PS-algol : a language for persistent programming

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

[¥]Department of Computer Science, University of Edinburgh, Scotland

[†]Department of Computational Science, University of St Andrews, Scotland

Abstract

PS-algol is the first language in a family that introduces the concept of persistence as a property of data. The persistence of data is the time extent over which the data may be used. This paper introduces the general design principles for persistent programming languages and describes our first attempt, PS-algol

1. 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.

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 **p**ersistent **id**entifier (**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.

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.

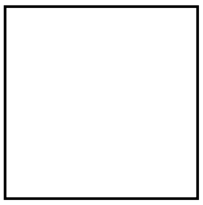
Where next with Pictures?

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

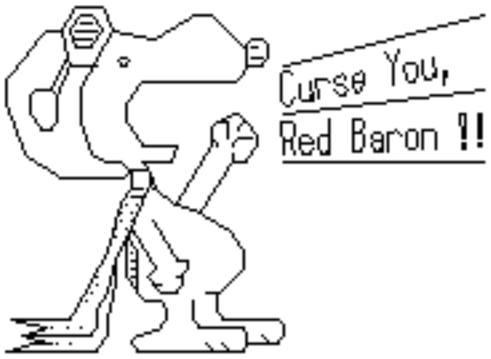
let sq = [0,0] ^ [0,1] ^ [1,1] ^ [1,0]

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
fBwrite "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
```