G. KIRBY

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

PS-algol Abstract Machine Manual

Persistent Programming Research Group

Persistent Programming
Research Report 11

PS-algol Abstract Machine Manual

Persistent Programming Research Group

Department of Computing Science
University of Glasgow
14, Lilybank Gdns
Glasgow G12 8QQ.

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

Persistent Programming Research Report 11

Preface

This manual presents the design of the PS-algol abstract machine that is used in distributed copies of PS-algol released in early 1985. It is hoped that the report will be of help to people considering porting PS-algol to new machines as well as to people wishing to understand our work. However, people intending to port PS-algol should contact the Persistent Programming Research Group for information on reports of porting in progress and for details of changes in the abstract machine.

This report describes the abstract machine which is the basis of the present interpreter written in C, and running on the DEC VAX computers running Berkeley UNIX 4.2 and on the ICL PERQ computers running PNX2. An implementation is in progress for the ICL 2900 and a code generator for the VAX is being constructed.

There are substantial changes in the abstract machine from earlier versions, and people with earlier information should regard it as superseded by this report.

The reader of this report is advised to read it in conjunction with PPR12 "The PS-algol Reference Manual : second edition" and PPR13 "The CPOMS Persistent Object Management System" which all describe different aspects of the same release of PS-algol and which will be published at the same time.

For further information please contact the addresses given above.

Contents

1 Introduction

The PS-algol abstract machine is a refinement of the S-algol abstract machine[1] but differs from it in four important aspects. Firstly PS-algol supports first class procedures which means that simple objects may be encapsulated in a procedure suspension long after they have gone out of scope. A consequence of this is that these objects must be kept on the heap and not the stack. However the recursive nature of the language lends itself naturally to implementation using a stack and thus the PS-algol machine simulates a stack using separate heap objects for each stack frame. Again because of the encapsulation the stack frame models a block and not a procedure as in S-algol i.e. the system uses block level addressing.

The second major difference between the PS-algol machine and the S-algol machine is the instruction code format. The PS-algol machine uses a byte orientated system with most instructions having long and short forms for different lengths of operands.

The PS-algol system also supports persistence. This does not require major changes to the abstract machine itself but to the facilities the machine provides and the software necessary to support the system.

Finally the PS-algol machine uses a display mechanism to identify the free variables of an environment. The display is kept on the pointer stack(see later) and is used by the stack load and stack assign instructions.

2 Abstract Machine Registers

The registers of the PS-algol machine are used to identify the local and standard frames on the heap as well as the top of the local main and pointer stacks and the code pointer. Since there is a main stack and a pointer stack for each environment there are two stack registers for each environment. The registers are

CP	code pointer
LMSP	local frame main stack top
LPSP	local frame pointer stack top
LMSB	local frame main stack base
LPSB	local frame pointer stack base
SMSB	standard frame main stack base
SPSB	standard frame pointer stack base

3 Heap Formats

All data objects are kept on the heap. This includes code for procedures and the main program and therefore the only dynamic storage system that need be supported is a heap system with a garbage collector. PS-algol is a block structured language that is well suited to a stack implementation technique. The action of the stack is simulated within the heap to implement the block structure.

To accommodate persistence the system requires object level addressing. However we must be careful that pointers in the system point to objects only and not to sub components of an object such as the field of a structure. Thus parts of an object always have a two part address.

The heap items are strings, vectors, images, structures, files, code vectors and frames. Their heap layout is as follows in the VAX and PERQ implementations. Note that bit 31 of a word is the most significant.

3.1 Headers

Bits 28-31 of the first word of a heap item (referred to as word 1 below) identify the type of heap object. The following are possible

value	**object**
0	reserved for compacting garbage collector
1	string
2	file
3	structure
4	vector of pointers
5	vector of procedures (closures)
6	vector of integers or booleans
7	vector of reals
8	frame
9	code vector
11	image

Thus the markers in the first word look like.

bits 28-31	identification of heap object
bit 20	mark bit for the garbage collector
bit 19	written bit for persistence object manager
bit 18	reserved for persistence object manager
bits 16-17	reserved for future use
bits 0-15	as defined below

3.2 Strings

word 1	bits 0-15	number of characters in the string followed by the characters 1 per byte padded up to a 4 byte boundary.

3.3 Files

word 1	bits 0-15	size in bytes of the file object including the buffer. This is padded to a 4 byte boundary.

3.4 Structures

word 1	bits 0-15	size of the object in 4 byte elements.
	bits 21-27	number of pointers including the class identifier
word 2		The class identifier (a pointer to a string) followed by the fields with the pointer fields first. The compiler performs this rearrangement.

3.5 Vectors

word 1	bits 0-15	not used
word 2		lower bound
word 3		upper bound
		followed by the elements

3.6 Frames

word 1	bits 0-15	size of the frame in bytes padded to a 4 byte boundary.

```
      -------------------------------------------
      |                                         |
----------------------------------------------------------------------
| H |   |          |            | D |   |   |   |   |   |   |   |    |
| E | P |          |            | S |   |   |   |   |   | P | M |    |
| A | S |---->LMSP | LPSP <--   | P | @ | S | D | R | L | S | S | LL |
| D | L |          |            | L |   | L | L | A | N | P | P |    |
|   |   |          |            | Y |   |   |   |   |   |   |   |    |
----------------------------------------------------------------------
^                                               ^
|                                               |
LMSB                                          LPSB
```

The frames model the environments. There are two stacks, the main stack which grows forwards and the pointer stack which grows backwards.

The compiler will calculate the maximum size of each stack and when set up LMSP points to after PSL and LPSP points to before the display. Note that the stacks grow in opposite directions and that the dynamic link(DL), static link(SL) and procedure address(@) are on the pointer stack.

In order that this frame may call others and return correctly we require to store the pointer stack link(PSL), the return address(RA) of the calling frame, the line number(LN) where called from and the values of LMSP and LPSP in MSP and PSP at the point of calling. So that pointers do not point into the middle of objects RA is relative to the start of the code vector of the calling procedure. The full address can be found by using the dynamic link and the procedure address in that frame. For the same reason LMSP and LPSP are saved as offsets in the same manner. PSL is relative to the start of the frame.

The first elements after the closure on the pointer stack make up the display of the frame. There is one display entry for each lexical entry outside the lexical level of the frame (i.e. LL-1) with the outermost lexical level being next to the procedure's address (@). This is used by the load and global instructions to find non local frames. Furthermore the frame records its own lexicographic level (LL) in order that it may be used to find the size of the display when entering procedures and blocks.

3.7 Code Vectors

word 1   bits 0-15   size in bytes padded to a 4 byte boundary

```
                 ---------------------------------
                 |                               |
--------------------------------------------------------------------------
| H |   |   |Max.ms|          | Vector of |            |                 |
| E | V | V |------|          | procedure |   Vector   |     String      |
| A | P | S |Max.ps|   Code   |  closures |     of     |    Literals     |
| D |   |   |      |          |           |   Strings  |                 |
--------------------------------------------------------------------------
      |                       ^           ^
      |                       |           |
      -------------------------           S
```

Associated with each code vector there may be a vector of procedure closures, one for each procedure declared inside this one, and a vector of strings which contains the string literals used by this procedure. Note that the vectors of closures and strings and the strings themselves are all separate heap items. The code vector contains pointers to these vectors(VP,VS) and the maximum size of stacks that it uses(Max.ms,Max.ps) in stack elements.

On initial loading all the code is brought in as one contiguous chunk. The string literal addresses are relative to S and the procedure addresses in the closure are relative to the start of the unit of compilation. The addresses must be fixed up on initial loading. If a code vector is brought in from the database the addresses will be fixed up automatically by the persistent object manager. The static links in the procedure closures are null initially. VS and VP are initially relative to the start of the code vector.

