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

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

# Polymorphic Names and Iterations

M. P. Atkinson and R. Morrison

Persistent Programming
Research Report 53
November 1987

Graham Kirby

## Preface

This paper was presented at the International Workshop on Database Programming Languages held at Roscoff in France during September 1987 and will also appear in the proceedings of that workshop.

# POLYMORPHIC NAMES AND ITERATIONS

Malcolm Atkinson[1] & Ronald Morrison[2]


## ABSTRACT

This paper presents polymorphic names as manipulable values in a strongly typed language. Their polymorphism is used to permit the programs to be statically type checked, except where the programmer explicitly requires otherwise. These names then allow code to be written which abstracts over names or iterates over names. The utility of such name manipulation is illustrated by demonstrating that the equivalent of file and directory operations may now be implemented. Its limitations are illustrated by considering the implementation of join.


[1]Computing Science, University of Glasgow, G12 8QQ, Scotland

[2]Computational Science, University of St. Andrews, North Haugh, St. Andrews, KY16 9SS, Scotland

## Introduction

Names are used for many categories of objects within programming languages - for example, to name constants, variables, points in the program, exceptions etc. When they name fields of records, then it is often the case that some input and output operations could use those names. For example, in a form filling system, or in a browser [Dearle & Brown, 87]. Diagnostic tools and program construction aids need to manipulate, input and output these names.

In operating system command languages, editors and other user interfaces, they are used to identify objects from different sets of categories, especially file directories and files. At present these names may obey different rules from those in the programming language. As we attempt to develop a single coherent system in which long and short term data (code, objects, etc.) are treated consistently [Atkinson *et al.*81, Atkinson *et al.* 83, Atkinson & Morrison 85b] it has been necessary to consider carefully the treatment of names.

Unfortunately, during the development described in those cited papers there were two flaws in our treatment of names:

i) the interpretation of field names in the type checking rules implied a single universe of names for fields - which is known to be unmanageable in large evolving systems; and
ii) program identifiers were used to name some things (e.g. procedures and structure classes) while strings were used to name other things (notably databases and entries in databases).

The former problem appears in many systems as we note in various surveys [Atkinson & Buneman 88, Buneman & Atkinson 86, Atkinson *et al.* 87]. The latter problem manifests itself in most languages as the use of strings for file names. It has the inconvenience of introducing a quite different, dynamic binding rule for the interpretation of these names. Normally, the operating system is responsible for providing this rule. The inconsistency introduced makes programming more difficult and requires program alteration when programs are moved between operating systems

In PS-algol and its descendents we have wished to encompass more of the semantics that affect the execution of programs to give the programmer a consistent world for the total computation. We have therefore sought to remove these anomolous string-names and their inconsistent interpretation. A similar motivation has influenced other work [Buhr & Zarnke 87, Richardson *et al.* 87]. We envisage that by continuing this development most of the functions of an operating system can be given a consistent semantics which is also consistent with the command languages and the programming languages provided. The task of learning to use the composition of these, and of implementing them is then much simplified.



For example, in many programming environments there are naming systems for: files, databases, schema components within the database, command language variables, commands, parameters, programs, processes, procedure libraries, modules within these libraries, etc. Often different rules apply to the name management for each of them which have to be both implemented and understood.

With this motivation we proposed name spaces [Atkinson & Morrison 85a] and have subsequently refined them and renamed them environments in our implementation of Napier [Atkinson *et al.* 86, Atkinson & Morrison 87, Atkinson *et al.* 87]. These ameloriate the two problems identified above but do not permit all aspects of an operating system to be modelled. At the first Appin workshop [Atkinson & Morrison 85a] we noted our inability to iterate over structures containing names. Interaction, that is data transfer across the inevitable boundary between the computation described by the language and the environment of that computation, was also incomplete. For example, names could not be communicated to and from the user without treating them as strings. The lack of iteration meant operating system functions like browsing a directory of files could not be implemented. The lack of interaction meant that generic I/O (e.g. forms packages) could not be implemented easily, though Dearle and Cooper has developed a use of the callable compiler which overcomes this deficiency [Dearle and Brown 87,] Cooper *et al.* 87, Cooper *et al.* 86, Cooper & Atkinson 87].

