**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 8SX**

# Procedures as Persistent Data Objects

Alan Dearle.

**Persistent Programming**
**Research Project 9**
**Also**
**Glasgow Computing Science**
**Department Report CSC/84/R3**

# PROCEDURES AS PERSISTENT DATA OBJECTS

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

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

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

PERSISTENT PROGRAMMING
RESEARCH PROJECT 9

Also:

GLASGOW COMPUTING SCIENCE DEPARTMENT
REPORT CSC/84/R3

## Preface

This report is the first produced since part of the team moved from Edinburgh to Glasgow. A preliminary report of the ideas in this paper will be presented at the Fourth Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India (15 December 1984). The version presented here has been submitted to ACM TOPLAS. Persons wishing to cite this paper are requested to contact the authors for the correct citation information.

An interpreter for the language, which will run all the examples shown here, and which will accommodate reasonably sized collections of data is available written in C, and running under Berkeley UNIX 4.2 on the VAX and under PNX2 on the ICL PERQ. Further information about these and other implementations, and about the development of our persistent programming research is available from the authors at either institution or at the address given with the bibliography.

This work supported at the University of Glasgow by SERC grants GRC 21977 and GRC 21960 and at the University of St Andrews by SERC grant GRC 15907. The work is also supported at both Universities by grants from International Computers Ltd..

Author's addresses: M. P. Atkinson, Department of Computing Science, 14, Lilybank Gdns., Glasgow, Scotland, G12 8QQ. R. Morrison, Department of Computational Science, University of St Andrews, North Haugh, St Andrews, Fife, Scotland, KY16 9SX.



We wish to extend the traditional role of programming languages to provide the facilities normally left to other functions such as the file system or linker. Our target is to provide a totally integrated environment where the user never has to step outside the programming language for any computational activity.

The advantage of such an approach to system construction is in simplifying the number of different ad hoc mechanisms used in constructing large systems. Of course, it would be easy to build these ad hoc mechanisms into the language and not achieve the desired effect. Therefore it is important to have a regular design of the programming language to minimise these mechanisms.

In this paper we concentrate on the use of first class procedures and orthogonal persistence and show how the two concepts may be used to implement abstract data types, modules, separate compilation, views and data protection. Furthermore we demonstrate how the ideas may be used in system construction and version control.

The safety of the system is provided through the type mechanism of the language and we discuss the tradeoffs between security and flexibility and static and dynamic type checking for dynamically evolving systems.

Some of the issues presented in this paper relating to first class procedures have been discussed before [9,20,24,25,26] and we use such higher order functions in conjunction with our notion of persistence to demonstrate the above claims.

We have built the programming language PS-algol [1,7] to investigate these ideas and all the programs presented have been run on that system.

### 1.1 What is persistence?

The persistence of a data object is the length of time that the object exists. In traditional programming languages data cannot last longer than the activation of the program without the explicit use of some storage agency such as a file system or a database management system. In persistent programming, data can outlive the program and the method of accessing the data is uniform whether it be long or short term data. We have discussed this concept fully elsewhere [1]. The language concepts presented in this paper depend on persistence being provided as an orthogonal property of data; all data objects, whatever their type, have the same rights to long and short term persistence.

### 1.2 What are first class procedures?

Most programming languages provide facilities for abstractions over expressions and statements. Indeed these abstractions, functions and procedures, are often the only mechanisms for abstraction in a programming language. The power of the mechanism is derived from the fact that the user of the procedure does not require to know the details of how the procedure executes, only its effect. We use the word 'procedure' to represent both procedure and function when it is not necessary to differentiate between them.

The procedures of Algol 60[21] and Pascal[28] can only be declared, passed as parameters or executed. However, as has been pointed out by Morris [20] and Zilles [31], to exploit the device to its full potential it is necessary to promote procedures to be full first class data objects. That is, procedures should be allowed the same civil rights as any other



data object in the language such as being assignable, the result of expressions or other procedures, elements of structures or vectors etc. Lisp [18] was the first language with first class procedures and other languages include Iswim [15], Pal [9], Gedanken [24], Sasl [26], ML [19] and with some restrictions Euler [27] and Algol 68 [32]. Of course the applicative programming technique revolves around the ability to have first class procedures in the language and central ideas such as partial application are not properly implementable otherwise.

### 1.3 What is closure?

The most important concept in the understanding of first class procedures is that of closure [12,25]. The closure of a procedure is all the information required to execute the procedure correctly. It has two parts. The first part is the code to execute the procedure and the second part is its environment which contains the local and free variables of the procedure and is often implemented by a static chain [23].

### 1.4 PS-algol notation

We will use PS-algol to illustrate our ideas and here we introduce some of the syntactic elements of the language to help the reader's understanding of the later programs. This is perhaps best done by an explanation of a program.

```
let sum = proc( *int A -> int )
          begin
               let total := 0
               for i = lwb( A ) to upb( A ) do
                        total := total + A( i )
               total
```

```
        end
write "How big is the vector "
let n := readi()
let this.vector = vector 1::n of 0
write "Please input ",n," elements "
for i = 1 to n do this.vector( i ) := readi()
write sum( this.vector )
```

This program reads in the size of a vector followed by the elements of the vector and outputs the sum of the elements. Points to note are

a) the dot in the identifier 'this.vector' is part of the identifier and does not signify anything like module qualification as in other languages

b) all identifiers are introduced by **let** declarations. The **let** is followed by a := or = signifying a variable or a constant and then an expression. The type of the identifier is deduced from the initialising expression which must be present

c) declarations and clauses may be freely mixed

d) the procedure expression comprises the procedure type, in this case **proc( *int -> int )** i.e. a procedure from a vector of integers to an integer, followed by an expression of the correct type, again in this case integer. Note we have used a block expression here where the last expression in the block denotes its value

e) the procedure is applied by using its name and the parameters of the correct type, in this case sum( this.vector )

This rather quick introduction to the language will be augmented by discussion of the later examples.