3.8 Images

```
-------------------------------------------------------------------
| h |        |         |          |          |        |         |
| e |        |         |          |          |        |         |
| a |bitmap  | file    |    X     |    Y     |   X    |    Y    |
| d |vector  |descptr  |  offset  |  offset  |  dim   |   dim   |
| e | ptr    |         |          |          |        |         |
| r |        |         |          |          |        |         |
-------------------------------------------------------------------
```

word 1	header
word 2	pointer to the vector of bitmap vectors
word 3	pointer to the file descriptor (if necessary)
word 4	X offset into the bitmap vector
word 5	Y offset into the bitmap vector
word 6	X dimension of the image
word 7	Y dimension of the image

The bitmap vector is kept as a vector of integers laid out as follows. There is one of these for each plane in the image.

word 1	header
word 2	lwb
word 3	upb
word 4	X dimension of the bitmap
word 5	Y dimension of the bitmap
word 6	offset to start of the image from the start of the object.
word 7	number of scan lines in this page.
words 8 -	bits

4 The Abstract Machine Code

The PS-algol abstract machine code, PS-code, is designed to fit exactly the needs of the PS-algol language. Appendix II describes the code generated for each syntactic construct in the language and Appendix III the operation codes and format of each abstract machine instruction. Here the individual instructions are described. They fall naturally into groups. Typed instructions have an encoded name with the following convention.

ib	integer or boolean
r	real
s	string
p	pointer, picture or image
pr	procedure
v	void

Reals, integers and booleans reside on the main stack all other objects on the pointer stack.

Jumps

All the jump addresses are relative to the location following the jump address. Only backward jumps have a short form. In the following instructions L is a byte offset from the next instruction in line.

fjump(L)	unconditional jump forward to address L.
bjump(L)	unconditional jump backwards to address L.
jumpf(L)	jump forward to L if the top stack element is false. Remove the top element of the stack.
jumptt(L)	jump forward to L if the top element is true. Otherwise remove the top stack element.
jumpff(L)	jump forward to L if the top stack element is false. Otherwise remove the top stack element.
cjump.ib,r,s,p,pr(L)	The type determines which stack to use. If the top two stack elements are equal, remove both and jump forward to L. Otherwise remove only the top stack element. Be careful on equality of strings and procedures.

bjumpt(L)	jump backwards to L if the top stack element is true. Remove the top element of the stack.
fortest.op(L,MS,PS)	The control constant, increment and limit are the top three elements of the stack. If the increment is negative and the control constant is less than the limit or the increment is positive and the control constant is greater than the limit then remove all three from the stack and jump forward to L. Otherwise create a frame using MS and PS to calculate the size. It is in this frame that the body of the loop will execute. This part of the instruction is like block enter. The control constant has to be copied into the new frame and the display is calculated.
forstep.op(L)	Perform a block exit. Update the control constant by adding the increment which is still on the stack. Then jump backwards to L.

Procedure and Block Entry and Exit

The instruction sequence to call a procedure is

load closure
evaluate the parameters
apply.op

apply.op(ms,ps)	ms and ps are the number of words of parameters on each stack. Use the closure under the pointer stack parameters to find the number of elements of each stack that this procedure requires. This is kept at the beginning of the code vector. Claim the space from the heap. Fill in the header, PSL,RA,LN and DL. Store the old values of LMSP and LPSP in the calling frame. Copy the closure to the new frame. Make up the display and then copy the parameters. The display is formed by copying the display of the frame pointed at by the static link and adding the new static link. The number of display items can be found from LL in that frame. Remember to generate the DL. Set up the new values for LMSP, LMSB, LPSP and LPSB. Transfer

control. Any procedure with a null static link is a standard function written in C and implemented in the interpreter. This may be implemented by a jump table for each standard function reflecting the standard function number generated by the compiler.

return.ib,r,s,p,pr,v	Return from a procedure. This must get rid of the old block. Move the result to the old stack top. Move DL to LMSB. Calculate the return address. Use the old frame to set up LPSB, LMSP and LPSP. The dynamic link of the returning procedure should be nulled out to prevent retention of the dynamic history over garbage collection and commit.
block.enter(MS,PS)	Enter a block. MS and PS are the stack sizes required by the frame. Enter this as if it were an in line procedure. The display is the display of the enclosing frame plus the current static link. The lexical level goes up one.
block.exit.ib,r,s,p,pr,v	Exit from a block. Same as return.
store.closure(n)	This is the nth procedure in the vector of closures for the current procedure. Load the closure from the vector on to the stack. Remember to overwrite the static link with the current value of LMSB and that these are pointer stack values.

Stack Load Instructions

To ease the problems of garbage collection the frame models two stacks. The main stack contains space for integers, reals and booleans while the pointer stack contains space for all the pointer objects. The pointer stack is used as the base for marking the heap. Objects on the pointer stack may be strings, vectors, pntrs, files, images, frames and code vectors. Reals take two stack elements each on the main stack as do procedures (closures) on the pointer stack. A closure is made up of the code vector address and the static link. These instructions are used to load any data item that is in scope, on to the top of the stack. The data items may be in the local, global, standard or intermediate environments and a separate instruction exists for each form. Different instructions

are also used for the separate stacks. The local, global and standard forms of the instruction have a parameter which is the displacement of the item from the stack base. The intermediate form of the instruction requires the address of the static environment as well as the displacement. This is kept in the display and is given as an offset from LPSB. The global environment is always the first entry in the display. Only one form of each type is described.

local(n),global(n),stand(n),load(r,n)	load on the main stack
dlocal(n),dglobal(n),dstand(n),dload(r,n)	load double length item on to the main stack
plocal(n),pglobal(n),pstand(n),pload(r,n)	load on the pointer stack
dplocal(n),dpglobal(n),dpstand(n),dpload(r,n)	load double length item on to the pointer stack

Stack Assignment

For each of the stack load instructions there is an equivalent stack assignment instruction. These instructions take the top element of the stack and assign it to the address indicated in the instruction.

Comparison Operations

The comparison operations act on the data types int, real and string. The top two elements of the stack are compared and removed. The boolean result true or false is left on the main stack. Care should again be taken in the comparison of strings which means element by element comparison. Equality is defined on all the data objects in the language. There is a separate form of the instruction for each type.

ge.i,r,s	greater than or equal to
gt.i,r,s	greater than
le.i,r,s	less than or equal
lt.i,r,s	less than
eq.ib,r,s,p,pr	equal to
neq.ib,r,s,p,pr	not equal to

Arithmetic Operators

These instructions operate on the data types real and integer. The top two elements of the stack are replaced by the result except for negate and float1 which use only the top element and float2 which uses the second top element.

plus,fplus	add
times,ftimes	multiply
minus,fminus	subtract
fdivide	divide real
div	divide int leaving quotient
rem	divide int leaving remainder
neg,fneg	negate
float1	coerce the int to a real on top of the stack
float2	coerce the int to a real second top stack element

Vector and Structure Creation Instructions

These instructions take information off the stack and create heap objects. These objects are then initialised and the pointer to them left on the top of the pointer stack.

make.vector.ib,r,s,p,pr(m)	For i,b,r m is an offset from LMSP to the lwb and the element values are above the lwb on the main stack. For s,p,pr m is an offset from LPSP indicating the extent of the elements. The lwb is the top element of the main stack.
iliffe.op.ib,r,s,p,pr(n)	n pairs of bounds are on the main stack. However, the top of one of the stacks will contain the initial value. The instruction creates an iliffe vector of the shape indicated by the bound pairs and the value of the initial expression is copied into the elements of the last dimension. The expression value and the bound pairs are removed from the stack and the pointer to the vector is placed on the pointer stack.
form.structure(m,n)	The expressions which initialise the structure fields have been evaluated on the appropriates stacks. This includes the class identifier which is a pointer to a string. m is the total size of the object on the heap and n is the number of pointers. After moving the fields from the stacks the pointer to the structure is placed on the pointer stack.

