Progress with Persistence in Poly and Poly/ML*

David C.J. Matthews

Abstract

The paper describes the progress made on using persistence in the Poly programming language and some of the issues which still have to be resolved.

For some time Poly has been implemented on a persistent storage system which has allowed objects to be loaded transparently into store and modified objects to be written out again. This has proved to be a convenient way of allowing large systems to be developed without the overheads of loading the whole of a core image into store. It was however necessary for each user to have their own copy of the whole system since there was no way of sharing data between users.

A new version of the system has been developed in which databases can be shared between users. Each user has their own "home" database in which they keep their private data. They can also make use of data in shared databases which can only be read from. The system is arranged as a tree structure with a root database containing objects which are likely to be generally useful, and below that specialised databases, with the leaves of the tree being the users' databases. This structure allows databases to be opened automatically as persistent addresses are used so maintaining the transparency of the system to the users. The present scheme allows shared databases only to be opened for reading so that objects read from them are not written back. To this extent the behaviour is not transparent.

The paper also describes work on persistence in relation to processes and windows.

This paper was presented at the 1987 Appin Workshop on Persistence and reprinted as Computer Laboratory Technical Report No. ???

*This paper was presented at the Second Appin Workshop on Data Types and Persistence, August 1987.

1 Background

There has long been a division in programming languages between those which are used in essentially a "batch" mode, where a source file is compiled into an object file, and the interactive languages, where the source text is compiled as it is typed in. The differences are in a sense more to do with the implementations than with the languages themselves and even in the interactive case there is usually a command to compile source text from a file. On the whole though languages tend to lend themselves better to one implementation or the other.

Apart from the way in which programs are constructed there is another difference between interactive and non-interactive languages. Non-interactive languages are compiled into separate programs and any communication between them must be through the operating system. This usually means through a text file or something similar. In contrast in an interactive session functions, or the equivalent, are applied to generalised data and generate data. The data may often be structured, such as lists or trees. Within a session this is easy to arrange, but we would often like to be able to suspend a session and come back later. This means that structured data have to be converted into a file that can be held in the filing system. The non-interactive system, which works on files anyway, does not have the same problem.

The conventional solution is simply to write out the contents of the memory at the end of the session and then read it back in at the start of the next[5]. This "workspace" or "core-image" idea allows a user to develop a system over a number of sessions. It has the advantage that the structuring and naming used is appropriate for the language rather than being imposed by the operating system. The disadvantage is that the facilities provided by the operating system for sharing between users is not available. The workspace has to be treated as a single entity and it is difficult to isolate a useful function from the workspace and make it available to other users, though[5] describes how a variable or function could be copied into the active workspace from another. In addition, since the whole workspace has to be read in to store there is the problem that it may grow too big to fit into the available memory. Solutions to this problem[3][7] have tended to require changes to the operating system rather than a system which would run on top of an existing system without modification.

2 Persistence

The idea of using persistent storage as a convenient way of implementing databases was proposed and implemented by M.P. Atkinson[1]. He noted that programs operating on databases often have two representations of data, one used internally by the program while it is running, and a different representation for the same data when held in the database or filing system. The program was forced to spend a considerable time translating between the two representations, because the types provided by the database for persistent data were different from those provided by the programming language for transient data. His solution was to suggest that persistence of data should be a property which could be possessed by a value independently of any other property. In particular whether an object could persist was not associated with its type. The programmer could then choose a representation for the data purely on the basis of the algorithm and not in order to satisfy the requirements of a database system.

The language PS-Algol[2] was designed to test these ideas. A PS-Algol program can open and operate on data in a database, modify it and add new data to it. There are functions to open and close the database and to *commit*, or write back changes. These same operations would be present in any database system. The difference with PS-Algol is that there are no other operations to read or write data. Instead objects are read in as they are required. The programmer can use any data structures appropriate to the task and the storage system will ensure that objects are brought into store. In this way persistent data are treated in exactly the same way as transient data and can, for example, be combined together in a single structure. Changes are made by the normal assignment operation but are only recorded in the database when the explicit *commit* is called. The action of *commit* is to preserve in the database every object which could be found by following pointers from a number of distinguished roots. Since these roots are the only way in which a program can get access to the data this rule ensures that all the useful data are preserved.

A persistent store can be thought of as a cross between a virtual memory system and a database. Objects in a database can refer to other objects and any object can only be reached by following these references from a few well-defined roots. Transfers to and from the database are usually made by calls to special procedures. In a virtual memory system pages are transfered to and from backing store without any explicit requests from the user. They are regarded simply as unstructured store which, if it can be retained in the filing system at all, must be retained as a whole. A persistent store combines these ideas by having automatic transfers but retaining only reachable objects in the database.