## 2. First class procedures in relation to abstract data types.

The supporters of abstract data types [17] argue that it is essential for powerful languages to have an abstraction mechanism over data objects. In the same manner that a procedure separates the implementation of a task from its use, the abstract data type separates the representation of a data object from its use. Thus we have at once an abstraction mechanism and a protection mechanism. The abstract data type defines the operations available on the data object while only allowing the definition of the type to manipulate or access the representation. Languages which support abstract data types include Simula [4], Clu [16], Alphard [30], Euclid [14], ML[19] and Ada [11].

None of the above languages, with the exception of ML and Clu, support first class procedures. However, as has been pointed out by Horning [10], the advantages and aims of procedural and data abstraction are similar. Indeed if procedures are data objects the mechanism for both abstractions can be the same --- that of the procedure. This, of course, is not a new idea and was present in the work of Strachey [25] and Zilles [31].

### 2.1 The complex number example

To explain the mechanism the following program segment written in PS-algol [2] is given in Figure 3. The task it sets out to solve is to define an abstract object for a complex number and to allow only the operations of addition, printing and creation on the complex number.

```
let add := proc( pntr a,b -> pntr ) ; nullproc
let print := proc( pntr a ) ; nullproc
let complex := proc( real a,b -> pntr ) ; nullproc

begin
     structure complex.number( real rpart,ipart )

     add := proc( pntr a,b -> pntr )
            complex.number( a( rpart ) + b( rpart ),a( ipart ) + b( ipart ) )

     print := proc( pntr a )
              write a( rpart ),
              if a( ipart ) < 0 then "-" else "+",rabs( a( ipart ) ),"i"

     complex := proc( real a,b -> pntr )
                complex.number( a,b )
end

let a = complex( -1.0,-2.8 ) ; let b = complex( 2.3,3.2 )
print( add( a,b ) )
```

**The definition of an abstract type for complex numbers in PS-algol**

**Figure 3**

In PS-algol a structure class is a tuple of named fields with any number of fields of any type. The **structure** statement adds to the current environment a binding in the closest enclosing scope for the class name ('complex.number' in this example), and a binding for each field name ('ipart' and 'rpart' in this case). When an instance of a structure class is created (by complex.number( a,b ) above), it yields an object of that class which may be assigned to an object of type **pntr**. The class of a pointer is not determined at compile time but at run time and since the structure class is similar to a type definition in other languages this gives a degree of polymorphism to PS-algol.

The structure declaration in the example



**structure** complex.number( **real** rpart,ipart )

defines a structure with two real fields 'rpart' and 'ipart'. To create an object of this class we may use the expression

complex.number( 3.2,5.4 )

The fields of the structure may then be accessed by using a pointer expression followed by the structure field name in parenthesis. e.g.

a( rpart )

The example, in Figure 3, shows three procedure variables being declared and in the following block being assigned values. The representation of the complex number is encapsulated in the block and is not available to other parts of the program. Since the field names of the representation of the complex number are local to the block only the procedures defined in the block may use these names. Outside the block the names are invisible. Thus we have completely separated the representation of the data object from its use and achieved one of the aims of abstract data types. Indeed the block could be rewritten to represent the complex number in polar co-ordinates without changing the external meaning. Furthermore we have demonstrated again that the traditional block structure and scope rules of Algol 60 with the addition of first class procedures are sufficient to support abstract data types. Figure 4 shows how the block can be made into a function itself perhaps to be located elsewhere in the program or separately compiled.

```
structure complex.arithmetic( proc( pntr,pntr -> pntr )cadd ;
                              proc( pntr )cprint ;
                              proc( real,real -> pntr )ccomplex )

let complex.arith = proc( -> pntr )
begin
     structure complex.number( real rpart,ipart )

     complex.arithmetic(
     proc( pntr a,b -> pntr )    !cadd
     complex.number( a( rpart ) + b( rpart ),a( ipart ) + b( ipart ) ),

     proc( pntr a )              !cprint
     { write a( rpart ),
             if a( ipart ) < 0 then "-" else "+",rabs( a( ipart ) ),"i" },

     proc( real a,b -> pntr )    !ccomplex
     complex.number( a,b ) )
end !of complex.arith

!main program --- redo the names
let t = complex.arith()
let add = t( cadd ) ; let print = t( cprint ) ; let complex = t( ccomplex )

let a = complex( 1.2,0.3 ) ; let b = complex( 9.4,-3.2 )
print( add( a,b ) )
```

**The complex number package**

**Figure 4**

The structure class 'complex.arithmetic' contains three procedures as elements. The notation

**proc( pntr,pntr -> pntr )**

denotes the type of a function from two pointer parameters to an object of type pointer whereas **proc( pntr )** denotes the type of a procedure with one pointer parameter.

In the main part of the program an application of the function 'complex.arith' yields a structure of class 'complex.arithmetic' which is



assigned to the name 't'. In the procedure 'complex.arith' the same three procedures as before are defined and their closures exported via a structure. This is slightly more complex than the last version in that there is an extra dereference to obtain the same names but that is a syntactic problem which can easily be overcome if necessary.

### 2.2 Data protection

Morris [20] specified three ways in which a data object may be used in a manner not intended. They are

> "1. Alteration : An object that involves references may be changed without use of the primitive functions provided for the purpose.
>
> 2. Discovery : The properties of an object might be explored without using the primitive functions.
>
> 3. Impersonation : An object, not intended to represent anything in particular, may be presented to a primitive function expecting an object representing something quite specific."

