A Persistent Storage System for Poly and ML

David C.J. Matthews

Abstract

The conventional strategy for implementing interactive languages has been based on the use of a "workspace" or "core-image" which is read in at the start of a session and written out at the end. While this is satisfactory for small systems it is inefficient for large programs. This report describes how an idea originally invented to simplify database programming, the *persistent store,* was adapted to support program development in an interactive language.

Poly and ML are both semi-functional languages in the sense that they allow functions as first-class objects but they have variables (references) and use call-by-value semantics. Implementing such languages in a persistent store poses some problems but also allows optimisations which would not be possible if their type-systems did not apply certain constraints.

The basic system is designed for single-users but the problems of sharing data between users is discussed and an experimental system for allowing this is described.

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[Fal67]. 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[Fal67] 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[Bob67][Ing78] 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[Atk83]. 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[Atk81] 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[Mat85] and Standard ML[Mil84] 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 a **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 Implementation of Persistence

Reading in a single core-image, operating on it and writing it out at the end is fairly simple. The addresses in the image may have to be relocated but this need just involve adding a fixed offset.[1] The relocation is done all at once for the whole system. After relocation the addresses are the normal memory addresses for the machine so compiled code in the memory can operate on them. If however, we go to a form of paging where some objects may be in memory while others are on disc things become more complicated. If an object may or may not be in memory its address cannot be a simple memory address. An address must give, in some form, the location on disc where the object is to be found and it will have to be translated into the real address after the object has been loaded. In a virtual memory system this translation is done each time an address is used, and because it is supported by hardware or micro-code that is acceptably fast. However if we are using software to do the translation, calling a subroutine every time we used an address would be far too slow.

The solution used in PS-Algol and adopted in this system was to allow the two forms of the address, the disc or *persistent* address and the address of the object in memory, the *local* address, to co-exist in the memory. In principle, local addresses are used for objects actually in the memory and persistent addresses for those on the disc. Whenever an object is read into memory all the persistent references to it are changed into local addresses. In practice this would require a complete scan of the memory whenever an object was read in to find all the references, so instead we only overwrite a persistent address when it is actually used.

By choosing the local address to be the actual memory address of the machine we can run ordinary machine code programs. It is still necessary to look at an address before it is used to decide whether it must be translated, and call the translator if not. This could be reduced to a few instructions but if we wish to run machine code they would have to be compiled in every time an address was used to refer to an object, in case it was a persistent address. If we choose values for the persistent addresses which are invalid as memory addresses, and we can usually find some part of the address space which is illegal, we can make use of

[1] We assume that it is possible to distinguish the addresses and data in an object.

the machine to distinguish addresses for us. We compile in a normal instruction to use the address as though it were an ordinary local address. If the code is run on a local address it will run normally, but using a persistent address will result in an illegal address fault being generated by the hardware and operating system. If this fault can be passed back to the persistent storage system we may be able to overwrite the persistent address with its local equivalent. The instruction which made the trap can be restarted and we will be able to continue executing as though a local address had been used all along.

There are two trade-offs made here. The first is that having the two addresses coexist means that local addresses will have to be translated back into their persistent form when objects are written out. The second is that taking and catching an address fault is likely to be expensive since it involves calling the operating system. If a particular object is used only once then the cost of loading it and later writing it out is going to be expensive. However in the kind of applications for which this system is used most objects are used repeatedly once they have been read in and the average cost is much lower. A more serious problem is that we may have several copies of an address. If we use a persistent address it will be overwritten by its local form, but other copies will not be, and we will get faults if we try to use those. There are various strategies which are used to overcome this.

5 Constraints on the Code

There are at present implementations of the system for the VAX and the Sun (MC68020), and there is a portable version using an interpreted code where the persistent addresses are detected by the interpreter. The persistent storage system is the same in each implementation apart from a small section involved in decoding an instruction when a trap has occurred and in restarting the instruction. This is obviously machine dependent and will not be described in detail.