Because objects are read from the database transparently the system behaves very much like one where the whole workspace is read in and written out at the end. Opening a database is similar to reading in the workspace and *commit* to writing it out again. The difference is that the cost of reading a persistent database is dependent on the amount of data used and not the overall size of the database.

3 Poly and Standard ML

Poly[8] and Standard ML[9] are general purpose programming language supporting polymorphic operations. They are both statically type-checked and statically scoped and treat closures as first-class objects. Their type systems are different but the underlying abstract machines are sufficiently similar for a common implementation to be used.

They are used interactively and in the Poly/ML system the two languages together comprise a single system. The original implementation of Poly, and other implementations of ML, have used the workspace idea to preserve data from one session to the next. When the size of the workspace became large an alternative was needed and a persistence storage system, based on that for PS-Algol, was designed.

The initial design of the persistent store for Poly and ML was similar to the PS-Algol work. However as the design progressed it became clear that there were properties of Poly and ML and the way they are used, that would affect the design.

It was regarded as important that the system should be transparent to the user as far as possible. The user should not have to think in terms of a database but in terms of the programming language and his own data structures. The persistent store in PS-Algol went most of the way towards this by making transfers of data from the database into store transparent. As far as the user of Poly or ML is concerned the system is very much like using a core image which gets read in at the start of the session and can be written out when required. The main difference is that the initial prompt appears almost immediately but there is a delay when the first command is executed as the compiler is brought into store. At any time the user can call the *commit* function which will write changes back to the disc. Changes are also normally written back at the end of the session.

Both languages are strongly type-checked and make clear distinctions between values which can be updated, *variables* or *references*, and those which cannot. These features are exploited in two ways. Since the type system will prevent addresses being used except as references to objects the system can operate on the addresses without the user being aware. This would not be possible in a language which allowed the user to extract the address of a word inside an object, for example. Distinguishing updatable, or *mutable,* objects makes it is possible to mark them when they are created so that the system can keep them separate from the non-updatable, or *immutable,* objects. In practice the vast majority of objects are immutable so allowing some optimisations which would not otherwise be possible.

The Poly and ML compilers are written in Poly and are part of the system. All the compiled code they generate and all the other data structures are also within the system. Unlike in PS-Algol, where programs from outside can operate on the data and so there have to be ways into the data from outside, in Poly or ML the only operation needed is to start running the *read-eval-print* loop of either Poly or ML. The root of the database is therefore just a procedure which is called from the system when the session starts.

4 Multiple Users

The system described so far is for single users who have all their data in one database. Single user systems have the big advantage that all accesses are sequential and there is no need for any transaction mechanism. However the inability to share objects means that not only is a lot of disc space wasted with multiple copies of the same objects, but a user who produces something which is of use to others cannot pass it on. A mechanism for sharing data between users has been developed to reduce this problem.

4.1 Multiple Databases

One of the principles behind the design of the persistent storage system for Poly and ML was that it should be transparent to the users. What this implies for a shared data system is that the user should not be aware that some object is shared between several users, and that he should be able to use it as though he had exclusive use of it. In practice this is impossible to achieve without explicit control of concurrency so a compromise has to be made. In this system that is done by preventing shared variables from being updated, while allowing shared objects to be read.

The Poly system uses a scheme where each user has his own database which he can read or write. A user starts the session by executing the root procedure of his own database, the *root database* for that session. If a persistent address is used which refers to an object in another database it will be automatically opened, but only for reading. Calling the commit function causes variables read from the root database to be written back but other variables are not.

New databases are created by a special function which “spawns off” a new database with a new *root database.* Typically the root procedure will contain references to objects in the parent, and the parent will contain references to objects in the new database. In order to be able to follow these references it must be possible to find one database from the other, so each database contains a list of file-names of the databases to which it refers.

4.2 An example

This scheme has been used in a practical application for a theorem proving system, Isabelle[11], written in ML by L. Paulson. The basic Isabelle system is independent of any particular logic and is parameterised by a logic in order to be able to prove theorems using it. There are at present two logics for Isabelle, one using Martin-Löf type theory and the other using a set-based logic.

The system has been set up using the database heirarchy with a system database at the root. This contains the Poly and ML compilers and also standard types such as *int* and *list*. Below that is a database with the unparamaterised Isabelle system. Below the Isabelle system are two databases, one containing the type logic and the other the set logic. Finally there are the users' databases below these.

Each user still has their own database, but these are now relatively small since they contain only their own data. They can make use of declarations in the parents as though they were in their own database.

