This paper should be referenced as:

Atkinson, M.P. & Morrison, R. “Persistent First Class Procedures are Enough”. In **Lecture Notes in Computer Science 181**, Joseph, M. & Shyamasundar, R. (ed), Springer-Verlag (1984) pp 223-240.

Persistent first class procedures are enough

Malcolm P. Atkinson and Ronald Morrison

University of Glasgow, Glasgow, Scotland G128QQ.
and
University of St Andrews, St Andrews, Scotland KY169SX.

Abstract

We describe how the provision of a persistent programming environment together with a language that supports first class procedures may be used to provide the semantic features of other object modelling languages. In particular the effects of information hiding, data protection and separate compilation are provided and a comparison of the method with more traditional techniques is examined.

Introduction

We explain what is meant by extending rights of procedures and functions in a procedural language to be consistent with those of other data types such as integer or array. This is shown to be useful and elsewhere we have demonstrated it is implementable [5].

In particular the effects of information hiding, data protection and separate compilation can be achieved without introducing new concepts such as modules and abstract types. The relative merits of the two approaches are reviewed. Separate program preparation depends on making the procedure a first class data object and providing orthogonal persistence. The power of this consistent treatment of procedures is obtained without adding to the complexity of the language. Indeed the language is simplified, there being fewer concepts for the programmer to understand.

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.

What are first class procedures?

Most programming languages provide facilities for abstractions over expressions and statements. Indeed these abstractions, functions and procedures let us say, are often the only mechanisms for abstraction in the 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[19] and Pascal[26] can only be declared, passed as parameters or executed. However, as has been pointed out by Morris [16] and Zilles [29], 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 [14] was the first language with first class procedures and other languages include Iswim [11], Pal [6], Gedanken [22], Sasl [24], ML [15] and with some restrictions Euler [25] and Algol 68 [30]. 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 impossible otherwise.

What is Closure?

The most important concept in the understanding of first class procedures is that of closure [23,9]. The closure of a procedure is all the information required to execute the procedure correctly. It is in 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 usually implemented by a static chain [21]. In order to execute the procedure correctly we must have both parts of the closure. We will restrict ourselves here to block structured languages with static scope rules.

In block structured languages such as Algol 60, Pascal and S-algol [17] we very rarely need both parts of the closure to be recorded explicitly for the procedure. This is because the scope rules determine that a procedure can only be called from a position in the program where all the free variables of the procedure are accessible. The local variables do not exist before and after the call so the static chain is computable at the time of the call.

This is illustrated in Figure 1 where we have a program written in S-algol. In procedure 'one' we have the free variables 'a' and 'b'. However since the procedure may only be called after its declaration in the same block or inner blocks it is always possible for the compiler to calculate the static chain of the procedure from the static chain of the block that calls it.

```
let a := 3
.
.
begin
        let b := 16
        .
        .
        procedure one
        ! start of scope of procedure one
        begin
                .
                .
                write a + b
        end
        .
        .
        one
        begin
                .
                .
                one
        end
        ! end of scope of procedure one
end
```

A program where the full closure is not required
Figure 1

Algol 60, Pascal and S-algol all allow procedures to be passed as parameters to other procedures and clearly from Figure 2 it can be seen that the static chain may not always be computable from the block surrounding the call.

```
procedure A( procedure( int -> int )B )
begin
        let p := 3
        .
```

```
        .
        write B( p )
end

begin
        let b := 14 ; let c := 3

        procedure C( int a -> int ) ; b * b - 4 * a * c

        A( C )
end
```

A program requiring the full procedure closure
Figure 2

When procedure 'A' is executed in this example function 'B', which is the formal parameter, is really function 'C' in the following block. In order to execute 'C' correctly we must know about the free variables 'b' and 'c'. To do this we need both parts of the closure for 'C' to be transmitted to 'A' when 'A' is called.

The p-code implementation of Pascal [20] falls into the trap of only recording the procedure address instead of the full closure for the procedure and thus disallows the passing of procedures as parameters. A solution to the problem is given by Morrison [18].