Vector and Structure Accessing Instructions

These instructions are generated by the compiler to index a vector or a structure. The index of the vector must be checked against the bounds before the indexing is done. Similarly the structure class of a structure must be checked.

subv.ib,r,s,p,pr	The vector index is on the top of the main stack and the vector pointer on the pointer stack. These are used to check that the index is legal and then to find the required value. They are removed from the stacks. The resultant value is put onto the relevant stack.
subs.ib,r,s,p,pr	The class identifier and the structure pointer are on the top of the pointer stack. The field address, an offset from the start of the structure, is on the main stack. The class identifier is checked against the structure class identifier and if it is the same the field address is added to the pointer to yield the absolute field address. The class identifier, field address and the structure pointer are replaced on the stack by the result.
subvass.ib,r,s,p,pr	This assigns a value to a vector element. The value is on top of the appropriate stack and the address is calculated as in subv.
subsass.ib,r,s,p,pr	This assigns a value to a structure field. The value is on top of the appropriate stack and the address is calculated as in subs.
lwb	remove the pointer to the vector from the pointer stack and place its lower bound on the main stack.
upb	remove the pointer to the vector from the pointer stack and place its upper bound on the main stack.
is.op	The class identifier is on the pointer stack and is

compared with the class identifier of the structure pointed at by the 2nd top element of the pointer stack. Remove both and place the boolean result of the comparison on the main stack.

isnt.op	This is the same as is.op except it has the opposite test.
load.trademark(m)	This loads the address of the class identifier on to the pointer stack. m is the index into the string literal vector of the current procedure.

Image operations

form.image	The pixel and the X and Y dimensions of the image are on the main stack. Create an image descriptor containing a pointer to a vector of bitmap vectors which must also be created. The dimension of the vector of bitmap vectors is obtained from the initialising pixel. The bitmap vectors have each pixel initialised to the value of the initialising pixel. Leave the descriptor to the image on the pointer stack.
limit.2	The top 2 values on the main stack are the X and Y offsets for the limit. A pointer to an image descriptor is on the pointer stack. Create a new image descriptor copying the bitmap and file fields from the old one. Add the X an Y offsets to the offsets in the old descriptor and place the result in the new descriptor. Calculate the dimensions of the new image and place these in the descriptor. Leave the pointer to the new descriptor on the pointer stack.
limit.4	This is like limit.2 except that in addition to the X and Y offsets being on the main stack there are also the X and Y dimensions of the new image. Using these values form a new image descriptor and place the result on the pointer stack.

subimage.op	The 2nd top element of the main stack indicates the start plane of the new image (numbered from 0) and the top element the number of planes. After checking the bounds with the image given at the top of the pointer stack replace it by the new image descriptor. Like limit this creates an alias.
subpixel.op	The pixel specified at the 2nd top of the main stack is indexed by the top element index. After checking that the index is legal replace the two elements by the new single depth pixel.
formpixel.op(n)	n is the number of individual pixels that are on top of the main stack and must be replaced by one pixel representing them. The zeroth plane is the nth top element and the n-1th plane the top element.

raster.copy,not,rand,ror,nand,nor,xor,xnor

These instructions result in almost the same code being executed. The only difference between them is the raster function actually used. The destination image is on top of the pointer stack and the source the second top. Clipping is performed according to the destination image descriptor. Notice that the image can be a bitmap or a file(for the screen). An image is a bitmap if the file descriptor is a nullfile pntr.

Load Literal Instructions

These are used to load the value of a literal on to the stack. The literal usually follows the instruction in the code stream and so the CP register has to be updated accordingly.

LL.nil.pntr	load the pntr value **nil.**
LL.nil.pr	Create the null procedure closure. This involves creating a dummy code vector so that equality of different nullproc's will give false.
LL.nil.string	load the empty string on to the pointer stack.
LL.file	load the nullfile on to the pointer stack.
LL.bool(n)	load the boolean value n (true or false) on to the

	main stack. (0-true,non-zero-false)
LL.sint(n)	load the value of a short integer (-128 to 127) on to the main stack.
LL.real(n)	load the real on to the main stack.
LL.string(n)	load the string address on to the pointer stack n is the index of the string in the string vector of the current procedure.
LL.lint	load a long integer, 32 bits, on to the main stack.
LL.char(n)	load the character n as a string of length 1.

String Operations

These are used to perform the string operations in PS-algol.

concat.op	remove the two strings from the top of the pointer stack and replace them with a new string which is the concatenation of them.
substr.op	A new string is created from the one at the top of the pointer stack and replaces it. It is formed by using the length at the top of the main stack and the starting position at the second top. After checking that these are legal they are removed.

Input and Output

read.op(n)	the stream descriptor is on the top of the pointer stack. This is removed and the value read is placed on the appropriate stack. n indicates which read function to use. They are
0 read	read a character and form it into a string.
1 reads	read a string.
2 readi	read an integer.
3 readb	read a boolean.
4 readr	read a real.
5 read.name	read a PS-algol identifier and form it into a string. This takes a seed from the top of the stack.

6 peek	same as read but do not advance the input stream.
7 read.byte	read an 8 bit byte and return it as an integer.
8 read.a.line	read from the current input position up to and including the first newline character. Return a string of the characters excluding the newline.
9 eoi	test for end of input on the input stream.
write.op(n)	The field width is either on the top or second top of the main stack and the item to be written out either under it or on the pointer stack. The stream descriptor is under all this on the pointer stack. The field width and the item are removed from the stack. In the case of out.byte the file descriptor is also removed from the stack. n indicates which function. They are
0 write.int	write an integer in the field size specified at the second top of the stack with s.w (at stack top) spaces after it.
1 write.bool	write a boolean in the field size specified. If the field size is -1 or less than the size of the boolean then write it out in the minimum space required. Booleans are written out as either "true" or "false".
2 write.string	write a string in the field size specified. If the field size is -1 or less than the size of the string then write it out in the minimum space required.
3 write.real	write a real in the field size specified at the second top of the stack with s.w (at stack top) spaces after it.
4 out.byte	write an 8 bit byte.

Miscellaneous

rev.ms,rev.ps	Swap the top two elements of the stack
erase.ib,r,s,p,pr	remove an element from the stack

finish.op	stop the program execution
abort.op	stop the program execution
not.op	perform a not on the boolean at the top of the stack
newline(n)	set the line indicator to n.

5 Loading and Initialisation Code

5.1 Initialisation

The initialisation sequence for the PS-algol interpreter may be split into 12 sections. These are listed below.

i) enable interrupt handling
ii) heap initialisation
iii) create single character strings
iv) create PS-algol vectors for the command line and environment vectors
v) read in and initialise user code
vi) read in and initialise POMS code
vii) tidy up heap initialisation
viii) create standard identifier frame
ix) create a unified class identifier vector
x) initialise the interpreter's database routines
xi) execute the POMS code to initialise the standard frame
xii) execute the user code

Note the order in which the interpreter executes these sections is up to the implementor. The order given above reflects that of the interpreter written at St. Andrews. Note also that the POMS code mentioned above is a PS-algol program that when executed initialises the standard frame with the standard functions written in PS-algol. Each section will now be explained in more detail.

5.2 Interrupt Handling

Interrupts can occur for several reasons, e.g. arithmetic errors, terminal hangup, memory addressing errors and forced program termination. In all these cases it is important for the interpreter to process interrupts so that it can if necessary terminate program execution cleanly. Program termination for interrupts such as arithmetic errors will result in an error message whereas other interrupts such as terminal hangup will not. In addition to printing any error messages the interpreter must also ensure the database locks held by the program are released.(see Chapter 6).