The first two problems are overcome by the methods already demonstrated in PS-algol. Since the names of the fields in the structure class are only known to the primitive procedures, by the scope rules, then the objects can never be accessed except by the primitive procedures. However impersonation is a problem in PS-algol because structure class pointers are checked at run time. It is not that the impersonation will not be discovered but that it will cause a hard failure at run time. If

this is not acceptable it is possible to check the class of the object before allowing any operation on it. Thus we can define the program's action if an impersonation is attempted. In our example the procedure 'complex.arithmetic' may be rewritten as in Figure 5.

```
let complex.arith = proc( -> pntr )
begin
     structure complex.number( real rpart,ipart )

     let error = proc( pntr a -> bool )
                 if a isnt complex.number then
                 begin
                      write error.message
                      true
                 end else false

     complex.arithmetic(
     proc( pntr a,b -> pntr )     !cadd
     if error( a ) or error( b ) then nil else
        complex.number( a( rpart ) + b( rpart ),a( ipart ) + b( ipart ) ),

     proc( pntr a )               !cprint
     if error( a ) then write "This is not a complex number"
     else { write a( rpart ),
                  if a( ipart ) < 0 then "-" else "+",rabs( a( ipart ) ),"i"

     proc( real a,b -> pntr )     !ccomplex
     complex.number( a,b ) )
end !of complex.arith
```

**The complex number package with impersonation checks**

**Figure 5**

The above example is rather cumbersome and demonstrates how a good exception handling mechanism [11,19] could be utilised to provide simpler code.

### 2.3 Comparison of first class procedures and abstract data types

Figure 6 below shows how the abstract type for complex numbers may be



declared in ML. We ignore the fact that ML does not have **real** as a base type for this example.

```
abstype comp = comp of real # real
with
    val add( comp( r1,i1 ) ) ( comp( r2,i2 ) ) = comp ( ( r1 + r2 ),( i1 + i2 ) )
    and print( comp( r,i ) ) = ( output( terminal,stringofreal( r ) ) ;
        output( terminal, if i < 0.0 then "-" else "+" ) ;
        output( terminal,stringofreal( realabs( i ) ) ) ;
        output( terminal,"i" ) )
    and complex r i = comp ( r,i )
end
```

**An example abstract datatype declaration written in ML**

**Figure 6**

It is useful to compare this with the declaration given in Figure 3. The **abstype .... with** construct in ML is essentially an environment manipulation, so that after the construct the declarations appearing between **with** and the corresponding **end** are installed in the subsequent environment, but the type 'comp' is available only in the environment of the declarations after **with.** This is nearly equivalent to the notation in Figure 3, with the following detailed correspondence.

1. In Figure 3 the three **let** clauses introduce the three names into the outer environment whereas in Figure 6 the same three names are left, by being declared after the **with**, in the outer scope.

2. The **begin end** pair delimits a scope level as does a **with end** pair.

3. In Figure 3 the representation of the complex number is introduced by the **structure** declaration which is local to this inner scope only. In Figure 6 the representation of complex is introduced by

the **abstype** statement and this binding is available only in the scope by **with** and **end**.

4. In both cases in the inner scope three bindings of names to procedural values are declared.

The similarity is semantically almost complete. As a consequence of the need to define the binding in one scope and introduce the name in another the names have been declared as variables as in Figure 3, whereas they are constants in ML. The other differences are merely syntactic --- the main one being the rather redundant declarations of 'add', 'print' and 'complex'. The designer has the choice of requiring this or adding new constructs such as **abstype** to the language.

Another aspect of using a procedural mechanism is that it provides parametric abstract types. Let us suppose that an abstract type for vectors is required but that different dimensional spaces may be used and that vectors from these require different representations and different operators. Figure 7 shows an appropriate definition.

```
structure vector.pack( proc( pntr,pntr -> pntr )add ; proc( pntr )print ;
                       proc( *real -> pntr )create )

let make.vector.pack = proc( int n -> pntr )
begin
     structure vec( *real rep )

     if n < 2 then { write "silly dimension" ; nil }
     else vector.pack(
          proc( pntr a,b -> pntr )  !add
          begin
               let v = vector 1::n of 0.0
               for i = 1 to n do v( i ) := a( rep )( i ) + b( rep )( i )
               vec( v )
          end,
          proc( pntr a )         !print
          begin
               write a( rep,1 )
               for i = 2 to n do write ", ",a( rep )( i )
          end,

          proc( *real r -> pntr )   !create
          if upb( r ) = n and lwb( r ) = 1 then vec(r)
                                           else { write "wrong size" ; nil } )
end ! of make.vector.pack
```

**An example of defining a parameterised type**

**Figure 7**

The operators may now be used as shown in Figure 8. To introduce parameterisation of abstract types may mean more complexity than utilising the parametric mechanisms we already have with procedures.

```
let Pack.2D = make.vector.pack( 2 )
let Pack.3D = make.vector.pack( 3 )

let add2 = Pack.2D( add ) ; let print2 = Pack.2D( print )
let mk2d = Pack.2D( create )
let add3 = Pack.3D( add ) ; let print3 = Pack.3D( print )
let mk3d = Pack.3D( create )

let v1 = mk2d( @1[ 1.1,2.2 ] )
let v2 = mk2d( @1[ 3.3,4.4 ] )
let v3 = add2( v1,v2 )

print2( v3 )

let w1 = mk3d( @1[ 1.1,2.2,3.3 ] ) .............
```

**An example of using the parameterised type**

**Figure 8**

### 3. First class procedures can perform as modules

Many languages have also introduced the concept of modules Ada, Clu, ML, Modula-2 [29].

These appear to serve three functions:

i) Provide a mechanism for own data, that is data bound with the module over the scope or lifetime of the module, rather than only for individual applications of the module.

ii) To be the unit of program building being used in system construction as a unit of specification, a unit of compilation, testing and assembly.

iii) As a localisation or hiding of certain design decisions, in other words, the provision of abstract types.

