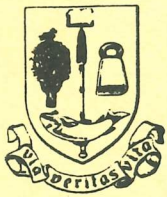
University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ



University of St. Andrews
Department of Computational Science
North Haugh
St Andrews KY16 9SS

Polymorphic Persistent Processes

Persistent Programming
Research Report 39
June 1987

A Dearle.

Polymorphic Persistent Processes

R.Morrison, C.J.Barter[++], A.L.Brown, R.Carrick,
R.Connor, A.Dearle, A.J.Hurst[+] & M.J.Livesey

Department of Computational Science, University of St Andrews,
North Haugh, St Andrews, Scotland KY16 9SS
Tel 0334 76161

[+] Department of Computer Science, Australian National University,
G.P.O. Box 4, Canberra, ACT 2601, Australia
Tel 062 49 5111

[++] Department of Computer Science, University of Adelaide,
Box 498, G.P.O., Adelaide, South Australia 5001
Tel 08 228 5333

Abstract

The problems of shared access to large bodies of information raise difficulties in the understanding and semantics of concurrency, distribution and stability. When the information is held in a persistent object store, the problems of understanding are extended to the interaction of the concepts of persistence and store with those above.

In this paper, we identify these difficulties and propose a model of concurrency which is integrated with a polymorphic type system. Such integration allows polymorphic, persistent processes, the advantage of which are discussed.

1. Introduction

In our attempts to design and build a persistent information space architecture (PISA) [1] we have identified a number of interacting and sometimes conflicting problems with regard to persistence, stores, concurrency, distribution, transactions and stability. Some of the difficulty is in deciding at what level the persistent architecture operates, be it a hardware or software architecture. Other difficulities arise in the complexity of the problems of concurrency. In this paper we identify these difficulties and clarify them. We propose a model of concurrency that may be used as solutions to the problems. It is based on the rendezvous of Ada and integrated with a polymorphic type system. Such integration allows us polymorphic, persistent processes.

2. A Persistent Information Space Architecture

2.1 Persistence

We have defined the persistence of data to be the length of time for which the data exists and is useable[2]. In a persistent system the use of all data is independent of its persistence. Here we extend this notion of persistence to abstract over all the physical attributes of data, for example where data is kept, how long it is kept and in what form it is kept. We have discussed the advantages of persistence elsewhere [3,13,14] and will not labour them here. It is sufficient to say that by ensuring that all data objects are persistent and that the persistence of data is invisible to the programmer, then this level of abstraction yields powerful software engineering gains in the life of large systems. The figure often quoted is 30% of the total cost of a system throughout its life cycle[2].

At this level of abstraction all physical properties of the data are invisible to the user since persistence is a concept that abstracts over them. It is important to distinguish between the conceptual and physical problems of building and using persistent spaces in order to identify the areas on which we need to concentrate to achieve true persistence. In the following discussion we will pursue this theme, separating the conceptual or logical properties of the persistent information space from the physical ones. This is not always an easy task especially with regard to concurrency.

We wish to build a total system capable of providing for all programming activity. Our traditional view of the persistent information space is that it will subsume the functions of a plethora of mechanisms currently supported by components such as command languages, editors, file systems, compilers and interpreters, linkage editors and binders, debuggers, DBMS sublanguages and graphics libraries[1]. The information space is composed of objects, which may be simple or highly structured, defined by the universe of discourse of the type system of the PISA architecture. Without prejudging the universe of discourse of the language used to program the information space or indeed the style of

language itself, we are left with the conceptual requirement that the space must be unbounded. That is, the user has the facility to create persistent objects forever. Most modern programming languages, database systems and operating systems provide this facility and if we are to unify these mechanisms then the persistent space must also provide it.

Our previous work[2] has shown that one of the main difficulties in using an unbounded space is in remembering an unbounded number of names. Traditional solutions to this problem have included block structure in programming languages, hierarchal file directories in operating systems and data dictionaries in database systems. None of these solutions are totally satisfactory and often other mechanisms, such as those for module construction to augment block structure in programming languages, have arisen. Mechanisms for controlling and using this unbounded name space must be made available to the user.

We define context to be the manner in which the persistent information space controls the unbounded number of names but will defer discussion of how context may be defined in the architecture language until later. The important point is that we have identified context as a conceptual requirement of using the persistent information space.