This paper shows how the new language construct: *polymorphic name types*, allows us to define iterators and transput operations and hence code these hitherto problematic functions. First the polymorphic name type, the universal extensible union type, the polymorphic I/O construct and the iterator construct of Napier are defined. Then example program fragments illustrate how they are used.

## The Polymorphic Type Name

Like procedures and abstract types in Napier [Atkinson & Morrison 87], names may be parameterised by any type thus specifying the type of objects they may name. Syntactically there is a name type constructor **name** which when parameterised with a type yields a type. For example:

**name** [string]

which is a set of all names which may name a string. More precisely, we consider all environments (those produced explicitly and manipulable with the **env** construct, those corresponding to records and those associated with the lexical block structure) to be sets of quadruples. Each quadruple is a name, type, constancy, value. The type with which a name value is parameterised must match under the type rules the second element of this tuple when the name value is matched with the first element.

The operations on names are:

- type test,
- input and output;
- type consistent assignment; and
- lexical ordering

These operations are further defined below. There are also two transfer functions on names:

**let** *nameToString* = **proc**[*t* : **type**] (*n* : **name** [*t*] → *string*)

and

**let** *stringToName* = **proc**[*t* : **type**] (*s* : *string* → **name** [*t*])

The type test has the form

<exp> **is** <ptype>

and type rule

t: *t* **is** ptype ⇒ **bool**

where ptype is:

a) any one of the predefined types (e.g. *int, real, bool*);

b) any user defined type name, (i.e. an in scope occurrance of <type_name> from **type**<type_name> **is** ...);

c) any type expression (i.e. such as may appear after **is in type** ... **is** ...);

d) any type *constructor* (e.g. **abstype**, which might have been used in **type** *stack* **is** **abstype** ...).

Figure 1 illustrates the use of the type test



```
let typeName = proc [t : type] (x : t -> string)
 begin
 case true of
   xisint           : "int"          ! base types
   x is real        : "real"
   x is bool        : "bool"
   x is string      : "string"
   x is picture     : "picture"
   x is pixel       : "pixel"
   x is image       : "image"        !constructors
   x is vector      : "vector"
   x is structure   : "structure"
   x is union       : "union"
   x is proc        : "proc"
   x is env         : "env"
   x is abstype     : "abstype"
   x is name        : "name"
   x is any         : "any"
   default "impossible"
   end
```

**Figure 1: A procedure to give a string corresponding to the type of its parameter**

The equality test on name is **true** if they both are represented by the same sequence of characters and if they are both restricted to exactly the same type. Thus the program:

**let** *p1* = **name** [*real*] *floccinaucinihilipilification*
**let** *p2* = *stringToName* [*real*] ("floccinaucinihilipilification")
**print** *p1* = *p2*

would print **true**.

Input and output are discussed in a subsequent section, and assignment is identical with all other assignments in the language.

Lexical ordering is defined for the corresponding strings and is irrespective of type.

$$\langle exp_1 \rangle < \langle exp_2 \rangle$$

for

∀ t, t' **name** [t] <**name** [t']

is exactly equivalent to

$$\mathit{nameToString}\,(<exp_1>) < \mathit{nameToString}(<exp_2>)$$

We use this ordering when defining iterators.

### The Universal Extensible Union Type

In PS-algol we had an extensible union type, **pntr**, and we grew to appreciate its utility; indeed much of the database programming, including the interface to persistent data and data model implementation depended on it [Atkinson *et al.* 87, Cooper *et al.* 87].

We refer to it as a *union* type because it may refer to an instance of *any* structure class. We refer to it as *extensible* as new classes declared after the use of **pntr** are eligible as referends, thus the set of possible referends is increased when each structure class is declared. It was not *universal* as there were types, e.g. **int**, which were excluded from its set.

It was valuable because it allowed a type check to be delayed, because it allowed us to limit the traversal of the type match algorithm, and because it allowed generic code to be written applicable to future types, possibly with the execution taking into account the actual type. It was, however, overused, as no more specific alternative was available when referend types were predetermined. It was also unfortunate as its pronunciation 'pointer' evoked connotations of other languages where such things provide a loop-hole in the type system and even pointer arithmetic. Of course, these horrors do not exist in PS-algol.

In Napier we therefore allow proper constraint of referend type where appropriate in data structures, and we use polymorphism to implement most generic code. But we have retained the valuable properties of **pntr** in a type **any**, but removed an irksome restriction by making it universal.

