforcing scope rules using the compiler. The third is handled already by our transaction mechanism.

There do not appear to be simple solutions to the concurrency problem. Many solutions are possible in simple cases [16,23,33]. That is, where each transaction is short, independent of human interaction with the data obtained during that transaction and where the probability of conflicting requests is small. However, in most circumstances automatic and general methods of handling a conflict are not appropriate. For example, if two people have done extensive design work, one doesn't wish to start again because of some minor overlapping change. The solution appears to be to have a notation with which to inform each program involved of the conflict in such a way that the program can respond to the exact form of the conflict. At present this is not possible, as it is not clear how to present the multiple states of data (before and after) and the data from different computations to each program. This must be solved, in order to make concurrent use of systems properly available to programmers. The most promising approach to this is to depend on object-based programming, with messages between programs which can effectively transport copies of or the properties of an object.

A particular problem peculiar to very long term persistence is that data structure and type has to evolve to meet changing needs. Consequently a necessary feature of a language designed with persistence in mind will be support of type and representation development. This would lead to a language which is properly interactive, as is suggested by Anderson [35]. Were this achieved, we still need the language to provide immediate evaluation expressions which meet query language needs, and to support as a continuum, all more complex uses, up to large scale system building. It is at least interesting to try to discover whether support of such a range of activities is achievable.

Acknowledgements

The work at Edinburgh was supported in part by SERC grant GRA 86541.

References

1. Codd, E.F. (1970). A relational model for large shared databases Comm.ACM 13,6 377-387.
2. Lochovsky, F.H. & Tsichritizis, D.C. (1978). Hierarchical database management systems ACM Computing Surveys 8,1 105-123.
3. Taylor, R.C & Frank, R.L. (1976). CODASYL database management systems ACM Computing Surveys 8,1 67-103.
4. Shipman, D.W. (1981). The functional data model and the data language DAPLEX ACM.TODS 6,1 140-173.
5. Hoare, C.A.R. (1974). Monitors : an operating system structuring concept Comm.ACM 17,10 549-557.
6. Liskov, B.H. et al (1977). Abstraction mechanisms in CLU. Comm.ACM 20,8 564-576.
7. Morrison, R. (1979). S-algol language reference manual. University of St Andrews CS/79/1.

8. Cole, A.J. & Morrison, R. (1982). An introduction to programming with S-algol. Cambridge University Press.
9. van Wijngaarden, A. et al (1969). Report on the algorithmic language Algol 68. Numerische Mathematik 14 79-218.
10. Wirth, N. (1971). The programming language Pascal. Acta Informatica 1 35-63.
11. Wirth, N. & Hoare, C.A.R. (1966). A contribution to the development of algol. Comm.ACM 9,6 413-431.
12. Strachey, C. (1967). Fundamental concepts in programming languages. Oxford University Press.
13. Landin, P.J. (1966). The next 700 programming languages. Comm.ACM 9,3 157-164.
14. Morrison R. (1979). Ph.d. Thesis University of St Andrews.
15. van Wijngaarden, A. (1963). Generalised algol. Annual Review of automatic programming 3 17-26.
16. Challis, M.P. (1978). Data Consistency and integrity in a multi-user environment. In Databases : improving usability and responsiveness Academic Press 245-270.
17. Atkinson, M.P., Cockshott, W.P. & Chisholm, K.J. (1982). CMS : A chunk management system. accepted for publication Software, Practice & Experience
18. Atkinson, M.P., Cockshott, W.P., Chisholm, K.J. & Marshall, R.M. (1982). Algorithms for a persistent heap. accepted for publication Software, Practice & Experience
19. Atkinson, M.P., Cockshott, W.P. & Chisholm, K.J. (1982). An exploration of various strategies for implementing persistence as an orthogonal property of data. in preparation.
20. Albano, A., Occiuto, M.E. & Orsini, R. (1982). A uniform management of temporary and persistent complex data in high level languages. DATABASE Infotech State of the Art Report 9,8 435-458.
21. Gordon, M.J.C., Milner, A.J.R.G., Morris, L., Newey, M. & Wadsworth, C. (1978). A metalanguage for interactive proof in LCF. ACM.POPL
22. Atkinson, M.P., Cockshott, W.P. & Chisholm, K.J. (1982). NEPAL - The New Edinburgh Persistent Algorithmic Language. DATABASE Infotech State of the Art Report 9,8 299-318.
23. Schmidt, J.W. (1977). Some high level language constructs for data of type relation . ACM.TODS 2,3 247-261.