2.2 Stores

In all our work so far the persistent information space has been built in the form of a store[5,7]. This is not entirely necessary but languages without store semantics, such as the applicative languages, can be accommodated as trivial subsets of the store semantics languages as far as persistence is concerned.

A store in the denotational semantics sense is a mapping from L-values to R-values. This definition says nothing about other desirable properties of a persistent store so we will add some of our own. These are

a an unbounded capacity to store objects
b infinite speed
c stability

There are, of course, a number of problems in building and using a persistent store with the properties above. The conceptual problems of persistent stores arise out of how the store is used and the physical problems arise from the engineering difficulties in building the store. We will look at each of the requirements on the persistent store given above to identify these conceptual and physical problems.

2.2.1 Unbounded capacity to store objects

The need for unbounded capacity comes from the unbounded nature of the modelling performed by the programmer in the persistent store. The conceptual problems of using an unbounded store are the same as for an unbounded persistent space. That is, we must contend with an unbounded number of storage identifiers.

One approach to this problem is to take the Ethernet solution. That is, all objects in the universe have a unique storage identifier. Most Ethernet systems do not actually run such a regime. Usually only rather large components of the universe, like machines, have unique addresses and smaller components are addressed contextually. This sensibly reduces the need for large addresses for all data objects. The great problem with the Ethernet solution occurs when machine addresses are accidently duplicated, as practicioners in the field will know happens every day, or when another Ethernet system is shipped in from outer space to be connected to an existing one, a less likely occurance.

The Ethernet solution gives us a large, potentially infinite, flat address space which is the model we wish to appeal to. Without any context mechanism it is equivalent to a telephone system where every subscriber is listed in the same large telephone directory. The telephone system analogy is a good one since in practice we do not use a large single directory but navigate around the world's telephone system by a collection of local directories. That is, a context mechanism.

Of course we cannot build a store with an infinite capacity. Any attempt to give the illusion of such a store will be built out of components, which may be of disparate technologies, and may include software technologies such as stacks and garbage collectors to reclaim unused space. This is, however, an engineering constraint and therefore a physical view of the store although it does introduce the notions of locality and distribution which cannot be seen at the persistent space level.

A data object can be resident in a local store or in a distributed store. Two data objects reside in the same locality if they live in the same physical store and are distributed from one another otherwise. By combining locality and distribution we can compose very large stores. Note, however, that locality and context are not necessarily equivalent since a context can easily spread over a number of localities or indeed a locality can contain many whole or partial contexts. Only when an object in the contextual name space is mapped onto a locality in the physical store does context and locality coincide. Context is a property of the conceptual space whereas locality a property of the physical store.

2.2.2 Infinite speed

An advantage of an infinitely fast store is that it can be operated sequentially since there is no speed advantage in operating it concurrently. This makes the semantics of the store easier to understand. A second advantage is that the programmer need never take

account of where information is stored. Multiple copies of objects for speed trade offs are unnecessary since there is no speed advantage to moving the object closer to its point of use. Thus the object may always reside in the one place.

Again we cannot build this infinitely fast store. The only way we know of approaching this is to duplicate components and make them operate concurrently. This again gives rise to locality and distribution but also adds a new dimension at the physical level - that of concurrency. This form of concurrency, for speed advantage, is an engineering decision and not fundamental to the operation of the persistent store.

2.2.3 Stability

All users of the persistent store would wish it to be stable. This mechanism ensures that data is always kept (or copied) on non volatile storage devices. Thus in the event of a system failure no data will be lost. This, of course, is only an illusion since no system can guarantee that even the non volitile devices are free from corruption by malice or error. Usually, however, an adequate level of reliability can be provided for any system.

Stability is a property of the physical store medium and is therefore a physical property of the information space and hidden from the user. In some systems stability and transactions are synonomous leading to some confusion of how the concept of stability arose.

From the above discussion we can see that the main conceptual problem that we have in using the persistent information space is that of context. That is, how do we partition the name space in order that we can master the complexity of a potentially unbounded number of names. The physical problems centre around how to build an infinite stable store. The issues of locality and distribution, that is where an object lives, allow us to simulate unbounded capacity out of smaller components. At this level, concurrency allows us to simulate higher speed out of slower components and stability can be simulated by a number of techniques such as incremental and total dumping.