We show that, in conjunction with persistence as an orthogonal property, first class procedures perform all these roles. The last has already been demonstrated, the first can depend either on partial application or be obtained in conjunction with the program building facilities. These are simply based on the idea that programs may use procedures which other programs have left in a database. Each of these will now be demonstrated.

It is important to note, once again, though lack of space precludes showing it in every example, that the normal parametric mechanisms of procedures means that we now have modules which may be parameterised, and for which many instances may exist. This is obtained without adding extra constructs or concepts to the language.

### 3.1 Partial application

An advantage of having procedures as first class data objects is the possibility of having partially applied functions. As a tutorial,



preparatory to discussing views, let us provide an abstract structure to maintain lists of things to do, for different people in different contexts. This may be defined as shown in Figure 9.

```
structure list.pack( proc( string )add ; proc()clear ; proc()print )

let make.list.Pack = proc( string person,context -> pntr )
begin
     structure cell( string item ; pntr next )
     let list.start := nil

     list.pack(
     proc( string s ) ; list.start := cell( s,list.start ),     !add
     proc() ; list.start := nil,     !clear
     proc()       !print
     begin
          write "'n list of tasks for ",person," doing ",context
          let l := list.start
          while l ≠ nil do
          begin
               write "'n",l( item ) ; l := l( next )
          end
          write "'n"
     end
               )
end
```

**Procedure to make various lists and provide routines to maintain them**

**Figure 9**

This can be used the way shown in Figure 10.

```
let RMs = make.list.Pack( "Ron","Finish Paper" )
let MPAs = make.list.Pack( "Malcolm","Finish Paper" )
let RMadd = RMs( add ) ; let RMprint = RMs( print )
let MPAadd = MPAs( add ) ; let MPAprint = MPAs( print )
RMadd( "read Malcolm's notes" ) ; MPAadd( "Write rest of comments" )
RMadd( "type corrections" ) ; MPAadd( "Read next draft" )
RMadd( "Fix references" ) ; MPAadd( "Post last corrections" )
MPAprint() ; RMprint()
```

**Using the procedures with local "memory" of lists**

**Figure 10**

Now on the assumption that a given person has tasks in a number of contexts, it may be preferable to partially apply this procedure to yield procedures for each person as in Figure 11.

```
let make.lists.for = proc( string person -> proc( string -> pntr ) )
                           proc( string context -> pntr )
                           make.list.Pack( person,context )
```

**Partial application of the make.list.Pack procedure**

**Figure 11**

This can be used as shown in Figure 12.

```
let Rons.list.maker = make.list.for( "Ron" )
let Malcolms.list.maker = make.list.for( "Malcolm" )

let MPA.paper = Malcolms.list.maker( "First Class Fns Paper" )
let MPA.shopping = Malcolms.list.maker( "Shopping" )
```

**Using the partially applied list maker**

**Figure 12**

In these examples the procedures yielded by functions have own data associated with them (the lists, the tasks and the persons in this example) and so we have demonstrated that the first requirement for modules can be met by first class procedures.

## 4. Separate compilation and binding

Assuming the provision of persistence we now demonstrate how the procedure may be used as the unit of system construction and the unit of definition. Suppose a system is to be built out of the list maintaining program - then to separately compile the list maintainer we could write a



program such as that shown in Figure 13.

```
structure list.Pack( proc( string )add ; proc()clear ; proc()print )

let make.list.Pack = proc( string person,context -> pntr )
begin
     let list.start := nil ; structure cell( string item ; pntr next )
     list.Pack(
                proc
                proc                        as in Figure 9
                proc
               )
end

structure mlp.container( proc( string,string -> pntr )mlp )

let db = open.database( "Library","Gigha","write" )
if db is error.record do { write "Database can''t be opened" ; abort }
s.enter( "make.list.Pack",db,mlp.container( make.list.Pack ) )
commit()
```

**A complete PS-algol program to compile a pack of procedures and store them in a database for future use**

**Figure 13**

Since the program utilises the persistent mechanisms of PS-algol they are reviewed here, but the reader who requires complete information should read [1,2]. The 'open.database' operation opens the database with the name given by the first parameter, establishing the rights specified by the third parameter by quoting the password given by the second parameter. It also begins a transaction which is completed by a 'commit' or aborted by **abort**. 's.enter' is one of the operations on tables, PS-algol's associative structures. By convention a successful 'open.database' yields one of these tables. 's.lookup' is also available to obtain entries from a table.

### 4.1 Separately compiled code

We now use the definition in Figure 13 in a program to start a database for a given person, in which are kept lists on various topics. This is shown in Figure 14.

```
!A program to start a new database for someone's collection of lists
structure error.record(string error.context,error.fault,error.explain)
!first get the predefined module for maintaining lists.
let db := nil
repeat
     db := open.database( "library","Gigha","read" )
while db is error.record do
     write "'n sorry the library is being updated"

structure mlp.container( proc( string,string -> pntr )mlp )

let MkListPack = s.lookup( "make.list.Pack",db )
if MkListPack = nil do { write "Make list pack not compiled yet" ; abort }

!find out about the customer
write "Who are you?" ; let p = read.a.line()
!set up this database
write "What password?" ; let pw = read.a.line()
let db2 = open.database( p++".lists",pw,"write" )
if db2 is error.record do { write "Sorry no db space" ; abort }
!insert a table for a person's lists indexed by topic
s.enter( "topics",db2,table() )
!part apply MkListPack to ensure name always p
let a.Persons.make.lists = proc( string topic -> pntr )
                           MkListPack( mlp )( p,topic )
!preserve that for future use
structure a.Persons.list( proc( string -> pntr )h.list )

s.enter( "a.Persons.MKlist",db2,a.Persons.list( a.Persons.make.lists ) )
commit()
```

**An example of using a separately compiled procedure in PS-algol**

**Figure 14**

Examination of Figure 14 shows a number of features. First, a precompiled collection of definitions was obtained from the communal database "Library". The code for this is the loop (normally executed once) to gain access to the program library down to the test that the list package has been defined. This is equivalent to the module being obtained in a typical module based language (ML for example) by

**get**<Module name>

**use**<Module name>

It seems that this latter form is more succinct. However if the arrangements for libraries and naming are agreed a standard procedure, such as that shown in Figure 15 can be defined to achieve the same effect equally succinctly.