24. Rowe L.A. Reference manual for the programming language RIGEL.
25. Wasserman, A.I., Sheretz, D.D., Kersten,M.L., van de Reit, R.D. (1981). Revised report on the programming language PLAIN. ACM SIGPLAN Notices 16,5.
26. Smith, J.M., Fox, S. & Landers, T. (1981). Reference manual for ADAPLEX. Computer Corporation of America, Cambridge, Massachusetts
27. Ichbiah et al (1979). Rationale of the design of the programming language Ada. ACM Sigplan Notices 14,6.
28. Cardelli, L. (1982). The semantics of a two dimensional language. University of Edinburgh, Department of Computer Science Report CSR-115-82.
29. Rowe, L. & Skeans Screen RIGEL. University of California at Berkeley, Department of Electrical Engineering.
30. Shu, N.C. & Lum, V.Y. (1982). Specification of Forms Processing and Business procedures for Office Automation. IEEE transactions on Software Engineering, SE-8,5 499-512.
31. Tesler, L. (1982). The Samlltalk environment. Byte 90-147.
32. Kung, H.T. & Robinson, J.T. (1982). On Optimistic Methods for Concurrency Control. ACM.TODS 6,2 213-226.
33. Date, C.J. (1979). Locking and recovery in a shared database system : An application tutorial. in proceedings of the 5th Very Large Database Conference. 1-15.
34. Morrison, R. (1982). Low cost computer graphics for micro computers. Software, Practice & Experience 12 767-776.
35. Anderson, B. (1980). Programming in the home of the future. International Journal of Machine Studies 12 341-365.
36. Bayer, B. & McCreight (1972). Organisation and maintenance of large ordered indexes. Acta Informatica 1 173-189.
37. IBM report (1978). on the contents of a sample of programs surveyed. San Jose ??????
38. Morrison, R. (1982). The string as a simple data type. Sigplan Notices Vol 17,3.

Appendix I

```
structure person( string name,phone.no ; pntr addr )
structure address( int no ; string street,town )
write "Name : "           ; let this.name = read.a.line
write "Phone number : "   ; let this.phone = read.a.line
write "House number : "   ; let this.house = readi
write "Street : "         ; let this.street = read.a.line
write "Town : "           ; let this.town = read.a.line
let p = person(this.name,this.phone,address(this.house,this.street,this.town))
let db = open.database( "Address.list","write","Ron","Ann" )
if db = nil then write "Invalid opening of a database" else
begin
    enter( this.name,get.root( db ),p )
    commit
end
```

```
structure person( string name,phone.no ; pntr addr )
structure address( int no ; string street,town )
write "Name : "    ; let this.name = read.a.line
let db = open.database( "Address.list","read","Ron","Ann" )
let this.person = lookup( this.name,get.root( db ) )
if this.person = nil then write "This person's name cannot be found " else
begin
    write "Phone number is : ",this.person( phone.no ),
          "Home address is : ",this.person( addr,no )," ",
          this.person( addr,street ),"'n",
          this.person( addr,town ),"'n"
end
```

PART 4

Some thoughts about the problems with persistent languages

Presented at the Workshop on Database Interfaces held at the University of Pennsylvania, October 1983, organised by P. Buneman.

A revised version will appear in the proceedings of the workshop

Some thoughts about the problems with persistent languages

M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott

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

Introduction

There is no doubt that provision of persistence is important. A large proportion of all code written is written to provide persistence. In most application programs it is either code to get data from files and code to write data to files or it is code to transfer data to and from a Database Management system. In both cases there is usually much code to perform translations between the prefered form for the program's task and the form used in the storage medium.