5.3 Heap initialisation and tidy up

The heap provides the interpreter with dynamic storage for the PS-algol data structures. To initialise the heap it is simply necessary to obtain a section of contiguous memory and initialise a pointer to the first free address and the first illegal address (see chapter 9). Having initialised the heap the interpreter's storage allocation package can now be used. However the garbage collector should be disabled during initialisation (except for part xi see later) since no space can be freed and the information necessary to perform the marking phase of a garbage collection will not be available.

Once the user and POMS code has been read in all the space already allocated on the heap will never be reused and will never contain pointers to the rest of the heap. This information should be used to reduce the area of the heap over which the garbage collector need operate.

5.4 Single character strings

Purely for efficiency considerations - reducing heap traffic; speeding up single string comparisons, it has been found advantageous to provide the run time system with a set of single character strings within the heap. This means that PS-algol I/O routines such as **read**, **peek** and the machine instruction LL.char need not go through the storage allocation process. Instead the address of one of these single character strings is calculated (normally one or two machine instructions). This makes execution faster and also helps reduce heap usage. Notice that substr.op and concat.op should make use of this facility.

5.5 Command line and environment vectors

It is undesirable for the standard functions options and environment to have to calculate their respective vectors on every call. To avoid this overhead the vectors are created during initialisation and pointers to them saved for later use. The environment vector is specific to UNIX, it provides a form of logical name mapping.

5.6 User code

A PS-algol code file consists of a sequence of one or more code vectors followed by the class.id vector followed by padding characters followed by 18 bytes of information. For further details of this layout see Appendix I.

All references in the code to the head of heap objects have to be specified by the compiler as offsets (the compiler cannot generate absolute addresses). It is required that these references be made

absolute before interpreting the code. There are four sets of such references:

i) a code vector may reference two vectors as described above. A null reference is indicated by zero.

ii) a procedure closure vector will reference code vectors within the code file. It is only necessary to make absolute the procedure address. This is accomplished by adding into the offset provided the base address of the code file in memory. The static link remains null at this time.

iii) a string vector will reference the string literals immediately following it. To make these absolute the base address of the string vector is added to the offset provided.

iv) the class.id vector contains the address relative to the start of the unit of compilation of every class identifier. These addresses must be made absolute on loading.

Having read in the code, checked the version number, made absolute the reference offsets it is only required to remember the starting address for the user code file for initialisation to be complete. The start address is a pointer to the first code vector to be executed.

5.7 POMS code

The reading in of the POMS code is identical to that for the user code as is its initialisation. Again it is necessary to remember the starting address. This code will later be executed to initialise part of the standard frame (see Chapter 7).

5.8 Standard identifier frame

This frame provides an environment for all the pre-declared identifiers in PS-algol e.g. sin,cos etc. A description of this frame is provided by a standard declaration file that is used by the compiler to allocate addresses to the pre-declared identifiers (see appendix IV). This file should be used in the same way by the implementor to ensure that the interpreter initialises the standard frame correctly. Among the values to be initialised by the interpreter are the real literals, standard I/O files and interpreter implemented standard procedures. The interpreter implemented standard procedures can be identified by having a zero static link in their closures. Any procedure with a static link of zero can then be treated specially at procedure application. A number of pre-declared procedures are written in PS-algol. All such procedures are initialised by executing the POMS code. The registers SMSB and SPSB are initialised at this point.

5.9 The Class identifier vector

Every PS-algol code file contains a class identifier vector, hence during initialisation there are two class identifier vectors, one for the user code and one for the POMS code. These two vectors are combined for the use of the database routines. Two uses are made of the combined vector, namely allocation of reserved pids to structure trademarks and optimisation of structure class comparisons. When a structure trademark is brought in from a database the class identifier vector is consulted. If the trademark is in the vector the pointer in the vector is used otherwise the trademark is added to the vector. Using this technique checking of trademarks should succeed without having to compare characters.

5.10 Initialising the C-POMS

Several tables are held by the interpreter to perform address mappings from pids to local addresses and from pids to disk addresses. These tables are initialised with a set of reserved pids for some structure trademarks, interpreter implemented standard procedures, single character strings, error message strings and the nil pointer literal. In addition a set of registers are initialised that will point to a list of the databases accessed by the interpreter. This list is required for the marking phase of the garbage collector (see Chapter 8).

5.11 Executing the POMS code

To initialise the PS-algol procedures in the standard frame the POMS code must now be executed. A global frame is first created for the POMS code and then the local stack registers are initialised. The size of the frame is encoded in the code vector pointed to by the POMS code start address. The interpreter's decode loop is then called to execute the POMS code from byte 16 (the start address) of this code vector. If the execution of the POMS code terminates normally the decode loop will return and initialisation continue. If it does not return the interpreter should have terminated with a suitable error message. Garbage collection is enabled before the decode loop is called and disabled after.

5.12 Executing the user code

The final step of the initialisation is to construct a global frame for the user code and enable the garbage collector. This last step and execution of the user code is identical to that for the POMS code.

6 Persistence Hooks

6.1 Interpreter support for the C-POMS

A program module within the interpreter implements the persistent object management system, called the C-POMS since it is written in C. It is interfaced to by means of private standard functions or by several internal functions. The private standard functions are used by the PS-algol POMS code to implement a higher level database interface for the user. In return for the database interface the interpreter must provide the C-POMS with the heap allocation routines and a pointer stack to allow it to preserve its pointers. The internal functions and the heap interface will now be described, the private standard functions are described in appendix IV.

6.2 Internal access to the C-POMS

The internal access to the C-POMS is necessary for translating PIDs, releasing database locks at program termination and for restoring one of the C-POMS address tables after a compacting garbage collection.

PID translation is done by an illegal address routine that returns the heap address of the PID's object. This is done by first looking up the PID to address table and if necessary copying the object in to the heap from its database. PID translation and testing if a pointer needs translating can be quite expensive so it is done as few times as possible. This is achieved by the interpreter using the following rules. Once a PID is translated the PID is overwritten with its corresponding local address so reducing the amount of PID translation. Note that this overwritting is most effective if it is at the source of the PID, that is in the field of the structure rather than on the stack. Secondly when illegal address is called for a code vector the vector's two pointers (VP,VS) are converted to heap addresses so that dereferences of its procedure and string vectors need not test if these pointers are PIDs. If a PID for a frame is translated the frame's display is converted to local addresses to avoid checking for a PID on every use of the display. Whenever a pointer value is first placed on the local stack frame it will be converted to a local address. To complete the set of rules dereferences of pointer objects in vectors and structures cause a test for a PID and possible translation.

These rules are sufficient to guarantee no PIDs are encountered on the frame of an executing procedure or block. If a frame does have a PID

in it then it must have been imported from the persistent store and so cannot belong to an executing procedure or block. Consequently instructions that operate on the local stack frame need never deal with PIDs.

Database locks need to be released at program termination. A call to the routine provided by the C-POMS should therefore be inserted in the code that performs program termination.

The C-POMS maintains two hash tables for mapping PIDs to addresses and vice versa so, if a compacting garbage collection occurs the address indices may be invalidated. A routine is provided by the C-POMS to enable the garbage collector to rethread the address to PID hash table.

6.3 Heap support for the C-POMS

As mentioned above the interpreter must supply the C-POMS with the heap allocation routines. However this is not sufficient as some of the C-POMS routines, particularly commit and open database are highly recursive and must be able to preserve their pointers over garbage collections. To overcome this a special stack should be provided for the C-POMS for it to keep its pointers on. This can be easily done using a vector of pointers with a register to point to the top stack element. Routines should also be provided to alter the stack size in use. Null pointers could be used to fill unallocated stack elements.

7 Bootstrapping POMS

7.1 Bootstrapping the C-POMS

In order to run the C-POMS, the interpreter must be given a directory in which to keep all its files. This should be an empty directory since the C-POMS will create any files it uses if necessary. In a UNIX environment a shell variable PSDIR is used by the interpreter to find the name of the directory. To discourage access to databases from outwith the C-POMS the supplied directory should be made accessible only to the PS-algol system. In a UNIX environment this can be done by creating a PS-algol user to manage the PS-algol system. This user owns the PS-algol interpreter and has sole access to the C-POMS directory. It is then only possible, using the set user id facility, for normal users to access databases if they are running the PS-algol interpreter, i.e. only if they are running PS-algol programs.