### 4.2 Example of software tool construction

```
!A standard procedure to obtain a module

let get.from.any = proc( string module,lib,libpw -> pntr )
begin
     repeat
          let db = open.database( lib,libpw,"read" )
     while db is error.record do
     begin
          write "Sorry for the delay, library",lib,"is being updated"
          wait( 5 )
     end
     let wanted.module = s.lookup( module,db )
     if wanted.module = nil do write "Warning: Module",module,"not defined"
     wanted.module
end
let get = proc( string module -> pntr )
          get.from.any( module,"library","Barra" )
```

**Standard module fetching procedure defined in PS-algol**

**Figure 15**

In Figure 14, the second part of the program uses the predefined list manipulating module to define a more specific module, which is left for

further programs to use. This demonstrates two aspects of module-use:

i) the module was used without its implementation being seen by the programmer - giving adequate protection against exploitation of accidents of the present implementation.

ii) modules can be synthesised using other modules, allowing construction of large programs, while the individual program text that has to be read to understand the program at a given level is kept small.

The approach to module collection demonstrated in Figure 15 is just one of many that could be defined. Different software construction groups may define their own module naming and module storage conventions, and may have their versions of 'get' and 'get.from.any' carry out authorisation procedures and keep records of what has been used. This gives the basis for constructing a variety of software construction tools within the language.

**4.3 Binding**

Our implementation combines both dynamic and static binding. Variables within program text are statically bound, but the values of pointer variables are dynamically bound to instances of structure classes. This dynamic binding does not prevent programs being stongly typed but it does imply that some type checking is dynamic (occurring near in time to the final evaluation) rather than static (occurring at initial program analysis).

We choose this phraseology, rather than referring to compile time and



execution time as we see program evaluation essentially distributed between the time the program source is first presented, and the completion of execution. Partially analysed programs may be left for further analysis and evaluation as data and resources become available. The consequence may be, that checks can be factored out of the internal operations.

This dynamic evaluation has inevitable costs. The program which depends on it may contain errors which could have been detected at compile time and hence reported more opportunely. The program which uses it may execute more slowly because of the run-time checks. But many programs, even in a statically checked language, may need equivalent run-time checks. For example, to check that the tag field of a case statement matches the field being used in a record, often not checked in Pascal, or that an index is within range, or that an input text can be converted to the expected type. By denying dynamic binding the class of programs that can be written is reduced, for example without it it would not be possible to write the associate structure is PS-algol that maps to members of a mixture of structure classes including those yet to be declared when the map is created. This lost opportunity can lead to other costs, either these general purpose components cannot be written within the language, so that some other language is used, or the programmer simulates types and languages within the original language. The former incurs the cost of linking between languages, loss of portability, and increased programming difficulty. The latter incurs the cost of an extra layer of interpretation and mapping and obscure convention to trap people who

subsequently modify the program. The worst danger of the first solution is that it requires the introduction of a Trojan horse into the type system. So the choice of dynamic or static checking is a matter of choosing the appropriate trade off between security and flexibility. We note that this is similar to the debate in mathematics between open and closed systems. In the context of software engineering it is still an open question as to which is preferable.

We suspect that for very large or very long lived systems local change and hence local evolution may be necessary. In which case dynamic binding is a necssary mechanism. The programmer should be able to choose the points at which he wants dynamic binding, since to bind all names dynamically would be uneconomic. Typically we would expect small units to be statically bound and to form larger units dynamically bound together.

The choice is offered via the constancy mechanism is PS-algol. If an identifier is bound constantly (=) the system may assume it will not change. If it is a variable (:=) the value may be replaced.

In either case, since many instances of a class may exist, an instance which implements an abstract data type in a way believed to be appropriate to that particular usage can be bound. In the latter case, if another implementation subsequently appears more appropriate, the new value can be assigned. There are difficulties here that have not yet been investigated, as the instance may have local storage.

The binding mechanism of PS-algol, in which the type of a referend is checked when it is first brought from the database, has provided incremental type checked linking and loading. A considerable



simplification is achieved over our previous system (prior to procedures as persistent data objects) where explicit use of a type checking linker was necessary.

## 5. Comparison of modules with first class functions

We can now compare the anatomy of a module with that of our definition using first class procedures. In a conventional modular language there are three separate components concerned with modules. These are:

i) the module interface definition
ii) the module body definition
iii) the module inclusion statement

The last has already been discussed in connection with Figure 15. The definition of a structure to carry the pack of interfaces is the first class procedure equivalent of the module interface definition. As in module based languages it appears both in the context where the module body is defined and in every context where the module is used. It completely defines the types of all objects that may pass across the interface, and with the type matching rules in PS-algol this ensures that only modules with correctly matching types are assembled together. Although only procedural components of a structure/interface are shown in the examples, other data types may appear allowing direct access between the module and its users to some shared variable. The program development environment may provide tools to permit this interface definition to be invoked by referring to its name.

The module body in a modular language usually contains concepts for defining imported, exported and private variable lists. It usually has a method of defining data storage and data manipulation. All these are defined here by use of the normal algol declarations and block structure without additional concepts.

Where a module has internal storage, there is often a need to make many instances of the module, possibly with different initial data. This can be achieved with these first class procedures by simply calling them repeatedly with different parameters - no special mechanism is required. This is illustrated in Figure 14, where each time that program is used, a new instance of the same module is created, with a different value for person stored within it. Note that different definitions of procedures which yield results of some type containing fields with equivalent procedures allows the provision of module implementations tailored to some known pattern of use.