Not only is this unsatisfactory because of the cost of writing essentially unnecessary code, but also since it has a deletarious effect on the quality of the systems so produced. Often the form of the program is ruined by the intrusion of this mapping, certainly it is obscured. Frequently the programmer is distracted by the difficulties of understanding and managing the mapping. Inevitably the benefits of a good programming language which provides type protection is lost as mappings may be inconsistent in different programs. The structure that might have been exploited in program and data, to aid comprehension, especially during system maintenance, is neither apparent nor protected, so it is soon lost.

Our story so far

By introducing persistence as an orthogonal, variable property of data we have been able to much simplify the provision of access to data which persists longer than the execution times of individual programs. We recognise certain principles:

1) persistence independent programming: It should be possible to apply any sequence of code written in a language to data of any persistence.

2) consistency in the availability of persistence: If data of one type may have a particular persistence, then data of any type may have that persistence.

3) the choice at a language level of how to provide and identify persistence does not need to be confused with the choice of what objects the language should be capable of manipulating.

To experiment with these ideas we have developed the programming language PS-Algol [1] from the language S-Algol [3,4] which was chosen as the basis of experiments because of its simplicity, its simple compiler and its type checking rules. Three ways of dealing with the identification of the degree of persistence that were considered and rejected were: making it a property of variables denoted at declaration time. All these were rejected as not permitting persistence independent programming. The final choice was to use reachability just as persistence on a conventional heap is defined by reachability, thus all data reachable from a root associated with each database must be preserved at the end of a transaction.

Some implementation details

It was not necessary to change the compiler at all since there are no syntactic changes to the language. A small number of routines were added to the language, idealised these are:

```
procedure open.database (string database.name, mode, user, password ->
pntr)
```

This opens the database, starting a transaction if this is the first call of this routine since the last abandon, commit or start of program, yielding the current value of the root.

```
procedure set.database (pntr old.root, new.root)
```

Record the new value the database identified by old.root is to take when and if the transaction commits.

```
procedure commit
```

Commit the current transaction on all the currently open databases.

```
procedure abandon
```

Abandon the current transaction on all the currently open databases, restoring their state to the last open or commit.

```
procedure close.database (pntr root.value)
```

Indicate the database is no longer required after the next commit or abandon.

These are the full set of routines concerned with persistence, but we have found it desirable to provide a set of associative routines, which in fact give access to B-Tree algorithms written in terms of PS-Algol.

```
procedure table ( ->pntr)
```

create a new, empty table

```
procedure lookup (pntr table; string key  -> pntr)
procedure enter  (pntr table; string key  -> pntr)
```

The tables provided are in fact maps from strings to structures identified by a pointer. Ideally we would provide a map from any type to another if we had suitable polymorphic functions. Lookup, given a key yields a value from the given table or nil if there is no such key in the table and enter enters a new key value pair, or removes the old entry if the given value is nil.

```
procedure scan (pntr table, environment; (pntr, pntr, string -> pntr)
                            user -> pntr)
```

is a function which applies the function user to every entry in a table.

In effect, providing such a library of routines is a step towards introducing a new type constructor for maps. It is also an example of the way the persistent programming environment may be used. We return to that later.

Transactions are implemented by Challis' algorithm [5,6]. The movement of data is achieved as follows:

a) Movement onto the heap
Data is either created during a transaction on the heap, or it migrates there as a copy of some persistent data object. This second mechanism occurs when a pointer is dereferenced, and that pointer is a PID (Persistent IDentifier) which is an object address which ultimately specified the site of the required data on some disc. In which case the object is copied, translated to the currently expected form and placed on the heap.

b) Movement from the heap
When a transaction is committed, all the data on the heap, which are reachable from the persistent objects which have been brought in and updated during the transaction, are transferred back to the database. Some anticipation of these transfers before commitment may be necessary if space for data on the heap is insufficient for the transaction.

Algorithms and data structures used to implement this data movement economically are described elsewhere [7].

Experience using this strategy

At first sight, the set of facilities proposed here may look fairly primitive, but note that:

i) the programmer never explicitly organises data movement, it happens when he tries to use data,

ii) type rules can be strictly enforced

andiii)the programmer uses a method already familiar to him to identify which data needs to be kept. Indeed preservation of data is a consequence of arranging that there is a way of using that data.