The system relies on being able to restart instructions after a persistent address has been overwritten by its local equivalent. An instruction may be part way through executing when the invalid address is detected, but in principle this should be no problem. The hardware of a machine which supports virtual memory must be able to restart any instruction after a page fault. On the VAX, where the internal state is backed-up by the machine to a point where the instruction can simply be re-executed this is true, but on the MC68020 a trap results in internal state being saved. This can be reloaded after a page fault under control of the operating system but is not available to a user process. This means that a general instruction cannot be restarted after a persistent store fault. However the code which is operating on the persistent store is generated by the Poly code-generator and it can be written to avoid instructions which would cause problems. Where this is not possible such instructions can be preceded by another instruction whose only function is to cause a persistent store trap on an address so that the next instruction will always have a local address to deal with. Adding these instructions means that code is slightly larger than it would be if it were working entirely on normal memory addresses but much less than if code had to be included explicitly to distinguish persistent addresses.

The other constraint on the code which may form part of the system is that garbage-collection may cause addresses to change. As well as being called when an object is created on the heap it may also be called when a persistent store fault occurs and space is needed for the object being read in. This may affect the code which can be generated.

6 Address Translation

A core-image is usually a contiguous file in which addresses correspond more-or-less to an offset in the file. We could use that for a persistent store and a persistent address would then be the offset of an object from the start of the file. When an object was written back to disc it would be written to the original place it was read from. Unfortunately if the machine or the program crashed during the writing process it might leave the file in a state where some objects had been written back and others had not. To avoid this problem we always write back blocks to areas of the disc which were not previously in use and never overwrite something useful. Since objects are written out to a different place from where they were read we need a map to give the current location of an object. The scheme is basically that suggested by Challis[Cha78].

We also need a map between persistent and local addresses so that we when we write an object out again we can translate the addresses in it back to their persistent form. It will also be used to translate other persistent addresses to an object which is already in store where there are several references to it. These maps both translate persistent addresses to either disc locations or local memory locations and for convenience they are combined into a single *address map*.

When a persistent address causes a fault we check in the map to see whether the object is already in store and if it is not we read it in. This will involve the operating system reading it from disc which it will almost certainly do by reading a disc block and extracting the bytes required. Since most objects are small there will be several in a disc block and it may be read several times to extract each object. Provided there is some locality of reference it may be better to extract all the objects from the block once the block has been read in. There is a problem with objects that straddle block boundaries so a better solution is to arrange that