There are few operations on values of type **any** (only equality, inequality and assignment) thus it is safe. To gain access to other operations on the values it is necessary to project out of the union, just as one projects out of a statically defined union. A delayed type check is needed in both cases. We now make this projection explicit. (The implicit projection from **pntr** was one of the causes of a single name space of field names.) Thus our **any** is similar to Cardelli's **dynamic** [Cardelli & MacQueen, 85, Cardelli 85].

Note **name [any]** is the type which includes all possible names.

### Polymorphic Input and Output

The output statment **print** in PS-algol [PPRR-12] is already polymorphic, and handles multiple fonts, multiple destinations and its default actions may be replaced by a programmer. In Napier we retain the essence of this **print** clause but we are revising details [Philbrow *et al.* 87].

Thus

**print** 4 + 3
**print** "freedom is never achieved by violence"
**print** 14·2 + 3·7

would print the values in the accepted format. Logically the print statement would be written as:

**print** [*int*] 4 + 3

etc. to be consistent with our other polymorphic constructs, however in this case we have decided the explicit type parameter is so tedious that we prefer to omit it and tolerate the inconsistency. (There is some hope that we may be able to return to consistency by omitting the type parameters elsewhere c.f. Poly [Matthews 85].)

In PS-algol input was performed by special functions indicating the type expected, e.g. *readstring*. This cannot be data type complete since the type space is infinite, and so is inconsistent with our design principles. It also masks the projection and dynamic type check from the sequence of user actions (e.g. key strikes, mouse clicks etc.) to the internal type. A dynamic type check, prevalent in languages, which we believe should be properly visible and parametric. **read** therefore takes a type parameter, as is shown below:

| | | |
|---|---|---|
| **let** *i* := | **read** [*int*] | ! create an integer variable *i* and |
| | | ! initialise it to the next input integer |
| **let** *r* = | **read** [*real*] | ! declare real constant *r* |
| **let** *vs* = | **read** [**string*] | ! *vs* becomes a constant refering to a |
| | | ! vector of strings which are read in. |

A consequence of this treatment is the **read** operation corresponds to a call of the compiler on the relevant input source seeking the specified type. Parts of a program to plot an arbitrary function is shown as figure 2. But it may also receive an already typed object via *cut & paste* actions, since, if we capture the system within one semantics the structure and type information is invariant over these operations.

. . .
**print** "'n supply initial X"
**let** *Xi* = **read** [*real*]
**print** "'n supply final X"
**let** *Xf* = **read** [*real*]
**print** "'n supply f(X)"
**let** *f* = **read** [**proc** (*real* —> *real*)]
...

Figure 2: A program fragment collects data describing what to plot

### Polymorphic Iterations

When a polymorphic procedure is defined this indicates that different applications of the procedure may have parameters of different type, but that for each application the procedure body will be executed with a consistent and constant substitution of the type variables. The polymorphic iterator is defined correspondingly. Each traversal of the iteration may be with a different type substitution, but within each execution of the controlled statement the type substitution is constant and consistent.

There are iterators to perform defined sequences of operations in the language e.g.

**for** *i* = 1 **to** 10 **do** ...

with the usual semantics and options. Note that *i* is a constant declared here with the scope of this **for** statement.

There is a similar iteration construct, introduced by **for each**, which iterates over compound objects. Each of the compound objects may be considered a map, e.g. a vector of type *t is a stored map from *int* to t. Identifiers may be provided in the iteration statement to range over the sequence of values in the map, and for every type of map the iteration sequence is defined. e.g.

**for each** *k* → *u* **in** *vs* **do** ...

where *vs* is a vector of strings would apply the controlled clause first with *k* set to the lower bound of *vs* and *u* set to the first string, and repeat for increasing index up to the upper bound. Either control variable may be omitted, e.g.

**for each** *k* **in** *vs* **do** ...

and

**for each** → *u* **in** *vs* **do** ...

Similar arrangements are available for iterating over indexes, with multiple keys having corresponding multiple control variables.



The other major classes of compound object (**struct** & **env**) all encapsulate environemnts (maps from names to values with different types for different names). Consequently the first control variable is a polymorphic name, and the second of the corresponding type, which constitutes polymorphic iteration, e.g.