The direct effect has been a reduction by a factor of about three in the size of the source code to implement the same tasks, in PS-Algol compared with implementing them in PASCAL-like languages with explicit data base calls. Coding time has been reduced by at least this factor.

These observations are based on the work of various students and colleagues using the language, and need confirmation by further experiment.

With our recent algorithms, despite the present version of the language being implemented using an interpreter, we have also found that these programs then run using less CPU time and less disc traffic than the programs called using the PASCAL-like languages and explicit database calls. For example, our B-Tree implementation was originally in such a language. When we reimplemented it, using the dereference driven data migration provided by PS-Algol it ran about five times faster.

The experimental implementations built in the language at present include a relational system, a DAPLEX [8] look-alike and various specific CAD and demonstration programs. We have found it a very good environment for building such higher level systems. The fact that pointers are properly supported permits programmers to build persistent structures which match well their particular problems and intended algorithms.

Where next?

In the short term we are improving algorithms, the scale of data supported and portability [9]. Next we expect to make functions first class data objects, that can be yielded by functions and be stored in persistent storage. This will greatly increase the power of the language as a means of controlling and packaging the access to data structures.

In the longer term we envisage adding similar support for persistence to a much improved language, and influencing others into incorporating support of persistence in their languages as an orthogonal property of data.

Ideally we would like to support much longer term persistence.

The Problems

1) What is a "much improved language"? One of S-Algol's greatest assets is its simplicity. How can we improve the language without losing that? What do we want?: polymorphic abstract data types influenced by ML [10], HOPE [11], and POLY [12]. Do we need modules and blocks - it seems difficult to have simple and consistent rules for both. We need a constructor for types to whose instances bulk operations apply. What should this constructor be - relations, as in RIGEL [13], PLAIN [14] and PASCAL/R [15] or sequences as in FQL [16]. Possibly we need to explore further the idea of tables, as they seem immensely popular with programmers. Can we generalise array, table, function and structure (record) into one concept for the programmer, the map? How can we support a continuum of uses from the simplest formal query languages to the most sophisticated application programs? It would be nice if they were uses of different subsets (or libraries) in the same language. Is this possible? One thing is certain - the language should be usable interactively. Does this change our views on the previous issues? Do we experiment by a) changing the existing language,

b) adding persistence to some other language, or c) starting a new language afresh? There is a sense that serious consideration of use of persistent data will modify most other aspects of language design.

2) What should be done about concurrency? Can we wait for languages which provide good concurrent programming and then add persistence? Alternatively should the concurrency be controlled outside the language? In either case we appear to lack good models and good implementation methods. Can scope rules be used to economise on implementation costs? [17]

3) What do we do to make systems safe? It is necessary to protect data from hardware, software and human failure to ensure that it achieves the intended longevity. Is it sufficient and practicable to depend on type rules and scope rules to do this? I would postulate that it is.

4) How do we achieve really long term persistence? The essence of a database is not scale, but the need to keep data long after the original concept of how it should be organised and used has changed. A particular, but relatively simple case of this is that the set of programs which use it change. This introduces the problem of what are sensible type equivalence rules between programs, and how they should be implemented [18]. In a monolinguistic culture that is a problem which, in a polylinguistic culture, is generally insuperable [17]. But the organisation of the data changes too - and must be changed by programs in our language. How do they do that? Even the set of structures in a database cannot be changed in most languages let alone the structures themselves. For example, in PASCAL/R a variable of type Database cannot be manipulated to add fields corresponding to extra relations. It is helpful perhaps to think of this as a problem of making contexts proper objects in the language and providing operators on these contexts. This we do not know how to do, but it requires that the language and the program development environment become one entity. One proposal by us is the NEPAL language [19] and a proposal has also been made by Albano et al [20]. Manipulation of contexts, and of metadata, where the contexts contain polymorphic abstract data types and this can be viewed as a hierarchy where the metadata at one level can be viewed as data at the next may meet this need. This itself seems to imply that the operations necessary for program development have to be captured within the language, making a closed system. We are not sufficiently clear of our concepts at present to be close to developing such a total language.