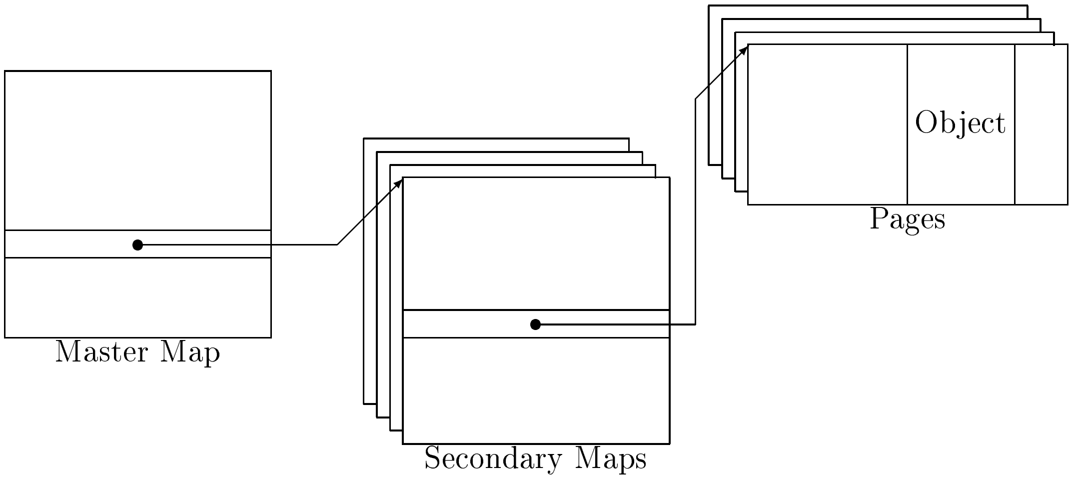

Figure 1: Map Structure

a set of objects always starts on a block boundary and waste a little space at the end of a block. If an object is larger than the block size we have to use more than one block for it. Since it will probably not be an exact multiple of the block size there will be spare space in the last block but this is left unused[2].

In practice we may not know the size of disc blocks on a particular machine so instead we assume a value for the *page* size which we hope will be a multiple of the block size. It is convenient to use a power of two so that the a persistent address can easily be separated into a page number and an offset. Grouping objects into pages also simplifies the address map. Since objects are always read in as a page the map need only give the address in memory or on disc of the page and not of each individual object.

Even though there is only one entry in the address map for each page it would still be fairly large if it were a single vector. Instead it is split into two levels with a master map giving the location of a set of secondary maps which in turn give the addresses of the pages 1. Translating a persistent address into a local address involves looking at the master map to find the secondary map, reading that into memory if it is not there and then reading the individual page if that is not in memory.

Using a two-level map like this has another advantage. When we come to write the memory back to the disc we first write the pages to new locations on the disc and then write back the secondary maps which give these new locations. These are also written to new locations on the disc and these are recorded in a new version of the master map. It is only when all of these are safely back on the disc

[2]This is because if we filled it up with small objects we might want to read in one of them before we read the large object, and end up with part of the large object read in.

that we write out the new master map. By following this procedure we have the database in a consistent state at all times. If a crash happens at any point before the master map is written, the database is in the state it was in before we started to write to it, because reading will always involve first finding the master map and then using that to find the secondary maps and the pages. Starting with the old master map will find the database in its old state, starting with the new master map will find it in the new state.

7 Optimisation

Unlike a conventional virtual memory system, the cost of dealing with a persistent address trap is considerable even if the object is already in store. The persistent store handler overwrites the address which caused the fault so there will not be another fault if that address is used again. That would suggest that there will be at most one fault for each copy of each address in all the objects read from the store. Unfortunately this does not work out in practice. Typically an address is loaded from store into a register and the register is used to find an object. The persistent store system will overwrite the copy in the register but the value in store will be unchanged. Repeating the process of loading and indirecting will cause another fault, and the only way to prevent it is to find and update the copy in store. The problem with that is that the cost of searching for it may be more than the cost of a few extra store faults.

Various methods are used to reduce the number of persistent store faults.

- When a new page is read in all the addresses which refer to objects already in store or in that page are updated.
- Clustering objects which refer to each other in pages will reduce the number of traps, since all the references between objects in the page will be converted when the page is read in. This relies on some degree of locality of reference and can be more or less successful depending on the data and the way objects are put together into pages.
- Pages previously brought in are periodically scanned for persistent addresses of objects which have since been brought in. The frequency of scanning has to be chosen with some care so that the time spent scanning is not greater than the saving in handling the faults.
- When a persistent store fault is taken other addresses "nearby" can be updated. Which addresses to look at depends very much on the structure of the code, but the savings can be considerable. For example the contents of

all the registers can be updated, but the success of this will depend on the extent to which values are cached in registers.

8 Writing Back

At some point the user will want to write the changes that have been made in local memory back to the file. This basically involves a reverse of the process used when the objects were read in. The addresses in objects to be written out are converted back to their original persistent form and the pages are written, as described above, to new locations on the disc.

It is only necessary to write back objects which have changed and these will be a small proportion of the total. It would be possible to find out precisely which pages had changed by comparing the new version with the page on disc, but it is sufficient to assume that any of the pages of mutable objects may have changed, since there are not many of these.