7.2 Bootstrapping the PS-algol POMS code

The normal PS-algol user is supplied, via the standard frame, with a set of database procedures that assume a table as the root object in a database, a set of table handling procedures, a real I/O package, raster graphics utilities and the outline graphics procedures. When the interpreter starts up none of these procedures are available although the standard frame has space reserved for them. Any attempt to use them before they are initialised will result in a run-time error.

There are two possible methods of initialising the standard frame with these procedures. Firstly the procedures may all be declared in the PS-algol initialisation program that is run before the user program. Indeed this was the original scheme for providing the tables and real I/O procedures. However as the number of procedures and their complexity has grown this method has become undesirable. Some of its disadvantages include having to recompile a large number of procedures when one is changed, a large amount of code must be placed in the heap that may never be used and more importantly it is not possible to allocate reserved pids to the procedures.

The alternative method that has been adopted is to make the PS-algol initialisation program import from a system database all the necessary procedures. In this way all the procedures and their objects are given unique pids and may be modified independently of each other. Each set of

procedures is then maintained by a program that places its procedures in the system database. An important feature of the initialisation program is that if it can't find a particular set of procedures, the initialisation continues leaving them uninitialised. This is necessary to allow the construction of the system database by a PS-algol program.

7.3 Constructing a PS-algol system database

As a result of implementing the basic database routines in the interpreter it is possible to manipulate databases independently of tables. A description of the relevant standard functions can be found in appendix IV. The first step in constructing the system database is to create the database using the 'createdb' private standard function. A set of table handling procedures can then be declared and used to create a table containing themselves. The root of the database resulting from the 'createdb' function is then made to point to this table. If a commit is done at this point then the system database will have a copy of the table routines in a table. Once this stage is complete all the high level database functions are available to future programs.

To ensure that the table procedures are available it is necessary to be able to interrogate a table without access to the table handling functions. This is accomplished by ensuring that when a table is created it is represented by a structure of procedures that operate on it. Hence if the structure class for a table is known the set of table handling procedures for it can be accessed. The initialisation program must therefore know the structure class of the table pointed to by the root of the system database. Assuming the table is of the appropriate form the initialisation program can extract all the procedures it holds and place them in their positions in the standard frame. Having extracted the table handling procedures the high level database functions can then be defined.

The next few steps involve running programs in which the other required procedures are declared and entered into the table in the system database using the high level database procedures. Again the structures in which these procedures are placed must be incorporated into the initialisation program.

8 Marking for Garbage Collection

Once it is impossible to satisfy a given request from the heap, it is neccessary to initiate a garbage collection sequence which will make available for re-use, heap objects which are no longer reachable. In order to distinguish garbage from wanted items all those items which are wanted need to be marked as such. This is where the marker comes in.

The marker is a routine which given an initial heap item will mark it and then recursively mark any other heap items which it references (see Chapter 9). There are basically seven distinct types of heap item, these being

i) strings,files
ii) structures
iii) vectors of pointers
iv) vectors of non-pointers
v) code vectors
vi) frames
vii) images

There now follows a discussion of the intricacies of marking these types of heap item.

8.1 Strings and Files

Strings and files contain no pointers to other heap objects, so the marker need go no further.

8.2 Structures

An instance of a structure contains a count of the number of pointer fields within its body. This value must always be greater than zero as there is always present a pointer field to the structure's class identifier which is held as a string. The class identifier field is the first word after the header word of the structure. Any other pointer fields occur immediately after the class identifier field. This rearrangement of the structure fields is handled automatically by the compiler and is transparent to the user. When marking a structure therefore, it is simply a case of iterating through the number of pointer fields.

8.3 Vectors of pointers

This category consists of vectors of procedure closures, pointers,

images and strings. The number of pointer elements is calculated from the vector bounds held in words two and three of the vector header. The elements follow the bounds. In the case of a vector of procedure closures it is important to remember that each element consists of two pointers - a pointer to the code vector of the procedure and a pointer to a frame ie. its static link.

8.4 Vectors of non-pointers

This category consists of vectors of integers, pixels, booleans and reals. As with strings all that needs to be done with these is to mark them.

8.5 Code vectors

Code vectors contain two pointers - one to a vector of procedure closures and the other to a vector of strings. These are found in the two words following the code vector header word. Again simply iterate through these two pointers.

8.6 Frames

Each instance of a frame contains a pointer stack. The base of this stack may be obtained by adding the offset found in word two of the frame to the base address of the frame. The third word after the pointer stack frame base (psfb) contains an offset from the psfb which indicates the pointer stack top address (pst). By iterating from the pst to the psfb all items referenced by the frame will be marked.

A useful aid to marking is to implement null pointers as zero. In fact the compiler does this when creating code vectors. If a given procedure has no procedures declared within it then the value for the procedure vector address in the code frame will be zero. Similarly, for the string vector address, if the procedure contains no string literals of length greater than one character.

8.7 Images

Images contain two pointers. The first is to the bitmap vector and the second to a file descriptor if the image is the screen. The two pointer fields occur immediately after the header in the descriptor.

9 Dynamic Storage Allocation With a Compacting Garbage Collector

The design aim of the storage allocator is to produce a utility which will allocate store as quickly as possible. To this end a storage allocator was written which allocates store from a single block of space. Assuming finite space this in turn requires a garbage collector which compacts wanted items to the bottom of the space.

In order to be able to garbage collect it is first necessary to mark all the heap objects which are reachable either from the user's data structures or from the run time support routines. This marking algorithm is described in detail in chapter 8.

As a consequence of the compacting garbage collector algorithm every pointer to an object that may move must be in the heap. This is achieved by having a vector of pointers at the re-use base to hold the pointers normally held in registers. The pointers that are saved in this vector are SMSB, LMSB, the pointer to the class identifier vector and the pointers to the tables and stack used by the C-POMS. The register CP is saved by entering it as an offset in the return address field of the current frame. The registers LMSP and LPSP are entered as offsets in the MSP and PSP fields of the current frame. All the other pointers held by the runtime support such as LPSB and SPSB can be calculated from the pointers held in the vector of pointers. Hence when garbage collection is complete all the registers can be restored from the vector of pointers. Once the vector of pointers has the registers copied into it the marking algorithm can mark from it and find every reachable object.

The compacting garbage collector algorithm to be described is an adaption of an algorithm given by Morris [2]. In general terms the algorithm requires two sequential passes across the heap with compaction resulting from sliding wanted objects over garbage. Before describing the algorithm in detail it is first necessary to consider the runtime heap layout.

```
-------------------------------------------------------------------------
|           |          |          |          |          |         |
| Run time  |  User    |  Save    | Standard | Other    |         |
| Support   |          |          |          | Heap     |-->      |
| Code      |  Code    |  Vector  | Frame    | Objects  |         |
|           |          |          |          |          |         |
-------------------------------------------------------------------------
^                      ^                                ^         ^
|                      |                                |         |
heap                 re-use                            heap      heap
base                  base                             top       max
```

Heap layout at run time

Because of the nature of the code produced by the PS-algol compiler it is guaranteed that the runtime support code and the initial user code can never contain pointers to objects created at run time. It is therefore possible to implement a re-use base, as shown in the above figure, beneath which compaction cannot take place and there are no pointers requiring updating due to the movement of objects during compaction.

For the following algorithm to be implemented it is necessary to be able to distinguish pointers from object headers. This has been achieved by making the assumption that pointers are not going to be larger than 28 bits and ensuring no heap object has a tag value of zero. In this way inspection of the high order 4 bits distinguishes pointers from object headers.