First class procedures in relation to abstract data types.

The supporters of abstract data types [13] 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 [12], Alphard [28], Euclid [10], ML[15] and Ada [8].

None of the above languages, with the exception of ML, support first class procedures. However, as has been pointed out by Horning [7], 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 [23] and Zilles [29].

The complex number example

To explain the mechanism the following program segment written in PS-algol [3] 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 brackets. 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 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 )
        complex.number( a( rpart ) + b( rpart ),a( ipart ) + b( ipart ) ),
```

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

        proc( real a,b -> pntr )
        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 this procedure 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.

Data protection

Morris [16] 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. The solution to the problem is to check the class of the object before allowing any operation on it. Thus we can define the program's action if an impersonation does take place. 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 )
        if error( a ) or error( b ) then nil else
                complex.number( a( rpart ) + b( rpart ),a( ipart ) + b( ipart ) ),
        proc( pntr a )
        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 )
        complex.number( a,b ) )
end !of complex.arith
```

The complex number package with impersonation checks
Figure 5

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 )

        let check =  proc( pntr v -> bool )
                     if v isnt vec then { write "error" ; false }
                     else if upb( v( rep ) ) ≠ n and lwb( v( rep ) ) ≠ 1
                          then { write "dimension error" ; false }
                          else true

        if n < 2 then { write "silly dimension" ; nil }
        else vector.pack(
                proc( pntr a,b -> pntr )
                if check( a ) and check( b ) then
                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 else nil,

                proc( pntr a )
                if check( a ) do
                begin
                        write a( rep,1 )
                        for i = 2 to n do write ", ",a( rep )( i )
                end,

                proc( *real r -> pntr )
                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.

```
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

First class procedures can perform as modules

Many languages have also introduced the concept of modules Ada, Clu, ML, Modula2 [27].

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.

Partial application

Another advantage of having procedures as first class data objects is the possibility of having partially applied functions.

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 ),
        proc() ; list.start := nil,
        proc()
        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.

Separate Compilation

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

As the program utilises the persistent mechanisms of PS-algol they are reviewed here, but the reader who requires complete information should read [1,3]. 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.

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.

```
structure error.record(string error.context,error.fault,error.explain)
!A program to start a new database for someone's collection of lists
!first get the predefined module for maintaining lists.
let db := nil
repeat
        db := open.database( "library","Gigha","read" )
while db is error.record do
begin
        write "'n sorry the library is being updated"
!       wait( 5 )
end

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 his 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 his lists indexed by topic
s.enter( "topics",db2,table() )
!part apply MkListPack to ensure name always p
let his.make.lists = proc( string topic -> pntr )
                        MkListPack( mlp )( p,topic )
!preserve that for future use
structure his.list( proc( string -> pntr )h.list )