Each address in an object to be written back must be converted to a persistent address. The maps are set up to make conversion of addresses from persistent to local form easy, but conversion the other way has to be done by searching through the maps for a local address. This can be made reasonably efficiently by having a simple cache of recently found local addresses and their persistent equivalent. Each entry in the cache refers to a page rather than an individual object in it so quite a small cache can be used.

Mutable objects are used for variables and arrays which can be updated. If a variable is set to refer to an object read from the file we will be able to convert the address back to the persistent store. However it is equally possible to create a new object in local memory and assign the address of that to the variable. This is what happens when new declarations are made in Poly or ML. Before the variable can be written back there must be a persistent address for the new object, so it must be turned into a persistent object, along with all the objects it refers to. The local objects are copied into pages by a process very similar to a copying garbage-collection. The addresses of the pages are put into free entries in the secondary maps and new secondary maps are made if there no free entries. It is possible to distinguish mutable and immutable objects by a flag bit so the two kinds of objects can be put into different pages.

Once the copies of the pages in store have been converted to the persistent form they can be written to the disc. They are written to previously unused blocks on the disc and to do this there is a bit map with a bit corresponding to each block. At the end of the writing process a new bit map is written out with the new locations of the pages shown as allocated and their previous locations now free.

In PS-Algol the system may write objects to the disc before *commit* is called when the memory becomes too full. In many database applications many different objects are used, each for only a short time. In these applications the memory would quickly fill up with objects which were no longer required so PS-Algol has to have a mechanism to cope with this. In the Poly and ML system however objects, particularly procedures, tend to be used repeatedly or not at all. So far it has not been found necessary to clear objects out of the memory.

9 Garbage Collection

In normal operation once an object has been written out to the database it remains on disc even if it becomes unreachable. There is, however, a garbage collector which can be run periodically on the database to recover the lost space.

The garbage collector can be run in one of two modes, either non-compacting in which case it merely makes a new bit map of free blocks, or compacting when blocks at the end of the file are copied to free space nearer the beginning. In both cases the basic principle is the same. The garbage collector starts from the root of the database and finds all the objects that are accessible from it. Any page containing an accessible object is retained and the rest are garbage. The bit map and the entries in the maps which show the position of pages on disc are modified so that the garbage blocks on disc and their persistent addresses can be re-used. Working on whole pages and not compacting objects within a page means that persistent addresses do not change but it could lead to fragmentation over long periods.

The garbage collection process is basically the same as a mark phase of an in-store garbage collector but with one crucial difference. An in-store garbage-collector can usually assume that the cost of reading a word in memory is independent of its location whereas when garbage-collecting a disc the cost depends on whether the word is in a block which is in store or not. A simple recursive marking phase might result in a lot of disc activity. A better scheme is one where as many objects as possible in a block are processed together. This will reduce the number of times a particular block is read in.

The normal recursive scheme follows the addresses in an object as soon as they encountered, unless they refer to objects which have already been dealt with. There is one bit, the *mark bit*, which indicates that the object has been processed. It is used both by the marking phase itself to prevent an object being scanned more than once, and also as the result of the marking phase to indicate that the object is not garbage and must be preserved. The mark bits may be held with the object or in a separate bit map.

The scheme used in the disc block garbage-collector is a variation of this but uses a second bit map, the *pending bit*, to indicate that an address has been found but not followed. This bit is set when the recursive algorithm would recursively process the object referred to. Using two bit maps allows the garbage-collector to work iteratively rather than recursively. Initially the pending bits corresponding to the root procedure of the database are set. The garbage-collector then reads a block which contains at least one object referred to by a pending bit, and ideally having several pending objects. Each of the pending objects in that block are now marked and removed from the pending map. All the addresses in these objects are added to the pending map, unless they refer to objects that have already been marked. Some of these addresses may be in the current block, especially if there is some locality of reference, and they can be processed now. When there are no pending objects left in that block another block is read in. This repeats until the set of objects to be scanned is empty. The blocks which were read in this process must be kept, the rest are garbage.