**6. New version installation**

With all large systems, constructed out of separate modules, there is a problem of managing the installation of new versions. It is necessary to modify the implementation of modules and then arrange for their subsequent use. Often this can only be done when no part of the system is running, then the new modules are installed by a complete system rebuild. This may take considerable resources. The alternative of replacing a module in situ has to be carefully managed, as it certainly could not be done safely when the module is in use if that execution were affected.

In PS-algol the transaction mechanism makes the concurrent revision and installation of modules safe as the effects of a transaction are only visible to other transactions that start after the transaction has committed. Programs starting after it will use the new one for the whole program execution if they are written in the style shown in Figure 14. Other software tools are necessary to complete an implementation of such a dynamic binding mechanism.



More sophisticated mechanisms can be implemented with these facilities. For example, a program may arrange to bind a particular version of one module to the package it constructs, by leaving it directly referenced, or leave it to be picked up when the package is run collecting the latest version. Software tools could be written, to build up systems where groups of modules could be installed, retained, replaced etc. using no more language concepts than the features illustrated here.

**7. First class functions as a view mechanism**

View mechanisms are used in databases to perform two roles:

i) to provide a stable and appropriate view to the programmer

ii) to implement protection and privacy controls.

The first class functions, together with partial application perform both of these roles. Stability means that the underlying data may be changed without impact upon programs it was not intended to alter apart from possible changes in performance. The person who changes the underlying data is usually responsible for redefining the mapping that provides the view except where the only available mappings are so simple

that the new mapping may be inferred. If we interpose a set of functions, then redefinition of these functions will provide the required stability. Similarly, they can be defined so as to provide the appropriate view and the access controls. We have discussed this use of first class persistent functions elsewhere [3].

Figure 14 will again serve as an example. The function saved in the database as 'a.Persons.MKlist' will now only make up lists, print lists etc. for the one person who created this database. Thus the view of the data has been made appropriate by allowing the person to avoid redundantly giving their own name every time, and has also been restricted to lists concerned with that data. Note that the control and the remapping is quite finely controlled but not over restrictive. For example there is nothing to stop the programmer using this database to hold other data as well, to which they may have any view or access. This seems correct.

In Figure 14 however the view constructed is not as secure as we might wish, as a programmer using it could operate directly on the table which holds the set of topics. To overcome this we refine the definition, as shown in Figure 16. The revised version prevents any misuse of the table of topics by making it available only within the body of the 'make.lists' procedure declaration. The refinement also produces four procedures to work over the data, one to initiate a list on a topic, and the others as before, except that they now take a topic as a parameter and work for any list for the given person. This illustrates the radical revision of views that may be constructed, and the way precise control over the operations on data may be obtained.



```
!Refined Program to start a database for lists

structure error.record( string error.context,error.fault,error.explain )
write "Who are you?" ; let p = read.a.line()
write "What password?" ; let pw = read.a.line()

let a.Persons.make.lists = proc( string p -> pntr )
begin
     let table.for.topics = table()

     let get.topic := proc( string topic -> pntr ) ; nullproc
     get.topic := proc( string topic -> pntr )
     begin
          let pack = s.lookup( topic,table.for.topics )
          if pack = nil then
          begin
               write "You have not started that topic'n"
               get.topic("dummy")
          end else pack
     end

     let db = open.database( "library","Gigha","read" )
     if db is error.record do
     begin
          write "Cannot open database Library",
                "'n",db( error.fault ),"'n",db( error.explain )
     end

     structure mlp.container( proc( string,string -> pntr )mlp )
     let mklp = s.lookup( "make.list.Pack",db )( mlp ) !see Figure 14

     structure list.pack( proc( string )add ; proc()clear ; proc()print )

     let start.topic = proc( string topic )
                       begin
                            let pack = mklp( p,topic )
                            s.enter( topic,table.for.topics,pack )
                       end

     let add.topic = proc( string topic,task )
                     get.topic( topic )( add )( task )

     let clear.topic = proc( string topic )
                       get.topic( topic )( clear )()

     let print.topic = proc( string topic )
                       get.topic( topic )( print )()

     start.topic( "dummy" )
```

```
    structure topic.pack( proc( string )start.t ; proc( string,string )add.t ;
                          proc( string )clear.t,print.t )

    topic.pack( start.topic,add.topic,clear.topic,print.topic )
end

let db = open.database( p++".lists",pw,"write" )
if db is error.record do { write "sorry no db space" ; abort }

s.enter( "a.Persons.MKlist",db,his.make.lists( p ) )
commit()
```

**A refinement of Figure 14 to give a more restrictive and convenient view**

**Figure 16**

Figure 17 then illustrates how this view may be used. Note that the programmer has only the four operations available, and has no knowledge of or access to the way the lists were represented. In this case the view was fairly appropriate for the task. Another view might have provided an extra operation to set the current topic, thus economising on the passing of the 'topic' parameter.

```
!program to provide end user interface to lists

structure error.record( string error.context,error.fault,error.explain )
write "Who are you?" ; let p = read.a.line()
write "Your password?" ; let pw = read.a.line()

let db = open.database( p++".lists",pw,"write" )
if db is error.record do { write "Sorry no db space" ; abort }
!get & unpack saved view
let hML = s.lookup( "a.Persons.MKlist",db )

structure topic.pack( proc( string )start.t ; proc( string,string )add.t ;
                      proc( string )clear.t,print.t )

let st = hML( start.t ) ; let ad = hML( add.t ) ; let cl = hML( clear.t )
let pr = hML( print.t )
let current.topic := "dummy" ; let todo := ""
repeat
begin
    write "'n what shall I do?" ; todo := read.a.line()
```



```
    case todo of
    "quit"   : {}
    "start"  : { write "topic?" ; current.topic := read.a.line()
                 st(current.topic) }
    "change" : { write "new topic?" ; current.topic := read.a.line() }
    "add"    : { write "item?" ; ad( current.topic,read.a.line() ) }
    "clear"  : cl( current.topic )
    "print"  : pr( current.topic )
    default  : write "Command not understood"
    write "'n"
end
while todo ≠ "quit"
commit()
```

**A PS-algol program utilising the view constructed in Figure 16**

**Figure 17**

## 8. Conclusions

We have tried to relate to the reader the potential of combining persistence with procedures as first class data objects. While all the examples in the paper run on our present implementation we recognise this as a demonstration rather than a completion of an area of research.