The following algorithm immediately follows the marking of accessible objects described in Chapter 8.

scan 1

The first scan of the garbage collector runs from the re-use base to the heap top. If a heap object is found to be unmarked the next 4 bytes are over-written with the object's size and the next object is considered. The over-writing with the object's size is done purely for efficiency reasons in the second scan. As the minimum size for a heap object is eight bytes this presents no problems. If the heap object is marked for retaining, all its pointer fields are examined. Each pointer which points backward into the heap but does not point before the re-use base is reversed ie. the contents of the 4 byte quantity being pointed to is swapped with the pointer value. Note that the contents pointed at may be either an object header or another pointer. Pointers pointing forward are left alone in this scan. The next object is then considered and the process continues until the heap top is reached.

scan 2

The second scan runs from the re-use base to the heap top. This scan requires a second pointer (shift address) into the heap indicating the point to which an object should be shifted in order to compact. Until an unwanted object is encountered this pointer will always coincide with the scanning pointer. If an unwanted object is found the scanning pointer jumps over it and the shift address remains unchanged. If the object has a valid object header then if necessary it is slid into position and the shift address updated. Should the first 4 bytes of the object be a pointer then this pointer will represent a previously reversed pointer from another heap object. In fact what has happened is that a set of nodes pointing to a common node has been transformed into a chain of pointers to the common node. What is required is to step down this chain over-writing the pointers at each step with the shift address. When the end of the chain is reached ie. a valid object header is found, this is placed into the original and the original slid down to shift address. Once an object has been correctly sited any pointers within it are examined. Any forward pointers are reversed (as for backward pointers in scan 1) so that they may be correctly updated when the objects they point to are moved later during this scan. Backward pointers are left unchanged as they are now pointing to objects which will not move throughout the remainder of this garbage collection. Lastly the object is marked unwanted ready for the next garbage collection. This process continues until the heap top is reached.

Once scan 2 has been completed the garbage collection is over. If not enough free space has been released to satisfy the original request then either the heap may be expanded or selected heap items may be written back to the data base in order to release more space. Should the second alternative be chosen it is not necessary to go through a mark phase again as a garbage collection has just completed and by definition the only objects left in the heap are those reachable. Because the above algorithm has the effect of turning multiple pointers to a common node into a chain of pointers to the common node, it may be adapted to enable all the common references to a node to be updated with the node's address in the data base. This therefore allows selected objects to be transferred back to the database whilst maintaining heap integrity. A careful strategy for expulsion of heap objects must be worked out before attempting this.

References

1. Bailey, P.J., Maritz, P. & Morrison, R. The S-algol abstract machine. University of St Andrews CS/80/2. (1980).
2. Morris, F.L. A time and space efficient garbage compaction algorithm. Comm.ACM 21,8 (1978), 662-665.
3. PS-algol reference manual. PPR12-85. Universities of Glasgow and St Andrews.
4. The CPOMS Persistent Object Management System. PPR13-85. Universities of Glasgow and St Andrews.

Appendix I

The Compiler Output Format

At the end of each program the compiler will pad the output code to a 128 byte boundary. That is the compiler outputs the code vectors, followed by the vector of closures, the string literal vector and the string literals. The last piece of information is the class.id vector followed by some padding characters and then the following six pieces of information.

code size in bytes (4 bytes)
start address in the code (4 bytes)
standard frame main stack size (4 bytes)
standard frame pointer stack size (4 bytes)
version number (1 byte)
POMS version number (1 byte)

Each unit of compilation is made up of a number of code vectors. At the end of the unit of compilation there is a string vector whose elements point to the string literals that are class identifiers. The string addresses are relative to the start of the unit of compilation. These addresses must be made absolute on loading.

Appendix II

PS-code Generated by the PS-algol Compiler

A summary of the PS-code generated by the PS-algol compiler for each syntactic construct is given here. In the description E, in the source code represents an expression and E, in the code represents the PS-code for that expression. Sometimes the expressions are of type void. A description of the instructions themselves is given in Appendix III.

Source	PS-code
~E	E not.op
+E	E
-E	E neg.op
unary.function(E)	E unary.function.op
write E1:E1′,.....En:En′	s.o E1 E1′ write.op....... ...En En′ write.op erase.op

Write operates for reals, ints, bools and strings.

output E0,E1:E1′....En:En′	E0 E1 E1′ write.op.......... ....En En′ write.op erase.op
out.byte E0,E1,E2	E0 E1 E2 write.op
read	s.i read.op
read(E)	E read.op

similarly for peek, read.name, reads, readi, readb, eoi and read.byte.

E1(E2) := E3	E1 E2 E3 subvass or subsass
E1 **or** E2	E1 jumptt(l) E2 l:
E1 **and** E2	E1 jumpff(l) E2 l:
E1 <binary.op> E2	E1 E2 binary.op
(E)	E
E1(E2|E3)	E1 E2 E3 substr.op or subimage.op
E1(E2)	E1 E2 subs or subv or subpixel.op
E(E1,.....En)	E E1....En apply.op
@E **of** T[E1,....En]	E E1.......En make.vector
E(E1,.....En)	E E1.......En form.structure
E ?	E finish.op
abort	abort.op
vector E1::E1′,...En::En′ **of** E	E1 E1′...En En′ E iliffe.op
if E1 **do** E2	E1 jumpf(l) E2 l:
if E1 **then** E2 **else** E3	E1 jumpf(l) E2 jump(m) l: E3 m:
repeat E1 **while** E2	l: E1 E2 bjumpt(l)
repeat E1 **while** E2 **do** E3	l: E1 E2 jumpf(m) E3 jump(l) m:
while E1 **do** E2	l: E1 jumpf(m) E2 jump(l) m:
for I=E1 **to** E2 **by** E3 **do** E4	E1 E2 E3 l: fortest.op(m) E4 forstep.op(l) m:
let I = E	E
let I := E	E
proc ; E	E return
structure I	load.class.id
image E1 **by** E2 **of** E3	E1 E2 E3 form.image
limit E1 **at** E2,E3	E1 E2 E3 limit.2
limit E1 **to** E2 **by** E3 **at** E4,E5	E1 E2 E3 E4 E5 limit.4
<raster.op> E1 **onto** E2	E1 E2 <raster.op>
<literal>	ll.literal dependent on type
<identifier>	load.stack
<identifier> := E	E load.stack.assign

A load.stack instruction may be one of load, pload, dload or dpload for each of the four environments. There is an equivalent instruction for each assignment.

The unary functions are upb, lwb, float.

The binary operations are eq.op, neq.op, lt.op, le.op, gt.op, ge.op, plus.op, times.op, minus.op, div.op, rem.op, divide.op, is.op, isnt.op and concat.op.

```
case E0 of                                       E0
E11,E12,......E1n : E10        E11 cjump(l1) E12 cjump(l1).....E1n cjump(l1)
                                jump(M1) l1 : E10 jump(xit)
E21,E22,......E2n : E20      M1:E21 ..........
  .
  .
  .
  .
  .
default : Ek+1 0              Mk:Ek+1 0
                             xit:
```

Appendix III

PS-algol abstract machine operation codes

Note

The operation codes are held as 8 bit quantities. If the instruction requires an operand then the operand will follow in the next 1, 2, 3 or 4 bytes of the code. Therefore most of these instructions have a long and a short form. If there are two forms the operation codes usually differ by 128. Typed instructions have an encoded name with the following convention.

ib	integer or boolean
r	real
s	string
p	pointer
pr	procedure
v	void

Jump Instructions

The code address in the instruction is relative to the code pointer. That is the address of the instruction following the jump in the code stream. Jump addresses are in bytes. The number in brackets is the number of bytes in the operand(s).