5) Persistent data development aids are needed. Given very long term data there is a need a) to be able to find out how it is organised, when a new user starts using it or a user returns to it, and b) to be able to examine the actual data and possibly its history. This extends from browsing and query aids to detailed exploratory tools. It is also necessary to have analysis and possibly repair tools [21].

Conclusion

There is plenty to challenge us in the design of good long term persistence languages, but the value of trying to define languages is twofold: it forces clarification of concepts and when achieved greatly simplifies the use of those concepts for others. It is important to convince language designers that the provision of persistence is an important consideration in their language designs.

REFERENCES

[1] 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 1982) (Also as University of Edinburgh, Department of Computer Science, Research Report CSR-94-81.)

[2] Not used

[3] Cole, A.J. & Morrison, R. "S-Algol Reference Manual" University of St Andrews Scotland 1980 . (Also as University of Edinburgh, Department of Computer Science, Research Report CSR-80-81)

[4] Morrison, R. Programming in S-Algol Cambridge University Press: Cambridge, England. October 1982

[5] Challis, M.F. "Version management - or how to implement transactions without a recovery log" in DATABASE Infotech State of the Art Report Series 9, No. 8 (January 1982) 435-458

[6] Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. "CMS: a chunk management system" To appear in Software Practice and Experience. (Also as University of Edinburgh, Department of Computer Science, Research Report CSR-110-82 April 1982)

[7] Atkinson, M.P., Chisholm K.J., Cockshott, W.P. & Marshall, R.M., "Algorithms for a persistent heap" to appear in Software Practice and Experience. (Also as University of Edinburgh, Department of Computer Science, Research Report CSR-109-82.)

[8] Shipman, D.W. "The functional data model and the data language DAPLEX" ACM TODS Vol.6, No. 1 (March 1981)

[9] Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. "POMS: a persistent object management system" Working documentation, available from the author October 1982

[10] Gordon, M.J.C., Milner, A.J.R.G., Morris, L., Newey,M. & Wadsworth, C. "A metalanguage for interactive proof in LCF" in procedings of ACM conference on Principles of Programming Languages, 1978

[11] Burstall, R.M., McQueen, D. and Sanella, D.T. "HOPE: an experimental applicative language" in conference record of the 1980 LISP conference.

[12] Mathews, D.C.J. "POLY report" & "Introduction to POLY" Technical reports 28 & 29 University of Cambridge, Computer Laboratory, Corn Exchange Street, Cambridge, August & May 1982

[13] Rowe, L.A. "Reference manual for the programming language RIGEL"

[14] Wasserman, A.I., Sheretz, D.D., Kersten, M.L., van de Reit, R.D. "Revised report on the programming language PLAIN" ACM SIGPLAN Notices Vol.16, No. 5, (May 1981)

[15] Schmidt, J.W. "Some high level language constructs for data of type relation" ACM TODS Vol.2, No. 3 (September 1977)247-261.

[16] Buneman, O.P., Frankel, R.E. & Nikhil, R. "An implementational

technique for database query languages" ACM TODS, June 1982

[17] Cockshott W.P. "Orthoganal Persistance: An abstract representation of disc storage in Algol like languages" Ph D Thesis, University Of Edinburgh. (Copies will be available as a Computer Science Departmental Report.)

[18] Atkinson, M.P. & Owoso, G.O. "Type checking between programs in a persistent language" in preparation

[19] Atkinson, M.P., Chisholm, C.J. & Cockshott W.P. "The New Edinburgh Persistent Algorithmic Language" in DATABASE (see 5) 299-318

[20] Albano, A., Occiuto, M.E. & Orsini, R. "A uniform management of temporary and persistent complex data in high level languages" in DATABASE (see 5) 319 - 344

[21] Owoso G.O. various working papers available from the author. 1982

PART 5

Temporary user specification of PS-algol persistence related procedures.