Since the parent databases are read-only they can be used in a distributed system. They are held on a shared virtual disc which can be mounted by users on different machines on a network who have their own databases on local discs.

4.3 Alternative Methods

This approach to sharing data is particularly suitable for large data structures which are read-only. It has a number of problems when writing to shared data. Perhaps the most serious is that the persistent store is no longer transparent since variables are only written back if they were read from the root database. The user can change variables read from other databases but those changes will not persist.

Another problem is the need for locking. One user may be using a database as their root database and periodically writing to it during a session, while another may be reading from it. The reader will have copies of values which have been read from the database in his own memory space and these may well be out of date. More seriously he will have copies of maps which give the disc addresses of pages on the disc. If the database has been updated several times it is quite possible that these maps will be out of date and the page at a particular disc address may be completely different to the one that was expected.

The only complete solution to implement a full transaction mechanism where objects are read in, modified and written out as a database transaction. As an object, or at least one which could be updated, is read from the database it would be locked so that it could not be read or changed by anyone else. The operation on it, either reading or writing, would be done, and the object would then be written back immediately and the lock released. This would be expensive since mutable objects cannot be held in store and every operation on them involves reading and writing to the database. An alternative would be to introduce the idea of a transaction into Poly and ML so that locking and unlocking would be done explicitly. This might reduce the cost since locking would be done at a higher level, however the user must now become aware of the transaction system. On the whole it was felt that this would be too complicated and the solution adopted was to lock the database as a whole.

5 Processes and Windows

As part of the basis of a programming environment a window package for Poly and ML has been developed. It provides the normal drawing operations for lines and text and for displaying bitmaps. It also allows for input from the keyboard or a mouse.

Handling activity on windows is most conveniently done by having a separate process manage each one. A process package was implemented for Poly and ML using ideas from CCS[10] and PFL[6]. This provides functions for forking processes and sending and receiving data on channels.

As with other objects in the Poly system processes are persistent, requiring the process state to be retained in the database. Most of the state is held in the process's stack and this can be written out to the database to save the process. A single stack segment is used for each process since it simplifies function calls but the segment is arranged so a small segment can be used initially and the segment will grow as required.

Persistent processes make possible a change in the way the database is written. Rather than having as the root of the persistent store a procedure which is called at the start of each session the *commit* operation now saves the list of active processes. Anything reachable from an active process is retained in the database. When the database is opened these processes are reactivated. The state is then the same as it was when *commit* was called. Normally there will be a process executing the read-eval-print loop for either Poly or ML.

Saving the list of active processes also allows the contents of windows to be saved, or more accurately, for them to be redrawn when the database is reopened in the same way that they appeared when it was committed. A window on the screen may need to be redrawn at any time because part of it has become exposed. When this happens the window package provides an event on the event stream of the window so that the program can redraw it. Because this can happen at any time there must be a process prepared to do the redrawing. The process can be saved and then reactivated in a new session. If it is now told that the window needs redrawing it will draw the contents exactly as they were in the last session. This is an active way of retaining data. Rather than save an image of the screen the process for drawing the data is retained.

References

[1] Atkinson M.P. et al. "An Approach to Persistent Programming". Computer J., Vol 26 No 4 1983.

[2] Atkinson M.P., Chisholm K.J. and Cockshott W.P. "PS-Algol: An Algol with a Persistent Heap." Technical Report CSR-94-81, Computer Science Dept., University of Edinburgh.

[3] Bobrow and Murphy. "Structure of a LISP System Using 2-Level Storage" Comm. ACM 10.3 March 1967.

[4] Challis M.P. "Data Consistency and Integrity in a Multi-User Environment" In Databases : improving usability and responsiveness. Academic Press. 1978.

[5] Falkoff A.D. and Iverson K.E. "The APL/360 Terminal System" Proc. ACM Symposium on Interactive Systems for Experimental Applied Maths.

[6] S. Holmström, *PFL: A Functional Language for Parallel Programming*, Proceedings of the Declarative Programming Workshop, University College London, 1983.

[7] Ingalls D.H.H. "The Smalltalk-76 Programming System – Design and Implementation" Proc. 5th ACM Symposium on Principles of Programming Languages. 1978.

[8] Matthews D.C.J. "Poly Manual" SIGPLAN Notices. Vol.20 No.9 Sept. 1985.

[9] Milner R. "A Proposal for Standard ML" in "Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming", Austin, Texas 1984.

[10] Milner R., *A Calculus for Communicating Systems*, Lecture Notes in Computer Science, no 81, Springer Verlag, 1981.

[11] Paulson L.C., *Natural Deduction as Higher-Order Resolution* J. Logic Programming 1986:3 237-258.