The goal of better programming environments led us to implement persistent procedures. Programming will be simpler when a programmer can work within a single environment which is described and manipulated in terms of a simple set of concepts. The introduction of persistence in conjunction with procedures is a step towards achieving this, as it permits program construction from separately defined parts to be described and implemented using the same language and same type enforcement as is used for writing programs. A simplification which is also sought by the authors of Pebble [6]. We differ from them in choosing to accept a dynamic binding of program parts.

We have decided to explore such dynamic binding as we believe that

for very large systems the cost of completely rebuilding the system to accommodate changes will be too great, and the opportunity for rebuilding too rare. But dynamic binding involves two costs: a delay in detecting some errors and increased run-time checks. We argue that the delay in detecting errors can be minimised by eager type checking, whereby the language processor attempts to detect errors as soon as possible. We suspect that any language powerful enough, and which also allows methods of detecting errors, will have some errors which cannot be detected until late in the elaboration (input errors, overflow errors and some bounds errors are examples). By permitting the programmer to choose to defer other checks, but still insisting that evaluation is preceded by or includes the checks, we change only the proportion of late detected errors. In doing so, we allow a class of programs to be written which do more general manipulation, the DBMS, data editors and browsers etc. than is otherwise possible. These programs seem to suggest that dynamic binding is necessary at the interface to long term data. The cost of dynamic checking can then be minimised by arranging to factor out checks (ultimately to language analysis time as in static type checking) with the aim of performing checks once only. Research into this is reported by Owoso [22]. Further research and experience of using such languages is necessary to determine whether this is an appropriate tradeoff.

A number of requirements of modern programming languages, abstract types, modules, separate compilation, module assembly with interface checking, incremental loading and views are met by the provision of procedures as first class data objects. It has long been understood that



it is desirable to be parsimonious in introducing concepts into a language design. It is a challenge to see if the present experimental language which provides the semantics can be extended to a production language, still retaining the parsimony.

At present the language is deficient in syntactic support for some common constructs which can be coded. This should be relatively simple to provide. More fundamentally the types, some of which persist, and the names, some of which persist, need to be treated more consistently as objects in the language. Standard concepts, such as exception handling, need to be added and our implementation of concurrency needs to be modified to reduce overlocking and be more under programmer control.

As this is the theme of our current research we hope to be able to report soon whether it can proceed while we retain simplicity.

The provision of procedures and procedure activations that can exist in a database opens up a number of interesting avenues of research, some of which we have introduced in the paper:

i) Software tools: managing versions, program assembly, program and data dictionaries, reporting on data and programs etc. can be written entirely within such languages. There are two advantages - the complexity of going outside the language or of building interpreted representations is avoided and the portability of implementations is improved.

ii) Views and complex mechanisms of authorisation can be coded within the language. New features are first needed to allow views to be bound to existing data if all the functions provided in databases are to be

accommodated.

iii) Dynamic mechanisms to be invoked on database operations can be implemented. This would include those proposals such as triggers in System/R [5] and conditions in CODASYL DBTG [8] neither of which were implemented, we suspect because the proper binding of procedures within the database had not been developed at that time. This could lead to various forms of active and deductive databases.

iv) Integrated interactive environments for general or special programming may prove easier to construct and to evolve given such persistent procedures.

Since readily understood and easily implemented languages are needed as a foundation for software engineering, we argue that serious consideration should be given to languages which support procedures as data objects, which have an orthogonal provision of persistence and which are not overgrown with numerous other concepts.

## 9. Acknowledgements

Our dependence on a working implementation, from which we have learnt much, cannot be overstated. It was built largely by the following members of our team: Pete Bailey, Fred Brown, Paul Cockshott and Al Dearle. Krishna Kulkarni partially explored the potential of our system as it was built [13] assisting greatly in refining the concepts and implementation. We benefitted from suggestions and ideas arising in discussions with Peter Buneman and Tony Davie. Thanks are also due to three anonymous referees who suggested many sensible improvements to our first submission.

We appreciated the support of the University of Pennsylvania and the Australian National University when we were preparing this paper.



## 10. References

1. Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. An approach to persistent programming. Computer Journal 26, 4 (1983), 360-365.

2. Atkinson, M.P., Bailey, P.J., Cockshott, W.P. & Morrison, R. PS-algol reference manual. Universities of Edinburgh and St Andrews PPR-8 (1984).

3. Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. Progress with persistent programming, **Database - role and structure**, Cambridge University Press, Cambridge, 1984.

4. Birtwistle, G.M., Dahl, O.J., Myrhaug, B & Nygaard, K. **SIMULA BEGIN.** Auerbach (1973).

5. Blasgen, M.W. et al. System R : an archectural overview. IBM systems journal 20 (1977), 41-62.

6. Burstal, R. & Lampson, B. A kernal language for abstract data types and modules. Proc. international symposium on the semantics of data types, Sophia-Antipolis, France (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. Codasyl DBTG., Codasyl committe on data system languages. Codasyl database task group report. ACM (1971).

9. Evans, A. PAL a language designed for teaching programming linguistics. Proc. ACM 23rd. Nat. Conf. Brandin Systems Press

(1968), 395-403.