Temporary User Reference Information

This document is made available pending the production of the new PS-algol reference manual to provide a definitive description of the currently available routines concerned with databases and tables only. The functions to support persistence are:

```
procedure open.database(string database.name,password,mode -> pntr)
```

This procedure attempts to open the database specified by database.name (which in general is a path[1] down a tree of directories, terminating in the database's name) in the mode (read or write) specified by mode[2], quoting password[3] to establish this right. If the open fails the result is a pointer to an error record[4]. If it succeeds the result is a pointer to a table (see belwo) which contains a set of name-value pairs[5] and the the root from which preserved data is identified by transitive closure of reachability. A table is chosen so as to permit programs which use one route to the data to be independent of programs using other routes. Many databases may be open for reading, but only one may be open for writing so that the destination for new persistent data may be deduced. If this is the first successful open.database then a transaction is started. If the mode is write, and the database name has not occurred before, a new database with the given name and password will be created.

```
procedure commit
```

This procedure commits the changes made so far to the database open for writing. The program may continue to make further changes. Only changes made before the last commit will be recorded, so that not performing a commit is equivalent to aborting the transaction.

```
procedure close.database(string database.name)
```

This procedure closes the specified database[1] making it available for other users who wish to write to it (multiple readers are allowed). It takes effect immediately. An implicit close database is applied to every database open at the end of the program. When a database has been closed references to objects not yet imported may remain accessible as may any imported data. An error is detected if the programmer tries to access any data that has not been imported from the closed database. A program may reopen a database it has closed but to avoid inconsistency between data that was held in the program and the database, it may only do so if there has not been an intervening write to that database.

Specification of the PS-algol tables procedures

There are also a set of procedures to manipulate tables, a mechanism for associative lookup, implemented as trees or equivalent algorithms. A table is an ordered set of pairs. Each pair consists of a key and a value. At present the key may be an integer or string value, and the value is a pointer to a structure instance or table. The basic library for tables is:

procedure table(-> pntr)

This procedure creates an empty table represented by an instance of the structure class Table. (Note that the structure class has an upper case tee while the procedure name is all in lower case.)

procedure s.lookup(string key ; pntr table -> pntr)

procedure i.lookup(int key ; pntr table -> pntr)

These procedures take a given key key and considering only the pairs within the table return the last value stored by a call of s.enter or i.enter for this key or nil if there is no such pair. Note that key must be of the correct type. IE 1 and "1" are different key values.

procedure s.enter(string key ; pntr table,value)

procedure i.enter(int key ; pntr table,value)

These procedures store in the given table a pair or if the supplied value is nil delete the previously stored pair for the given key. (Note any previously stored value for key will be lost.)

```
procedure s.scan(pntr table,environment ;
                 (string,pntr,pntr -> bool)user -> int)
```

```
procedure i.scan(pntr table,environment ;
                 (int,ntr,pntr -> bool)user -> int)
```

These procedures provide iteration over tables. The function user is applied to every pair with a key of the appropriate type, in ascending order of integers or lexical order of strings, until either it yields false or the whole table has been scanned. The first parameter supplied to the function user is the key, the second is the corresponding value and the third parameter the value given as environment. The result of these scan procedures is the number of times user is applied. An example of their use is given in figure 3.

procedure cardinality(pntr table -> int)

This procedure returns the current number of entries in the given table (without scanning the table).

Tutorial examples

The table facilities were envisaged as a way of packaging index construction. For example singly and multiple indexed relations can readily be constructed using them. They have also proved very popular as a dynamic data structure constructor. We present as an example of PS-algol programs using them, three programs to maintain a simple address list. The program in figure 1 adds a new person, that in figure 2 looks up the telephone number of a person and that in figure 3 finds the longest telephone number.

```
structure person( string name,phone.no ; pntr addr,other )
structure address( int no ; string street,town ; pntr next.addr )
let db = open.database( "Address.list","Morwenna","write" )
if db is error.record then write "Can't open database" else
begin
     write "Name : "                 ; let this.name  = read.a.line
     write "Phone number : "         ; let this.phone = read.a.line
     write "House number : "         ; let this.house = readi
     write "Street : "               ; let this.street= read.a.line
     write "Town : "                 ; let this.town  = read.a.line
     let p = person( this.name,this.phone,address( this.house,this.street,
                     this.town,nil ),nil )
     let addr.list = s.lookup( "addr.list.by.name",db )
     s.enter( this.name,addr.list,p )
     commit
end
```