**for each** [*t* : **type**] *aName* : **name** [*t*] → *aValue* : *t* **in** ...

where *aValue* is of type *t*. Note *t* is available as a type variable in the controlled clause. The iteration substitutes from the quadruple with the least name first.

## Illustrating the use of constructs to manipulate environments

Our environments have been described elsewhere [Atkinson *et al.* 87, Atkinson & Morrison 87]. They may be used to provide extensible objects, and one such application of those would be as file directories - where files are now properly typed.

Figure 3 shows the insertion of a new quadruple in an environment, equivalent to adding a file (with or without write protection) to a directory, *an Env*.

```
...
print  "n' What is the name?"
let    newName = read [name [*int]]
print  "n' What is the intial value for ", newName , " ?"
let    initialValue = read [*int]
print  "n' is the field updateable?"
let    constantField = replyAffirmative ( )
if     constantField then
       insert newName = initialValue in anEnv
else
       insert newName := initialValue in anEnv
```

Figure 3: Inserting a new quadruple in an environment

To illustrate the iterator construct more fully suppose that environments have been chosen to represent some entity, and that now a new property is to be recorded for every instance. The programmer/data designer has decided that such transitions are likely, and considered it worth incurring the additional costs of using **envs** rather than static records. The iteration in Figure 4 would then achieve this.

```
...
print  "'n is the field updateable?"
let    constantField = replyAffirmative ( )
print  "'n What is the name of the new integer field?"
let    newName = read [name[int]]
for each →     anEnv in theIndexToEnvs do        ! don't care about the key
               begin                             ! once for each env
                             ! show the user the environment
               envShow (anEnv)
               print "'nWhat is the initial value for ", newName, " ?"
               let initialValue = read [int]
               if constantField then
                             insert newName = initialValue in anEnv
               else
                             insert newName := initialValue in anEnv
               end              ! of iteration through index
```

Figure 4: An example program fragment to add to a new integer field to all the environments in an index

That example has assumed the existence of a procedure, *envShow*, capable of printing any environment. A simple implementation, utilising polymorphic iteration, is shown in Figure 5.

Figure 6 shows a procedure to copy one element of an environment, then Figure 7 shows how that and polymorphic iteration can be used to construct a back up copy of any environment.

Figure 8 shows how two environments may be combined using the same facilities, and figure 9 shows how a user controlled directory (environment) editor might be built.



```
let    envShow = proc (theEnv : env)
       begin
       for each [t : type] aName :name [t]  → aValue in theEnv do
            begin
            print "'n", nameToString (aName) using xor    ! invert name
            print "=" using copy
            if    aValue is int or aValue is real or aValue is bool or
                  aValue is string then
                  print aValue
            else
                  begin        !here print type name rather than value
                  let typeString = typeName [t] (aValue)   ! see fig 1
                  if typeString = "pixel" or typeString = "picture" then
                     print typeString
              else
                  printEnboldened (typeString)
              end
          end
       end
```

Figure 5: Procedure to print any environment

```
let    copyOneEntry = proc [t : type] (e1, e2 : env; n : name [t])
       if constant e1 (n) then
              insert n = e1(n) in e2
       else
              insert n := e1 (n) in e2
```

Figure 6: A procedure to make an exact copy including constancy of one binding from one environemnt to another

```
let    snapshotEnvs = proc (e : env  → env)
       begin                  ! makes a constant snapshot of its argument
       let res = emptyEnv ( )
       for each [t : type] n:name [t]  → v in e do
          insert n = v in res
       res
       end
```

Figure 7: A proceudre to produce a copy of an environment with all the fields constant

```
let   mergeEnvs = proc (env1, env2 : env)
      begin    adds to env1 all the bindings in env2
      let duplicates = emptyEnv ( )
      for each [t : type] n : name [t] in env2 do
            if n in env1 then
                  copyOneEntry [t] (env2, duplicates, n)
            else
                  copyOneEntry [t] (env2, env1, n)
      if size duplicates ≠ o do raise nameClashes (duplicates)
      end
```

Figure 8: A procedure to add the contents of one environment to another

```
let   userControlledCopy = proc (e : env  → env)
      begin
      let res = emptyEnv ( )
      for each [t : type] n : name [t] in e do
            begin
            print "'n include", n, "?"
            if replyAffirmative ( ) do
               copyOneEntry [t] (e, res, n)
            end
      res
end
```

Figure 9: Procedure that allows the user to control the parts of an environment copied

Finally a program to emulate the *ls* shell command (a simple version) as in UNIX™ is shown as figure 10. Note that *nameToString* is used explicitly because otherwise the name would be printed like a name literal expression, e.g.