10. Horning, J.J. Some desirable properties of data abstraction facilities. ACM Sigplan Notices 11 (1976), 60-62.

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

12. Johnston, J.B. The contour model of block structure processes. ACM Sigplan Notices 6, 2 (1971), 56-82.

13. Kulkarri, K.G. & Atkinson, M.P. Experimenting with the functional data model. **Database : role and structure**, Cambridge University Press, Cambridge (1984).

14. Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G. & Popek, G.J. Report on the programming language Euclid. ACM Sigplan Notices 12, 2 (1977), 1-79.

15. Landin, P.J. The next 700 programming languages. Comm.ACM 9, 3 (1966), 157-164.

16. Liskov, B.H., Synder, A., Atkinson, R. & Schiffert, C. Abstraction mechanisms in CLU. Comm.ACM 20, 8 (1977), 564-576.

17. Liskov, B. & Zilles, S.N. Programming with abstract data types. ACM Sigplan Notices 9, 4 (1974), 50-59.

18. McCarthy, J. et al. Lisp 1.5 Programmers manual. M.I.T. Press Cambridge Mass. (1962).

19. Milner, R A proposal for standard Ml. Technical Report CSR-157-83 University of Edinburgh. (1983).

20. Morris, J.H. Protection in programming languages. Comm.ACM 16, 1 (1973), 15-21.



21. Naur, P. et al. Revised report on the algorithmic language Algol 60. Comm.ACM 6, 1 (1963), 1-17.

22. Owoso, G.O. Types and data management for persistent data. Ph.D. thesis (submitted), University of Edinburgh (1984).

23. Randell, B. & Russell, L.J. **Algol 60 Implementation.** Academic Press (1964).

24. Reynolds, J.C. Gedanken a simple typeless language based on the principle of completeness and the reference concept. Comm.ACM 13, 5 (1970), 308-319.

25. Strachey, C. Fundamental concepts in programming languages. Oxford University Press, Oxford (1967).

26. Turner, D.A. SASL language manual. University of St.Andrews CS/79/3 (1979).

27. Wirth, N. & Weber, H. EULER a generalisation of algol. Comm.ACM 9, 1 (1966), 13-23.

28. Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.

29. Wirth, N. **Programming in Modula-2** : Second Edition. Springer-Verlag, Berlin, 1983.

30. Wulf, W.A., London, R.L. & Shaw, M. An introduction to the construction and verification of Alphard programs. IEEE Soft. Eng SE-2, 4 (1976), 253-265.

31. Zilles, S.N. Procedural encapsulation : a linguistic protection technique. ACM Sigplan Notices 8, 9 (1973).

32. van Wijngaarden, A. et al. Report on the algorithmic language Algol

68. Numerische Mathematik 14,1 (1969), 79-218.

Papers Published by the
Data Curator Group

# 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.


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 department as CSR-26-78).

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

Atkinson, M.P.
'Progress in documentation: Database management systems in library automation and information retrieval', Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as departmental report CSR-43-79.

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

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

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

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

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 (January 1982) - also as Departmental Report CSR-90-81.

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

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
'EDQUSE reference manual', Department of Computer Science, University of Edinburgh, September 1981.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'PS-algol: An Algol with a Persistent Heap', ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as Departmental Report CSR-94-81.

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

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'Algorithms for a Persistent Heap', Software Practice and Experience, Vol.13, No.3 (March 1983). Also as Departmental Report CSR-109-82.

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

Atkinson, M.P.
'Data management', in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

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

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

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
Databases - Role and Structure, CUP 1984.



Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. To be published (revised) in the Workshop proceedings 1983, see PPR-2-83.

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', in The Computer Journal, 1983, Vol.26, No.4 - see PPR-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 - see PPR-2-83.

Morrison, R., Weatherall, M., Podolski, Z. & Bailey, P.J.
High level language support for 3-dimension graphics, Eurographics Conference Zagreb, Sept. 1983.

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, January 1984.

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

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", to appear in the proceedings of the Fourth Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India, December 1984. See PPR-9-84.

Hepp, P.E. & Atkinson, M.P.
Tools and components for rapid prototyping with persistent data, to be submitted.

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, (July 1984).

Kulkarni, K.G. & Atkinson, M.P.
EFDM : Extended Functional Data Model, submitted to The Computer Journal.

Kulkarni, K.G. & Atkinson, M.P.
EFDM : A DBMS based on the functional data model, submitted to the IEEE transactions on SE.

Atkinson, M.P. & Buneman, O.P.
Design issues in database programming languages, in preparation.

Atkinson, M.P. & Morrison, R.
"Proceduresx as persistent data objects", submitted to ACM TOPLAS (Sept. 1984) - see PPR-9-84.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
A persistent graphics facility for the ICL PERQ.

## Theses

The following Ph.D. theses have been produced by member of the group and are available from The Secretary, Persistent Programming Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland.

W.P. Cockshott
Orthogonal Persistent, February 1983.

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

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

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



## Data Curator Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 24 October 1984.

Copies of documents in this list may be obtained by writing to The Secretary, The Persistent Programming Research Group, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ.

| | | |
|---|---|---|
| PPR-1-83 | The Persistent Object Management System | [Printed] |
| PPR-2-83 | PS-algol Papers: a collection of related papers on PS-algol | [Printed] |
| PPR-3-83 | The PS-algol implementor's guide | [In Preparation] |
| PPR-4-83 | The PS-algol reference manual | [Printed] |
| PPR-5-83 | Experimenting with the Functional Data Model | [Printed] |
| PPR-6-83 | A DBS Architecture supporting coexisting user interfaces: Description and Examples | [Printed] |
| PPR-7-83 | EFDM - User Manual | [Printed] |
| PPR-8-84 | Progress with Persistent Programming | [Printed] |
| PPR-9-84 | Procedures as Persistent Data Objects | [Printed] |
| PPR-10-84 | A Persistent Graphics Facility for the ICL PERQ | [Printed] |