A program to add one person to the address list

Figure 1

```
structure person( string name,phone.no ; pntr addr,other )
let db = open.database( "Address.list","Kernow","read" )
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup( "addr.list.by.name",db )
write "Name : "               ; let this.name = read.a.line
let this.person = s.lookup( this.name,addr.list )
if this.person = nil then write "Person not known" else
   write "Phone number is : ",this.person( phone.no )
```

A program to look up the telephone number of one person

Figure 2

```
structure person( string name,phone.no ; pntr addr,other )
let db = open.database( "Address.list", "Kernow","read" )
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup( "addr.list.by.name",db )
structure env( int max ; string longest )
let e1 = env( 0,"" )
procedure phone( string name ; pntr val,e1 -> bool )
begin
     let number = val( phone.no )
     let len = length( number )
     if len > e1( max ) do { e1( max ) := len ; e1( longest ) := number }
     true
end
let count = s.scan( addr.list,e1,phone )
if count = 0 then write "Nobody in the list yet" else
   write "Longest telephone number is : ",e1( longest )
```

A program to find the longest telephone number in the address list

Figure 3

Note that in these examples the declared data structures are different. This identifies which parts of the data each program may touch and so behaves like a simple database view mechanism. Only those quoted will be matched against the stored set of data descriptions when they are first used. Paths to other data, eg alternative addresses via next.addr and extra information about people via other has not been used. It is there to give access to data pertinent to other programs or to future needs and does not clutter programs which do not use it. Similarly the standard table is used to name access paths and only those used need be considered in a particular program. Figure 4 shows a program to introduce a new access path to the address list. With these disciplines, which we find useful, data design and programs can grow together. These examples illustrate the ease with which small programs operating on persistent data may be composed to provide a complete system. In this example one would imagine other programs: to start an address list, to delete it, to remove entries, amend addresses, print lists etc. We believe this method of constructing software tools to be an attractive way of building such systems.

As well as showing these features of persistence the programs illustrate features of the parent language S-algol. For example, the type of a name is deduced from the initialising expression, this leads to conciseness. The freedom to place declarations where they are needed means that constant names are common and programs may be read without searching far for declarations.

```
structure person( string name,phone.no ; pntr addr,other )
let db = open.database( "Address.list","Morwenna","write" )
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup( "addr.list.by.name",db )
let phone.number.table = table      !create a new empty table
procedure put.it.in.phone.number.table( string name;pntr val,env -> bool )
   { s.enter( val( phone.no ),phone.number.table,val ) ; true }
let count = s.scan( addr.list,nil,put.it.in.phone.number.table )
s.enter( "addr.list.by.phone.number",db,phone.number.table )
commit
```

A program to construct a new index onto the address list by phone number

Figure 4

Notes referred to by superscripts in the preceding text giving further detail

notes on open.database

1) database.names

A database is named by a path down a tree of directories, by a string of the form
    <dirname>/<dirname>.....<dbname>
where the first <dirname> identifies an entry in the base directory, the next <dirname> identified an entry in that directory and so on, until the name following the last oblique stroke (or the only name for a database in the base directory) is the name of a database in the directory that the path so far identifies.

The individual names <dirname> and <dbname> are identifiers comprised of letters (case significant), digits, dot or underscore, and may not exceed 15 characters in length. The path may not be more than 7 directories long.

The set of databases open must be compatible. In the POMS implementation this means that a database opened for reading will have to be one which has previously been established by use of the CLEAN utility as referencible from any database currently open for writing. Conversely if a database is opened for writing all the currently open databases must already have the reference permission established for this database.

2) modes.

The mode must be one of "read" or "write", and must be presented in lower case. In read mode, the programmer may alter the structures that have been brought from the persistent store, but these will not be recorded in the persistent store even if he attempts to commit.

Many databases may be opened simultaneously for reading, but only one for writing. If a database is already open for reading by some other process, it may be opened again for reading or writing. Existing readers will not see the effects of the write transaction. Once a database is open for writing, no other processes may open it for reading or writing, until that transaction has completed or aborted.