0		128	fjump(2)
1	bjump(1)	129	bjump(2)
2		130	jumpf(2)
3		131	jumpff(2)
4		132	jumptt(2)
5		133	for.test(2,2,2)
6	for.step(1)	134	for.step(2)
7	cjump.ib(2)	135	cjump.r(2)
8	cjump.s(2)	136	cjump.p(2)
9	cjump.pr(2)	137	
10	bjumpt(1)	138	bjumpt(2)
11		139	

Stack accessing instructions

The address is the stack offset from LMSB or LPSB in the appropriate frame. The instructions starting with the letter p refer to the pointer stack.

12	local(1)	140	local(2)
13	plocal(1)	141	plocal(2)
14	dlocal(1)	142	dlocal(2)
15	dplocal(1)	143	dplocal(2)
16	global(1)	144	global(2)
17	pglobal(1)	145	pglobal(2)
18	dglobal(1)	146	dglobal(2)
19	dpglobal(1)	147	dpglobal(2)
20	stand(1)	148	stand(2)
21	pstand(1)	149	pstand(2)
22	dstand(1)	150	dstand(2)
23	dpstand(1)	151	dpstand(2)
24	load(1,1)	152	load(1,2)
25	pload(1,1)	153	pload(1,2)
26	dload(1,1)	154	dload(1,2)
27	dpload(1,1)	155	dpload(1,2)
28	local.ass(1)	156	local.ass(2)
29	plocal.ass(1)	157	plocal.ass(2)
30	dlocal.ass(1)	158	dlocal.ass(2)
31	dplocal.ass(1)	159	dplocal.ass(2)
32	global.ass(1)	160	global.ass(2)
33	pglobal.ass(1)	161	pglobal.ass(2)
34	dglobal.ass(1)	162	dglobal.ass(2)
35	dpglobal.ass(1)	163	dpglobal.ass(2)
36	stand.ass(1)	164	stand.ass(2)
37	pstand.ass(1)	165	pstand.ass(2)
38	dstand.ass(1)	166	dstand.ass(2)
39	dpstand.ass(1)	167	dpstand.ass(2)
40	load.ass(1,1)	168	load.ass(1,2)
41	pload.ass(1,1)	169	pload.ass(1,2)
42	dload.ass(1,1)	170	dload.ass(1,2)
43	dpload.ass(1,1)	171	dpload.ass(1,2)

The load instructions have as a first operand the th index into the display and as the second operand the stack address.

Procedure and block entry and exit

44	apply.op(1,1)		
45	store.closure(1)	173	store.closure(2)
		174	block.enter(2,2)
47	return.ib	175	return.r
48	return.s	176	return.p
49	return.pr	177	return.v
50	block.exit.ib	178	block.exit.r
51	block.exit.s	179	block.exit.p
52	block.exit.pr	180	block.exit.v

Image Operations

57	subimage.op	185	subpixel.op
58	form.image	186	form.pixel.op(1)
59	limit.2	187	limit.4
60	raster.copy	188	raster.not
61	raster.rand	189	raster.nand
62	raster.ror	190	raster.nor
63	raster.xor	191	raster.xnor

Structure and Vector instructions

64	form.structure(1,1)	192	form.structure(2,2)
65	is.op	193	isnt.op
66	subs.ib	194	subs.r
67	subs.s	195	subs.p
68	subs.pr		
69	subsass.ib	197	subsass.r
70	subsass.s	198	subsass.p
71	subsass.pr		
72	makev.ib(2)	200	makev.r(2)
73	makev.s(2)	201	makev.p(2)
74	makev.pr(2)		
75	iliffe.ib(2)	203	iliffe.r(2)
76	iliffe.s(2)	204	iliffe.p(2)

77 iliffe.pr(2)	
78 subv.ib	206 subv.r
79 subv.s	207 subv.p
80 subv.pr	
81 subvass.ib	209 subvass.r
82 subvass.s	210 subvass.p
83 subvass.pr	
84 upb.op	212 lwb.op
85 concat.op	213 substr.op
86 load.class.id(1)	214 load.class.id(2)

Load literals

90 ll.int(1)	218 ll.int(4)	32 bits
91 ll.bool(1)	219 ll.real(8)	64 bits
92 ll.string(1)	220 ll.string(2)	
93 ll.char(1)	221 ll.file	
94 ll.nil.string	222 ll.nil.pntr	
95 ll.nil.pr		
96 eq.ib	224 eq.r	
97 eq.s	225 eq.p	
98 eq.pr		
99 neq.ib	227 neq.r	
100 neq.s	228 neq.p	
101 neq.pr		
102 lt.i	230 lt.r	
103 lt.s		
104 le.i	232 le.r	
105 le.s		
106 gt.i	234 gt.r	
107 gt.s		
108 ge.i	236 ge.r	
109 ge.s		
110 plus	238 times	
111 minus	239 div	
112 rem	240 neg	
113 fplus	241 ftimes	
114 fminus	242 fdivide	

115 not.op	243 fneg
116 float1	244 float2
120 erase.ib	248 erase.r
121 erase.s	249 erase.p
122 erase.pr	
123 rev.ms	251 rev.ps
124 newline(1)	252 newline(2)
125 finish.op	253 abort.op
127 read.op*(1)	255 write.op**(1)

* read.op uses the second 8 bits for the following functions

0	readi	1	reads
2	readb	3	read.byte
4	peek	5	read
6	eoi	7	read.name
8	read.a.line	9	readr

** write.op uses the second 8 bits for the following functions

0	write.i	1	write.s
2	write.b	3	out.byte
4	write.r		

Appendix IV

The Standard Identfier File and Standard Frame

The standard identifier file is used by the compiler to declare the procedures and identifiers that are global to every program. Therefore it is important that every program is compiled using the same standard identifier file and that the standard frame is initialised accordingly. To ensure this is the case the implementor should use the standard identifier file to provide the compiler with a suitable address allocation strategy.

Addresses are allocated on the appropriate stacks in increasing order as identifiers are encountered in the standard identifier file. The main stack starts at stack offset 2 and the pointer stack at stack offset 3. However it is not desirable to include all of the standard functions in the standard identifier file because some of them may violate the type rules or the database locking conventions. To overcome this problem three integers are inserted at the start of the file to indicate how many identifiers are missing. The first integer is the number of main stack addresses allocated, the second is the number of pointer stack addresses allocated and the third is a version number for the PS-algol POMS code. Using this mechanism it is possible to have two versions of the standard identifier file, i.e. one for ordinary user programs and one for system programs.

The structure of the standard identifier file

Everything in the standard identifier file is legal PS-algol code except for the three integers mentioned above. The initial values of the identifiers are ignored at runtime because no code is generated for these declarations. As a consequence of the hiding mechanism any standard identifier that is to be hidden must come at the beginning of the file. The standard identifier file is therefore divided into a public and private section.