2.3 Concurrency

We have argued above that a major motivation for concurrent activity is execution speed. The need for concurrency increases as machines approach their theoretical speed limit at the same time as the complexity of the applications becomes great enough to require even greater power.

There is, however, a second major need for concurrency. Many of the activities that we wish to model are inherently parallel. For example, in a model of a supermarket there will be many customers and shop assistants working autonomously and in parallel. If we wish to capture the essential nature of this real world activity then we require language

primitives powerful enough to model it.

One of the major breakthroughs in the design and understanding of operating systems was gained by modelling the system as a set of co-operating sequential processes[8]. Since most of the early operating systems modelled in this manner ran on uni-processor machines this modelling was not done to simulate infinite speed. It was done to simplfy the complexity of the system being built in order to gain greater insight into its operation. This method of modelling, first applied to operating systems, has now been applied to database systems, graphics systems and general problems in computer science[9]. It yields a new style of program construction and understanding and therefore can no longer be regarded as a physical property of the store.

In order to accommodate this wish to model using concurrency, a host of languages have been invented or proposed that include the notion as part of their universe of discourse. Thus concurrency in the persistent information space sense is not a conceptual requirement of the space but of the manner in which we wish to model. This is equivalent to deciding as to whether we wish to use other data objects such as arrays or functions to model with. However, since we wish to unify the activity of operating systems and database systems with our information space it would be wise to have concurrency as a conceptual requirement of the language or languages supporting the space.

We therefore make concurrency a conceptual requirement of thePISA architecture rather than the information space itself which we traditionally view as figure 1.

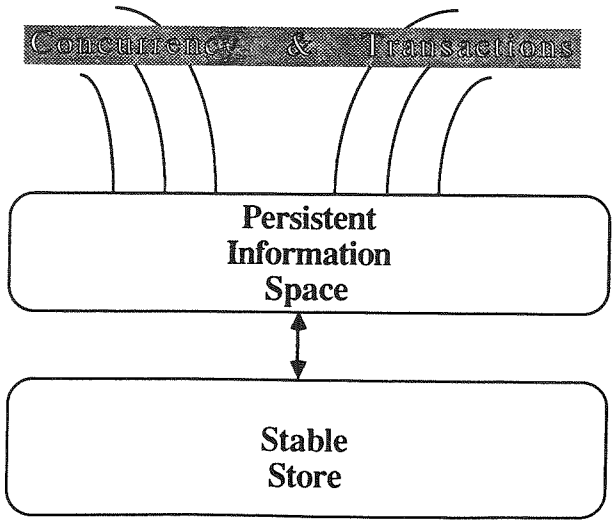


Figure 1 A Persistent Information Space Architecture

The persistent space is composed of objects defined by the universe of discourse of the PISA language. For the present the only requirement that we have of that language is that it must support concurrent computation and an unbounded name space. At this level the programmer has no notion of where the data resides, be it locally, on disk or on a remote processor, how long the data may be kept or in what form it may be stored.

At a lower level the information space is supported by a stable store. This store may be distributed over many storage devices and machines and is likely to be built out of many disparate technologies both software and hardware.

The focus of this paper is to look at how concurrency may be integrated with the persistent information space. In particular what primitives are necessary to support concurrency in the language and at what cost do we include them? For example, if we promote concurrency to being a conceptual requirement of the system what must we bring with it? Are we forced to accept distribution as a conceptual requirement and what then of locality and stablility?

3. Concurrency Models

In designing a concurrent language for use in programming the persistent store we have two conceptual problems. The first is how to impose context on the unbounded name space. The second is how to specify concurrent activity, that is separate threads of control.

There are many different styles of concurrency in modern programming languages. The applicative languages such as SASL[15] have implicit concurrency due to the fact that they are referentially transparent. This style of concurrency is transparent to the user and will merely add speed to the execution of the programs. It is difficult to see how the applicative languages can make full use of a persistent store since the store would have to remain static to ensure referential transparency.

Store semantics languages have themselves split into two paradigms. The first is a shared store semantics where the whole store is available for use by all customers. The use must be synchronised to avoid indeterminate results. The shared store model roughly characterises a multiprocessor system where many processors share the same store.

The second paradigm is a message passing paradigm where independent tasks have their own store and communicate with other tasks by sending messages to them. The message passing paradigm roughly models a distributed system where separate processors, perhaps with their own local store communicate over a communications channel.