The password is a string of up to 16 characters. The utility CLEAN may be used to change the password.

3) The structure class error.record is defined as

structure error.record (string error.context,error.fault,error.explain)

All error returns from open.database will have error.context set to "open.database".

The rest of the possible errors are:

error.fault	error.explain
"bad.name"	"<database.name> incorrectly formatted"
"nonexistent"	"<database.name> not found when opening in read mode"
"incompatible"	"<database.name> not compatible with the other databases open"
"no.access"	"<database.name> and <password> do not permit open in <mode> mode"
"no.dir"	"<dir.name> does not exist in directory <path>"
"no.entry"	"<path> directory has its quota of databases"
"no.space"	"<path> cannot be created, due to lack of disc space"
"too.many"	"maximum number of databases open for this runtime system"
"write.already"	"<path> not opened as you already have <path> open for writing"
"changed"	"<path> has been changed since this program last had it"

5) contents of the root table

A root table is provided as a standarad root object, rather than allowing the programmer to choose the first object in the closure of reachable objects. This is to establish a programming style which permits there to be objects in the database about which the programmer need not be cognisant. The system exploits this, by reserving the right to put a small number of entries in this table. However, to avoid clashes with the user's names all the names used in the system will begin with the string "PS-algol$".

notes on commit

6) erroneous commits

The only possible erroneous commits that may occur are:

a) No database open for writing
b) Insufficient space in the open database to write the new data.

Either of these are fatal to the process, causing the transaction to abort and the program to terminate with an error condition.

Papers published by the Data Curator Group

Bibliography

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 Confer- on Very Large Data Bases, Berlin, (Ed. S.B.Yao), IEEE, Sept. 78, 408-419, (A revised version of this is available from the department as CSR-26-78).

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

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 departmental report CSR-43-79.

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

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

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

Atkinson, M.P. (ed.) 'Databases' Pergammon Infotech State of the Art Report, Series 9, No.8, January, 1982. (535 pages, about half of which I wrote)

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

Davie, A.J.T., & Morrison, R. Low cost computer graphics for micro computers. Software, Practce and Experience, 12, 1981, 767-776.

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

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 Departmental Report CSR-94-81

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

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.	'Algorithms for a persistent Heap' Software Practice and Experience, Vol.13, No.3 (March 1983). Also as Departmental Report CSR-109-82
Morrison, R.	The string as a simple data type. Sigplan Notices, Vol.17,3, 1982.
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.	'CMS - A chunk management system' Software Practice and Experience, Vol.13, No.3 (March 1983). Also as Departmental Report CSR-110-82.
Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.	"Progress with Persistent Programming" presented at CREST course UEA, September 1982, to be published revised in <u>Databases - Role and Structure</u>
Stocker, P.M., Atkinson, M.P. & Gray P.M.D. (eds.)	<u>Databases - Role and Structure</u> To be published by CUP. 1983
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. To be published (revised) in the workshop proceedings 1983.
Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., & Morrison, R.	'Current progress with persistent programming' To be presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.
Stocker, P.M., Atkinson, M.P. et al	"Proteus : a distributed database system", to be published in <u>Database - Role and Structure</u>, CUP 1983
Atkinson, M.P., Bailey P.J., Chisholm, K.J., Cockshott, W.P., & Morrison, R.	'An approach to persistent programming' accepted to be published in The Computer Journal, 1983
Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R	"PS-algol : the current position in experiments with persistent languages", to be published in <u>Database - Role and Structure</u>, CUP 1983

<u>Submitted for publication</u>

Atkinson, M.P., Bailey P.J., Chisholm, K.J., Cockshott W.P., & Morrison R.	"Ps-algol a language for persistent programming" submitted to the Australian Computer Conference 1983.
Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., & Morrison, R	POMS : a persistent object management system", submitted to Software Practice and Experience
Atkinson, M.P. & Kulkarni, K.G.	"Experimenting with the Functional Data Model" submitted to the Ninth International Conference on Very Large Databases, October 1983.