Private standard identifiers

Only those identifiers declared in the private section will be described here as the public identifiers are described in the PS-algol language reference manual [3].

```
let structure.table = proc( -> *cstring )
! this returns a vector of strings containing all of the
! class identifiers used in the program

!---------------------- File I/O Identifiers
let read.blocks = proc( file f ; int blks -> *int )
! this reads blks four byte words from the file into an integer vector

let write.blocks = proc( file f ; *int blks )
! this writes a vector of integers to a file

let open = proc( cstring f ; cint m -> file )
! this opens the file with name f in mode m
! modes are: 0 - read only, 1 - write only, 2 - read and write

let close = proc( cfile f )
! this closes the file for file descriptor f.

let seek = proc( cfile f ; cint offset,from )
! this moves offset bytes through file f from 0 - start of file,
! 1 - current position and 2 - end of file

let create = proc( cstring f ; cint m )
! this creates a file with name fname in mode m, m is the decimal value
! of the file protection bitmap

let flush = proc( cfile f )
! this causes the output buffer for file f to be written out

let read.real = proc( cfile f -> real )
! this reads characters from file f to form a real number
! this is written in PS-algol and loaded by the PS-algol POMS code

let writer = proc( cfile f ; creal n ; cint a,b )
! this writes a real number n to file f
! in a field width of a with b spaces after the number
! this is written in PS-algol and loaded by the PS-algol POMS code

!------------------------ C-POMS database functions
let createdb = proc( cstring name,pass -> pntr )
! this creates a database with the given name and password, the root
! object is a opdb.result structure (see below). nil is returned
! if successful, otherwise an error.record

let opendb = proc( cstring name,pass ; cint m -> pntr )
! this opens the database with the given name if the password pass is
! correct and a lock for mode m (0 read, 2 read and write) can be
! acquired. When locked all reachable databases must also be locked.
! A pointer to the root object is returned if successful, otherwise
! an error.record

let garbage.collect = proc( cstring name -> pntr )
! this causes a garbage collection to occur that will try to restrict
! its effect to the named database, this is to enable a form of
! incremental disk garbage collection. nil is returned if
! successful, otherwise an error.record

let Garbage.collect = proc( -> pntr )
! a full disk garbage collection is performed, any unreachable
! databases are removed. nil is returned if successful,
! otherwise an error.record

let error.records = @1 of c*cpntr[ @1 of cpntr[ nil ] ]
! error.record structures returned by POMS procedures

let shift.l = proc( int value,count -> int )
! shift the first parameter left 'count' places
! bringing in zeros at the low order end

let shift.r = proc( int value,count -> int )
! shift the first parameter right 'count' places
! bringing in zeros at the high order end

let b.and = proc( int value1,value2 -> int )
! logical (bitwise) 'and' of value1 and value2

let b.or = proc( int value1,value2 -> int )
! logical (bitwise) 'or' of value1 and value2

let b.not = proc( int i -> int )
! boolean not operator

let fiddle.r = proc( real n -> *int )
! split a real into a vector of two integers.
```

```
let class.identifier = proc( pntr p -> string )
! given a pointer return the class identifier of the structure
! that it points at

!------------------------- Process control primatives
let fork = proc( -> int )
! Unix process fork - returns process number of child to parent
! returns 0 to child

let exec = proc( *string s )
! Unix exec - calls the shell using the command held in s

let wait = proc( int i -> *cint )
! Unix system call - waits for a signal from a child process
! or for the process i to die
! returns the process id and the status

let system = proc( string s -> int )
! Unix system library call
! does a fork and execs a shell to execute the command s


!------------------------- Private Graphics functions
let line.end = proc( #pixel i ; pixel p ;
                     cint x,y,direct -> int )
! searches for the first pixel of colour p from position x,y in i
! direct specifies the search direction
! odd numbers do not look at boundary pixels
! 0,1 - left ; 2,3 - right ; 4,5 - down ; 6,7 - up
! return the position of the pixel or 1 past the position last searched

let set.locator = proc( cint mode,flags,tabs,ticks )
! set the conditions which govern when records are generated by the mouse

let set.cursor = proc( cint mode )
! set the mode in which the cursor works

let plane.of = proc( c#pixel i ; cint p -> *int )
! this procedure returns the ith plane of image i

let pnx.line = proc( c#pixel p ; cint x1,y1,x2,y2,style )
! this procedure draws a line from x1,y1 to x2,y2 on image p
```

! style may be 1 - pixels off ; 2 - pixels on ; 3 - pixels opposite

PS-algol Prelude

In addition to the standard identifier file a prelude is used to predeclare some useful structure classes. Several of these are used by the PS-algol compiler to generate code to construct the Outline picture representation. The first two are used by the C-POMS and so must remain in a fixed position in the prelude. At runtime the C-POMS can index the class identifier vector to allocate reserved pids to these structure classes and use the trademarks it finds to construct error records and database root objects.

```
structure error.record( cstring error.context,error.fault,error.explain )
! trademark for an error record

structure opdb.result( pntr root.of.db )
! trademark for the real root object of a database

structure culr.strc( cpntr nxtre ; cstring shade )
! trademark for the Outline colour of expression

structure oprtn.strc( cpntr lft,rght ; cint opoo )
! trademark for the ^ and & Outline operators

structure scrbl.strc( cstring msge ; creal xxl,yyl,xxr,yyr  )
! trademark for Outline text from to expression

structure poin.strc( creal pnx,pny )
! trademark for Outline point [pnx,pny]

structure trnsfrm.strc( cint trnsfrm ; cpntr mrtre ;

                        creal trnsfrm.x,trnsfrm.y )
! trademark for Outline shift, rotate and scale expressions

structure mouse( cint X.pos,Y.pos ;
                 cbool selected ; c*cbool the.buttons )
! this is the structure returned by locator

structure point.strc( cint point.x,point.y )
! this is the structure returnd by cursor.tip
```

APPENDIX 5

History & People

PS-algol is the third member of a family of languages. The first, although we did not recognise it at the time, was a language invented by David Turner called algol-s and was used by him and Ron Morrison as the basis of a Senior Honours project at St Andrews University [9]. David Turner subsequently took up an appointment at the University of Kent at Canterbury and the second language, S-algol [8,12] was developed by Ron Morrison and Pete Bailey using the main ideas of algol-s. The third language is PS-algol and it incorporates the main concepts of both algol-s and S-algol as well as the concept of persistence developed at the University of Edinburgh.

The Data Curator project began in November 1979 funded by SERC grant GR/A/86541 to Malcolm Atkinson at the University of Edinburgh. Malcolm was joined by Ken Chisholm and Paul Cockshott and work began on the theoretical basis of persistence, its integration into a programming language and a support system for persistent data. After some dissatisfaction with attempts to integrate persistence with Algol 68 and Pascal contact was made with Ron Morrison and Pete Bailey in St Andrews.

In 1983, Ken Chisholm left the project and Malcolm Atkinson spent a year (August 1983-July 1984) at the University of Pennsylvania, USA. On his return he took up the second Chair of Computing Science at the University of Glasgow. Ron Morrison spent July 1983 to December 1983 at the Australian National University, Canberra and on his return was joined by Alfred Brown and Alan Dearle in St Andrews.

In 1983 ICL began collaboration with us on Persistent Programming at both Universities funding some of the work and setting up an absorber project to build compilers for PS-algol on the ICL machines - initially the ICL 2900 series. Late in 1984 the combined teams were notified that they had been awarded funding for the PISA project, which will among other things develop further PS-algol.

Present Team Members

University of Glasgow	University of St Andrews	ICL
Malcolm Atkinson	Ron Morrison	Graham Pratten
Paul Cockshott	Pete Bailey	John Robinson

Paul Philbrow	Alfred Brown	Malcolm Jones
Jack Campin	Al Dearle	

Associates

Peter Buneman	Tony Davie	Dave Tuffs
Larry Krablin		Nick Capon
		Mike Guy

Support Grants

SERC GR/C/21977	SERC GR/C/15907
SERC GR/C/21960	ICL URC
SERC GR/C/86280	
SERC GR/C/68415	
ICL URC	

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", Foundations of Software Technology and Theoretical Computer Science, (ed M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

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.
"Procedures 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", submitted to Software Practice and Experience, November 1984.

Morrison, R., Dearle, A., Bailey, P., Brown, A. & Atkinson, M.P.
"An integrated graphics programming system", submitted to Eurographics '85, Nice, (Sept. 1985).

Morrison, R., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", submitted to 8th International Conference on SE Imperial, (August).

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, University of Edinburgh, February 1983.

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

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

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

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

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 4 February 1985.

Copies of documents in this list may be ootained by writing to The Secretary, The Persistent Programming Research Group, Department of Comouting 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: Descriotion and Examoles	[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]
PPR-11-85	PS-Algol Abstract Machine Manual	[Printed]
PPR-12-85	PS-Algol Reference Manual - second edition	[Printed]
PPR-13-85	CPOMS - A Revised Version of Persistent Object Management in C	[In Preparation]