s.enter( "hisMkList",db2,his.list( his.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.

```
!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. Thus 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.

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 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.

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. The effects of a transaction are not visible to other transactions until 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.

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.

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 [2].

Figure 14 will again serve as an example. The function saved in the database as 'hisMkList' 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 his 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 he 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 his.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 { write "You have not started that topic'n"
                                     get.topic("dummy") }
                          else pack
        end

        let db = open.database( "library","Gigha","read" )
        if db is error.record do { write "Cannot open database Library",
```

```
                "'n",db( error.fault ),"'n",
                    db( error.explain ) }

        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( "hisMkList",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( "hisMkList",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

Conclusions

A number of requirements of modern programming languages, abstract types, modules, control of module assembly, separate compilation and views of data are met by the provision of first class procedures and orthogonal persistence. It has long been understood that it is desirable to be parsimonious in introducing concepts into a language design. The preceding demonstration therefore challenges language designers as to whether it is necessary to introduce a long list of concepts which can be covered by the persistent procedural mechanism.

Considering the semantic properties of languages the case for introducing different concepts rather than depending on these first class procedures appears to be weak. However, the text necessary to 'unpack' and introduce into the local environment, the interface of a module using this method leaves the question of whether extra syntactic constructs are necessary. If they are, they should probably be some general purpose shorthand (such as the patterns of ML) rather than a specific construct for modules.

Using the general properties of persistent procedures seems to have a number of advantages:

i) Software construction tools may be built within the language.
ii) The composition of separately produced software is type checked.
iii) The power of the language is much increased, for example, parameterisation is always available. The structures, interrelationships and naming rules which may be constructed are extremely flexible.

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.

Acknowledgements

This work was supported in part by SERC grant GRA 86541 at the University of Edinburgh. It is now supported at Edinburgh 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. The paper was partly written while Malcolm Atkinson was visiting the University of Pennsylvania, Philadelphia and Ron Morrison the Australian National University, Canberra.

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., Chisholm, K.J. & Morrison, R. Progress with persistent programming, in Database - role and structure, Cambridge University Press, Cambridge, 1984.
3. Atkinson, M.P., Bailey, P.J., Cockshott, W.P. & Morrison, R. PS-algol reference manual. Universities of Edinburgh and St Andrews PPR-8 (1984).
4. Birtwistle, G.M., Dahl, O.J., Myrhaug, B & Nygaard, K. SIMULA BEGIN. Auerbach (1973).
5. Cockshott, W.P., Atkinson, M.P., Bailey, P.J., Chisholm, K.J. & Morrison, R. The persistent object management system. Software, Practice & Experience 14 (1984).
6. Evans, A. PAL a language designed for teaching programming linguistics. Proc. ACM 23rd. Nat. Conf. Brandin Systems Press (1968), 395-403.
7. Horning, J.J. Some desirable properties of data abstraction facilities. ACM Sigplan Notices 11 (1976), 60-62.
8. Ichbiah et al,. Rationale of the design of the programming language Ada. ACM Sigplan Notices 14, 6 (1979).
9. Johnston, J.B. A contour model of block structured processes. ACM Sigplan Notices 6, 2 (1971).
10. 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.
11. Landin, P.J. The next 700 programming languages. Comm.ACM 9, 3 (1966), 157-164.
12. Liskov, B.H., Synder, A., Atkinson, R. & Schiffert, C. Abstraction mechanisms in CLU. Comm.ACM 20, 8 (1977), 564-576.
13. Liskov, B. & Zilles, S.N. Programming with abstract data types. ACM Sigplan Notices 9, 4 (1974), 50-59.
14. McCarthy, J. et al. Lisp 1.5 Programmers manual. M.I.T. Press Cambridge Mass. (1962).
15. Milner, R A proposal for standard ML. Technical Report CSR-157-83 University of Edinburgh. (1983).
16. Morris, J.H. Protection in programming languages. Comm.ACM 16, 1 (1973), 15-21.
17. Morrison, R. S-algol language reference manual. University of St Andrews CS/79/1 (1979).
18. Morrison, R. A method of implementing procedure entry and exit. Software Practice and Experience 7, 5 (1977), 535-537.
19. Naur, P. et al. Revised report on the algorithmic language Algol 60. Comm.ACM 6, 1 (1963), 1-17.
20. Nori, K.V. et al. The Pascal P compiler implementation notes. Technical Report, 10 Zurich (1974).
21. Randell, B. & Russell, L.J. Algol 60 Implementation. Academic Press (1964).
22. 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.
23. Strachey, C. Fundamental concepts in programming languages. Oxford University (1967).
24. Turner, D.A. SASL language manual. University of St.Andrews CS/79/3 (1979).
25. Wirth, N. & Weber, H. EULER a generalisation of algol. Comm.ACM 9, 1 (1966), 13-23.
26. Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.
27. Wirth, N. Programming in Modula-2 : Second Edition. Springer-Verlag, Berlin, 1983.
28. 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.

29. Zilles, S.N. Procedural encapsulation : a linguistic protection technique. ACM Sigplan Notices 8, 9 (1973).
30. van Wijngaarden, A. et al. Report on the algorithmic language Algol 68. Numerische Mathematik 14,1 (1969), 79-218.