**name**[*int*]*fred*

since a language must be able to read its own handwriting.

```
let   listEnv = proc (e : env)
      for each [t :type] n :name [t] in e do
      print nameToString (n)
```

Figure 10: Procedure to list the contents of a name space c.f. ls in UNIX™



**Typing relational join**

At the workshop in Appin in 1985 Peter Buneman [Buneman 85] posed the problem of declaring a procedure which implements join. There were three sub problems:

i) to provide a type which will pass the parameters, i.e. the two relations and the names of the columns on which the join is to be performed;

ii) to check the mutual consistency of these parameters eg that the columns named appear in both relations and have the correct type; and

iii) to generate the type of the result relation.

These language features provide a *partial* solution to the posed problem. Figure 11 shows a polymorphic procedure to perform an equijoin on two relations over a list of columns of type *t*. Each relation is presumed to be a vector of environments, and the columns are identified by a vector of names. The procedures used by *equijoin* are shown in figures 12 to 14.

```
let equijoin = proc [type t] (rel1, rel2:*env; cols:*name [t]  → *env)
      begin
        ! check column names are present in each relation
      allIn[t] (rel1(1),cols)              ! each env in a rel has some set of names
      allIn [t] (rel2(1), cols)
                                           ! set up temporary result structure
      let resSize: = 0
      type tupleList is struct (tuple: env; next: tupleList)
      let tl: = tupleList (emptytuple( ), nil)
                                           ! n*m naïve algorithm
      for each   → e1 in rel1 do           ! each tuple in rel1
      for each   → e2 in rel2 do           ! each tuple in rel2
      if match [t] (e1, e2, cols) do
         begin
         resSize: = resSize + 1
         tl: = tupleList (merge (e1, e2), tl)
         end
         ! final result
      let res = vector 1::resSize of tl (tuple)
      for i = 2 to resSize do
         begin tl := tl (next); res (i) := tl (tuple) end
      res
      end
```

Figure 11: declaring a polymorphic equijoin procedure in Napier

```
let allIn = proc [ type t] (rel: *env; names: *names [t])
        for each  →n  in names  do
            if not  (n in rel(1)) do
                raise wrongColumn
```

Figure 12: check all the columns names are in the first environment

```
let match = proc [type t] (t1, t2: env; cols: *name [t]  → bool)
        begin
        let equal:= true
        for each  → n in cols do
            equal: = equal and t1(n) = t2(n)
        equal
        end
```

Figure 13: test two tuples for equality

```
Let  merge = proc (t1, t2: env   → env)
        begin
        let newTuple = emptyEnv ( )
        mergeEnvs (newTuple, t1)                    ! all columns from rel1 - see fig 8
        for each [t: type] n: name [t] in t2 do
            if not (n in t1) do
                    copyOneEntry [t] (t2, newTuple, n)          ! see fig 6
        newTuple
        end
```

Figure 14: generate the new tuple from the two that matched

Subproblem (i) is solved using this type system. We consider below whether the solution is adequate. The check (subproblem (ii)) has been programmed - figure 12 - verifying that all the columns appear in each relation. The dynamic specification of this condition is acceptable since the check is inherently dynamic; the relevant properties of the parameters may not be determined until the code which calls *equijoin* is executed. The result type (iii) is statically specified and consequently the third subproblem is avoided.



When the quality of this solution is considered, the problems rearise. The type of a relation ***env** is unsatisfactory for a number of reasons:

a) its cardinality is inflexible, leading to the final copy phase of the algorithm;

b) it is not space or update efficient, as the use on **env** rather than **struct** requires a flexible map to be stored and maintained;

c) it does not indicate that every tuple in a relation is over the same columns, thus factoring out the check that columns are valid depends on programmers complying with this *unwritten* convention (it also repeats the type and name information redundantly with every tuple).