It is generally accepted that where large amounts of communication are required then the shared store model is more efficient in speed since local store is usually faster than a

communications channel. However, where large amounts of computation are performed between communications then the message passing model may be more appropriate.

Our dilemma should now be apparent. For persistence we wish to appeal to the large flat store model on which we can impose some context mechanism. Thus the shared store model would seem more appropriate. On the other hand we know that this unbounded information space will be constructed out of components and it would seem sensible to build this in from the beginning to allow for expansion. For this the message passing model is more appropriate.

The answer is to have a model that will allow the programmer the freedom to choose.

3.1 Napier Model

We have proposed the language Napier in which the persistent store may be regarded as an unbounded collection of objects, each one sharable among the active processes in the system. We will use this language to demonstrate the concepts necessary to allow polymorphic, persistent processes.

In Napier, all data is persistent. That is, data is kept for as long as it is useable. This can be determined from the fact that it is reachable by the computation of the transitive closure of objects from the persistence root, called PS. When a process terminates all its data objects may be destroyed except those that the process has arranged to be reachable from PS. It should be noted that the persistent store will in general be a graph since it is a generalised data structure and it may be distributed over many machines. Given such a model of the information space we must define mechanisms for context and concurrency.

3.1.1 Context

Context is controlled by two methods in Napier, one static and one dynamic. The block structure of the language allows objects to be hidden to the outside world. Thus within the context of the block the name has a unique interpretation. Block structure forms a tree of contexts which may be used to segment the unbounded information space. The space, however, needs to be a graph and not a tree and that would suggest that block structure is not powerful enough to model all the required contexts.

In most programming languages it is recognised that block structure is not sufficient for all our modelling needs. Recursive data structures with references that outlive the block in which they were created are a common solution to this problem.

In languages with higher order functions, such as Napier, we can overcome this same difficulty by another method. For example, we could write a block that returned a procedure which held within its closure a hidden object. A random number generator is a good example of this.


```
let random = begin
                let rand := maxint div 2
                proc ( -> int)
                begin
                    rand := rand div (maxint -1)
                    rand
                end
             end
```

The block expression when executed, returns a procedure which contains the actual random number, 'rand', in its closure. 'rand' is initialised in the block expression. When we exit the block the object 'rand' cannot be destroyed as it will be required if the procedure random is called. Languages with such semantics are called block retention languages and any block structured language with higher order functions falls into this category. By adding some concurrency control to the system we could ensure mutual exclusion on the access to rand and extend the block to a monitor[10].

Although the above method allows users to dynamically create and manipulate contexts, the technique is essentially static since the scope of the objects must be defined by the programmer and may never be changed. A second method of context control is provided in Napier. The technique is similar to block structure except that we are allowed to dynamically nest the blocks. To do this the data type environment is used.

Objects of type environment are collections of bindings, that is name-value pairs. The distinguished point of the persistence graph, PS is of type **env**. Objects of type **env** belong to the infinite union of all cross products of named-value pairs. They differ from structures in this and by the fact that we can dynamically add bindings to objects of type **env**. This is perhaps best shown by example. We will create an environment that contains a counter and two procedures, one to increment the counter and one to decrement it. This may be done by

```
let e = environment ()
let count := 0 in e
!we have now placed the binding count : int = 0 in the environment e
use e with count : int in
begin
        let add = proc (n : int -> int) ; {count := count + n ; count} in e
        let subtract = proc (n : int -> int) ; {count := count - n ; count} in e
end
!the environment now has three bindings, count, add and subtract
```

The **use** clause binds an environment and its field names to the clause following the **in**. In the above the name count is available in the block as if it had been declared in the immediate enclosing block. The binding occurs at run time since in general the environment value, 'e', may be an expression evaluating to an environment. The binding is

therefore dynamic and is similar to projection out of a union. The difference is that here we only require a partial match on the fields and other fields not mentioned in the **use** clause are invisible in the qualified clause and may not be used.

The environment mechanism provides a contextual naming scheme that can be composed dynamically. The **use** clauses can be nested and the environments involved calculated dynamically and therefore the name bindings can be constructed dynamically. This does not yield full dynamic scoping in the Lisp sense since all the objects in the individual environments are statically bound. The technique complements the block structure in the language and completes the context mechanisms required for persistent information spaces.