10 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.

10.1 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.

10.2 Implementation

It is implemented by using part of a persistent address as a *file number.* The address now consists of file number, page number within that file and an offset in the page. The file number is an index into a table of file names which can be found from the master map. Translating a persistent address may now involve opening a new file if the address refers to an object in a file which has so far not been used. It is necessary to modify the persistent addresses in objects read from other files because the file-number fields will index into the file name table of that file and will have to be changed into an index into the file name table of the current file.

10.3 Alternative Methods

This approach to sharing data has a number of problems. 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.

11 A Persistent Environment

The persistent Poly system has as its root a procedure which is called when the system starts. The root procedure calls through a procedure variable so that new procedures can be installed. This allows the user to decide, for example, whether to enter Poly or ML at the start of the session.

When using either Poly or ML the root procedure enters the read-eval-print loop for the language. This reads input from the terminal and sends it to the compiler. The compilers are pure functions operating on input and output streams and taking an *environment* as a parameter to maintain the state. In Poly the environment is a pair of procedures, one of which takes a string and returns a value, the other enters a string/value pair into the table. For ML the environment is rather more complicated because the name spaces for values, types, exceptions and infix status are distinct. Environments may be implemented in any way but a hash-table is convenient. The environment given to the compiler by the root procedure is part of that procedure's closure. During a session declaration of objects (procedure, values or types) made by the user are added to this table. When the session ends and the data are written back to the database the modified environment is written back as well. Hence the next session is run using the new environment and all the declarations made during the previous session are available.

An environment which maps names onto objects is very similar to a directory in a filing system so the environment system can be used as a form of typed filing system. The objects themselves are basically pairs of a value and information about its type. There is no reason why there should not be many environments available, some contained in others. This would correspond to a filing system which allows arbitrary directory structures to be built up. There is no requirement that the system of environments be a simple tree structure, an environment could contain a reference to itself or an environment pointing to it. Since the user can write his own environment in any way he likes he can incorporate any access controls he feels desirable. For example the environment could be written so that a password must be given before a function will return the environment.

12 Conclusions

Adding a persistent store system to Poly and ML was a fairly simple exercise, though work was needed to get it running efficiently. It would generally be the case that a persistent storage system as described could be added to any language in which all objects reside in the same memory space and are garbage-collected. It is also necessary for instructions to be restartable if the technique of using persistent addresses which cause memory traps is used.

The advantages of a persistent storage system over other methods of storing data are worth noting. Reading and writing the whole of a large core image is expensive and the overall size may be limited by the memory, real or virtual, of the machine. It is also impractical to explicitly read and write large numbers of objects.

Finally, the system has been in use for some time, and the ML implementation in particular is being used for developing large proof systems.

References

[Atk83] Atkinson M.P. et al. "An Approach to Persistent Programming". Computer J., Vol 26 No 4 1983.

[Atk81] 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.

[Bob67] Bobrow and Murphy. "Structure of a LISP System Using 2-Level Storage" Comm. ACM 10.3 March 1967.

[Cha78] Challis M.P. "Data Consistency and Integrity in a Multi-User Environment" In Databases : improving usability and responsiveness. Academic Press. 1978.

[Fal67] Falkoff A.D. and Iverson K.E. "The APL/360 Terminal System" Proc. ACM Symposium on Interactive Systems for Experimental Applied Maths.

[Ing78] Ingalls D.H.H. "The Smalltalk-76 Programming System – Design and Implementation" Proc. 5th ACM Symposium on Principles of Programming Languages. 1978.

[Mat85] Matthews D.C.J. "Poly Manual" SIGPLAN Notices. Vol.20 No.9 Sept. 1985.

[Mil84] Milner R. "A Proposal for Standard ML" in "Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming", Austin, Texas 1984.