These new subproblems are not entirely a result of pedagogical simplification, nor is the naïve algorithm. Subproblem (a) could be overcome by a better data structure, eg a list of vectors. Suppose we used ***struct** (...) to overcome subproblem (b), then we lose the polymorphism and name abstraction of *equijoin*. This could be solved if we say that **env** ⊃ **struct** so that ***env** would type match ***struct** for the relation parameters. But this doesn't deal with the result type, as somewhere we need to compute the appropriate **struct** (...) of the result type, which is dependent on the two input relation parameters. The equivalent calculation takes place on each iteration in procedure merge (fig 14) in the presented solution. At present we have no mechanism for calculating this result type, at the start of *equijoin* and using it (statically) for each iteration. If this deficiency were overcome ***struct** (...) would also deal with subproblem (c), but variants of this subproblem then tends to reappear as solutions to subproblem (a) are constructed.

The *n*m* algorithm should be replaced by sort merge, or use of indexes, but the polymorphic requirement militates against this. The type parameter *t* could be **image** or **proc (int, string** → **real)** or any etc. To use indexes we need to either calculate a hash code or perform a 'less than' comparison operation. This leads us to identify another unsolved subproblem:

d) either a generic operation hashcode [**type** *t*] (*x*: *t* → *int*) or a generic comparison operation less than [**type** *t*] (*x*, *y*:*t* → **bool** ) is required to achieve efficiency,

but we do not know how to define and implement them.

Because of these outstanding problems we build a polymorphic index type constructor into Napier i.e. **index** *t1*, *t2*, ... *tn* → *t*. Using this we can overcome subproblem (a) to (d) but we have achieved this by passing them to its implementor. Even then we cannot properly type *equijoin*, since, if the parameters and the results were to include indexes, these types are static and the result type cannot be computed.

For the moment we remain unable to define an adequate type system for generic applications, and we overcome the problem by synthesising a specific procedure for each type parameterisation of join when it is needed, and then using the callable compiler to build the operation before applying it. Persistence and the universal extensible union type allow us to memoise this operator construction [Cooper *et al*.87]. It is not clear whether a type system which does better than this is achievable.

## Conclusions

The sequence of examples show that scanning directories is now possible, and that other data dependent generic algorithms can be written. The constructs introduced to achieve this - polymorphic name types, type constrained name values, environments and polymorphic iterators - are individually simple to understand and use, they combine well, and they do not result in a loss of type control or incomprehensible computations.

Use of these constructs to build replacement operating system structures will eliminate strings as names. We need to start the bootstrap as a program binds to its environment, and do this by introducing one standard variable *PS* (Persistent Space).

These structures need to be updated to reflect changes in the environment, e.g. addition of new network addresses, new discs etc. It does not appear possible to include that within the language. However we extend the scope of the language there will always be external agents affecting the computation, and consequently a closed universe is impossible, i.e. *deus ex machina* will occur. If we wish to use the same naming system for everything, then we need to expand the type system to contain everything we wish to name. Examples might be machines, devices etc. if they may be explicitly manipulated or selected by the user/programmer. But this makes it difficult to adhere to the principle of data type completeness.

The section on the implementation of a join procedure is included to show that type systems are *still* not adequate for all we would wish to do. We pose the question: "Can we do better than synthesis of code followed by calling the compiler?" for these remaining generic tasks. The advantage of that approach is that more than type checking may be 'statically' determined, i.e. factored out of the operator's iterations.

## Acknowledgements

This work was done at the Universities of Glasgow and St. Andrews with support from the British SERC, from the Alvey programme, and from the SETC and URC sections of STC Ltd. The authors also acknowledge the peace and stimulation of walking on Blackford Hill during their many discussions.



## REFERENCES

**Atkinson *et al* 81** Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P. - "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, **17, 7** (July 81), 24-31

**Atkinson *et al* 83** Atkinson, M.P. Bailey, P.J., Chisholm, K.J., Cockshott, W.P and Morrison R. - "An Approach To Persistent Programming", *The Computer Journal*, **26**, 4 (1983), 360-365.

**Atkinson *et al* 86** Atkinson, M.P., Morrison, R. and Pratten, G.D. - "Designing a Persistent Information Space Architecture", in Proceedings of Information Processing '86, Dublin, Eire (Sept. 1986), 115-9.