For information to outlive the process it must be reachable from the distinguished root, PS. In the above case we could do this by adding e to PS by

```
let ee = e in PS
!add the binding ee : env = e to the environment PS
```

and to retrieve it again we could use

```
use PS with ee : env in ...
```

Having defined a mechanism for the conceptual problem of context we must now define one for concurrency.

3.1.2 Concurrency

The model of concurrency in Napier is based on CSP [9] and Ada [11]. Processes are a type in the language and many instances of processes of the same type may be created. Two processes are equal only if they are the same process. They have the same type if they have the same entries. A process may be specified by the following syntax

```
<process_type>    ::=  process (<parameter_list>)[with<entry_list>]<clause>
<entry_list>      ::=  <identifier_list>:<proc_type>[,<entry_list>]
```

Thus the example of a counter given earlier may be extended to a safely updated counter by specifying the type

```
type shared_counter is process (init : int) with add,  subtract : proc (int ->int)
                <clause>
```


where the clause specifies the synchronization of the operations 'add' and 'subtract'. For processes that do not offer any services the entry list may be empty. Inside the process there is only one locus of control and therefore only one entry maybe active at any one time. To receive entries the process must execute a **receive** clause which is defined as follows

```
<entry_clause>    ::=  receive<identifier>([<parameter_list>])[do<clause>]
```

In the receiving process, the protocol for communication is wait for sender and the rendezvous is only active during the execution of the entry clause. Thus very short synchronization can be achieved when there is no **do** part of the entry clause.

Non-determinism is provided by the **select** clause. The syntax is

```
<select_clause>      ::=  select<guarded_command>
                          [or<guarded_command>]*
                          default : <clause>
<guarded_command>    ::=  [<guard>] : <guard_clause>
<guard_clause>       ::=  <entry_clause> | <clause>
<guard>              ::=  <boolean_clause>
```

The **select** clause evaluates all the guards in the **or** options. An option is open if it has no guard or the guard is true. One open option is chosen for execution non-deterministically. If none of the options is open then the **default** option is selected. Thus the shared counter example could be written as

```
type shared_counter is process (init : int) with add,  subtract : proc (int -> int)
begin
        let i := init
        while true do
                select
                        :       receive add (val : int) do { i := i + val ; i }
                or      :       receive subtract (val : int) do { i := i - val ; i }
                default {}
        end
end
```

This specifies the type 'shared_counter' which is a process requiring an integer parameter on creation. Processes of this type have two entries 'add' and 'subtract' which are used to call services in the process. The process will loop forever receiving requests to 'add' and 'subtract' in any order. The critical sections are governed by the **do** clauses.

A process of this type may be created by

```
let counter = shared_counter (0)          !create a new process running in parallel
```

Thus 'counter' is a handle on a process of type 'shared_counter' which is executing in parallel. The process 'counter' will run forever but may be garbage collected when it is no longer possible to communicate with it. To the creator process, the entries of the created process act like procedures. Calling one of the entries establishes a remote procedure call with the protocol of wait for reply.

For convenience we can rename the entry procedures. For example

```
let Add = counter (add) ; let Subtract = counter (subtract)
```

and we can pass the procedures 'Add' and 'Subtract' to other components of the system.

We may wish to establish a rendezvous by calling the procedure. For example

```
let answer = Add (2)
```

The process itself will ensure mutual exclusion of multiple calls.

3.1.3 Locality and Distribution

So far all the processes that we have described are lightweight in that they share the address space of their creators. It so happens that in the 'shared_counter' example, the processes do not use any free variables and create their own environment, thus making it look heavyweight in nature. This we can use to our advantage to give both lightweight and heavyweight processes in the persistent store.

Our ideal model of the persistent architecture is a large flat object space fragmented by the context mechanism. We know, however, that the system will be built out of localities and we can accommodate this by arranging that localities are always controlled by one context. By making the context mechanism the environment, we can achieve the correct mixture of bindings necessary to support a distributed system.

To support the distribution mechanism we require two procedures to be built into the system. They are

```
let copy = proc [t : type] (item : t -> t )
let copy_to_env = proc [t : type] (environment : env ; N : name[t] ; item : t)
```

The copy procedure makes a 'copy' of the object. That is, it copies the transitive closure of the object to ensure that it will work correctly. The 'copy_to_env' procedure makes a copy of the object using the copy procedure and moves it into the same locality as

the environment placing a new binding in the environment. The new binding is the name 'N' to the copy of the value 'item'.

The 'copy_to_env' procedure is essentially the bootstrap mechanism for a new locality being added to the persistent store. Initially there is one distinguished point PS. To add a new locality, a binding is placed in PS or any environment reachable from PS, so that it may be seen by all users. Binding to this new environment, and thus locality, is performed by the environment binding mechanism described above. The placing of the new locality in the environment is equivalent to plugging in a new component to the system and must be performed by a low level implementation as it is not possible at this level of abstraction.

3.1.4 Stability

Each locality is stabilized independently in the system. When the standard procedure 'stabilize' is called the locality in which the processes is operating is stabilized. This may be done automatically or by the user. The 'stabilize' procedure calculates which objects are in the locality that it is going to operate on by computing the transitive closure of all the objects local to the environment. Cross locality pointers are ignored. The procedure is made available to users so that higher level transaction mechanisms may be built. Krablin[12] has shown how this may be done in the language CPS-algol.

3.1.5 Polymorphic processes

In the following example we demonstrate that by integrating the process concept with the type system, we can define polymorphic persistent processes. The example is that of readers and writers accessing a shared database. The procedure that creates the database is given an initial value for the database, which is copied to remove any aliases, along with the database type. The procedure creates a process to contol access to the database and returns the procedures 'read' and 'write', in a structure, which may be used to access the database in a controlled manner. Indeed since the database is always copied it is the only manner in which it may be accessed. The example can be extended to allow alteration to parts of the database but that is not relevent here. The algorithm is taken from Barnes [4] page 228.

```
let Readers_Writers = proc [t : type] (init : t
                -> structure (Read : proc ( -> t ) ; Write : proc (t) ) )
begin
        type Control is process () with  start : proc (int),
                                         stop_read, write_it, stop_write : proc ()
        begin
                let readers := 0 ; let writers := 0 ; let READ = 0 ; let WRITE = 1
                while true do
                begin
```

```
                select
                        writers = 0   : receive start (service : int) do
                                if service = READ then readers := readers + 1
                                                  else writers := 1
                or                    : receive stop_read () ; readers := readers - 1
                or      readers = 0   : receive write_it ()
                or                    : receive stop_write () ; writers := 0
                default { }
        end
end

let control = Control () ; let item := copy (init)

struct (        Read <- proc ( -> t)
                        begin
                                control (start) (READ)
                                let result = copy (item)
                                control (stop_read) ()
                                result
                        end,
                Write <- proc (X : t)
                        begin
                                control (start) (WRITE)
                                control (write_it) ()
                                item := copy (X)
                                control (stop_write) ()
                        end )
end

let synchronised_integer = Readers_Writers (0)
let Read = synchronized_integer (Read) ; let Write = synchronized_integer (Write)
```

In this example, with the call Readers_Writer(0), the user process creates a database which is a synchronised integer with initial value zero. The two database operations are given local names 'Read' and 'Write', which will give controlled access to the integer.

While the process itself ensures mutual exclusion of access to its entries, the procedures 'Read' and 'Write' may be made available to any number of other (user) process, to provide concurrent interfaces to the database. Because such procedures execute within the closure of their creator (in this case the procedure 'Readers_Writer'), care must be taken with access to shared variables in this closure; in this example, only 'item' is shared, and access to it is controlled by the process 'Control'.

3.1.6 Protocals and inheritence

The entry list for a process specifies its type and can be considered as the protocol through which it may be accessed. By utilising the multiple inheritence scheme of Cardelli [6] we can place process types in the type lattice and define a partial ordering of processes. Thus it is possible to define procedures that will operate on processes with at least a given defined protocol. If the process has a more specialised type then that may also be used. For example


```
type RON is process (init : int) with add : proc (int -> int) <clause>
type FRED is process (init : int) with add, subtract : proc (int -> int) <clause>

let Ric = proc [t : type ≤ Ron] (A : t )
        begin
                .
                let q = A (add (2))
                .
        end

let ron = RON (0)                       ! create a process of type RON
Ric [RON] (ron)                         ! pass it to the procedure Ric
let fred = FRED (1) ; Ric [FRED] (fred)
```

The procedure 'Ric' takes as a parameter an object of type 't' which is a process with at least the entry 'add'. In the example, the procedure is called twice with a process parameter. The first 'ron' has exactly the entry 'add' whereas the second 'fred' has more. Inside the procedure, only the entry 'add' may be used. By using this subtype inheritence we can abstract over entry protocols that are common to processes.

4. Conclusions

We have presented a model of a persistent information space which is composed of objects. The information space is unbounded in that the user has the capacity to create objects forever. A context mechanism is introducted in order to control the unbounded name space.

It is argued that at a persistent level of architecture, concepts such as locality, distribution and stability are physical properties of how we might build an unbounded information space. By using one of the contextual mechanisms of the system to incorporate locality we can provide all the necessary functionality by two procedures 'copy_to_env' and 'stabilize'.

The integration of the concept of a process with a polymorphic type system allows us to write polymorphic descriptions of processes. Since these are then naturally persistent we have polymorphic, persistent processes.

5. Acknowledgements

This paper is the result of an intensive 5 day study group held in St Andrews during

the study leave periods of Chris Barter and John Hurst and during a subsequent visit by Ron Morrison to Adelaide. Although the views in the paper do not accurately record all the contributions of the participants, they are all listed as authors. We see this as a fitting conclusion to the exercise. We are, of course also grateful to our collaborators in the PISA project, particularily Francis Wai and Malcolm Atkinson at Glasgow University who are also working on these problems and with whom we have shared many ideas. The work was supported by SERC grants GR/D 4326.6, GR/D 47790 and GR/D 8823.

6. References

1. Atkinson, M.P., Morrison, R. & Pratten, G.D.
Designing a persistent information space architecture. 10th IFIP World Congress, Dublin (September 1986),115-120. North-Holland, Amsterdam.
2. Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
An approach to persistent programming. Computer Journal 26,4 (November 1983),360-365.
3. Atkinson, M.P. & Morrison, R.
Procedures as persistent data objects. ACM.TOPLAS 7,4 (October 1985),539-559.
4. Barnes, J.G.P.
Programming in Ada. 2nd Edition. Addison-Wesley (1984).
5. Brown, A.L. & Cockshott, W.P.
The CPOMS reference manual. The Universities of Glasgow and St Andrews PPRR-13. (1985).
6. Cardelli, L.
A semantics of multiple inheritence. In **Lecture Notes in Computer Science**. 173, 51-67. Springer-Verlag (1984).
7. Cockshott, W.P., Atkinson, M.P., Bailey, P.J., Chisholm, K.J. & Morrison, R.
The persistent object management system. Software, Practice & Experience 14 (1984).
8. Dijkstra, E.W.
The structure of THE multiprogramming system. Comm.ACM 11, 5 (May 1968), 341-346.
9. Hoare, C.A.R.
Communicating sequential processes. Comm.ACM 21, 8 (August 1978), 666-677.
10. Hoare, C.A.R.
Monitors : an operating system structuring concept. Comm.ACM 17, 10 (1974), 549-557.

11. Ichbiah et al.,
The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. (1983). Also **Lecture Notes in Computer Science**. 155. Springer-Verlag (1983).
12. Krablin, G.L.
Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117.
13. Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P.
An integrated graphics programming environment. 4th UK Eurographics Conference, Glasgow (March 1986). In Computer Graphics Forum 5, 2 (June 1986),147-157.
14. Morrison, R., Bailey, P.J., Brown, A.L., Dearle, A. & Atkinson, M.P.
A persistent store as an enabling technology for an integrated project support environment. IEEE 8th International Conference on Software Engineering, London (August 1985),166-172.
15. Turner, D.A.
SASL language manual. University of St.Andrews CS/79/3 (1979).

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

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

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

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 Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

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 EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

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

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

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

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 EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

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. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

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.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, N0. 2, (Summer 1986) - see PPRR-23-86.

Internal Reports

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

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

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.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following Ph.D. theses have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 15th December 1986. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R	Presently no longer available
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-86	PS-algol Reference Manual - fourth edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-86	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00
PPRR-34-87	Binding Issues in Database Programming - Atkinson, M.P., Dearle, A., Cooper, R.L. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00

PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00
PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00
PPRR-38-87	Flexible Incremental Bindings in a Persistent Object Store Morrison, R., Atkinson, M.P. and Dearle, A.	£1.00
PPRR-39-87	Polymorphic Persistent Processes Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J.	£1.00