**Atkinson *et al* 87** Atkinson, M.P., Buneman, O.P. and Morrison, R. - "Delayed Binding and Typechecking in a Database Programming Language", to be published in The Computer Journal, April 1988

**Atkinson & Buneman 88** Atkinson, M.P., and Buneman, O.P. - "Types and Persistence in Database Programming Language Design", to be published in *ACM Computing Surveys.*

**Atkinson & Morrison 85a** Atkinson, M.P. and Morrison, R. - "Types, Bindings and Parameters in a Persistent Environment", in Proceedings of the 1st International Workshop on Persistent object Systems: Data Types and Persistence, Appin, Scotland (Aug. 1985) PPRR-16-85*, 1-24.

**Atkinson & Morrison 85b** Atkinson, M.P. and Morrison, R. - "Procedures as Persistent Data Objects", *ACM TOPLAS* 7, 4, (Oct.1985) 539-559.

**Atkinson & Morrison 87** Atkinson, M.P. and Morrison, R. - "Polymorphism, Type checking and Labels in a Persistent Object Store", in Proceedings of the 2nd International Workshop on Persistent Object Systems: their Design, Implementation and Use. Appin, Scotland (Aug. 1987) PPRR-44-87*.

**Buhr & Zarnke 87** Buhr, P.A. and Zarnke, C.R., - "Persistence in an Environment for a Statically-Typed Programming Language", in Proceedings of the 2nd International Workshop on Persistent Object Systems: their Design, Implementation and Use, Appin, Scotland (Aug. 1987) PPRR-44-87*.

## Bibliography

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

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

or

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

## Books


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

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

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

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


## Published Papers


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

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

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

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

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

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

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

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

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

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

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

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

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

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

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

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

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, NO. 2, (Summer 1986) - see PPRR-23-86.

Kulkarni, K.G. & Atkinson, M. P.
"Implementing an Extended Functional Data Model Using PS-algol", Software - Practise and Experience, Vol. 17(3), 171-185 ( March 1987)

**Buneman & Buneman,** O.P. Data types for Database programming in proceedings of the 1st International Workshop on Persistent Object Sysyems: Date Types and Persistent, Appin, Scotland (Aug. 1985) 285 - 98.

**Buneman & Atkinson 86** Buneman, O.P. and Atkinson, M.P. - "Inheritance and Persistence in Database Programming Languages", in Proceedings of ACM SIGMOD CONF. '86, Washington, USA, (May 1986).

**Cardelli 84** Cardelli, L., "Amber", *Technical Report*, AT&T, Bell Laboratories, Murray Hill, N.J. USA, 1984.

**Cooper & Atkinson 87** Cooper, R.L. and Atkinson, M.P. - "Requirements Modelling in a Persistent Object Store", in Proceedings of the 2nd International Workshop on Persistent Object Systems: their Design, Implementation and Use, Appin, Scotland (Aug. 1987) PPRR-44-87*.

**Cooper *et al* 86** Cooper, R.L., Atkinson, M.P. and Blott, S.M. - "Using a Persistent Environment to Maintain a Bibliographic Database", PPRR-24-86*.

**Cooper *et al* 87** Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane D. - "Constructing Database Systems in a Persistent Environment", in Proceedings of the 13th International Conference on Very Large Data Bases, Brighton, England (Sept 1987), 117-125.

**Dearle & Brown 87** Dearle, A. and Brown, A.L. - "Safe Browing in a Strongly Typed Persistent Environment", PPRR-33-87*.

**Philbrow *et al* 87** Philbrow, P., Armour, I., Atkinson, M.P. and, Livingstone J. - "A Device-independent Output Statement", to be submitted to *ACM SIGPLAN Notices*.

**PPRR12** "The PS-algol Reference Manual: Fourth Edition", PPRR- 12 - 87

**Richardson *et al 87*** Richardson, J.E., Carey, M.J., DeWitt, D.J. and Schuh, D.T. - "Persistence in Exodus", in Proceedings of the 2nd International Workshop on Persistent Object Systems : their Design, Implementation and Use, Appin Scotland (Aug. 1987) PPRR-44-87*.

*Persistent Programming Research Reports (PPRRs) are available from the Computing Science Departments at the Universities of Glasgow and St. Andrews, Scotland.



Cooper, R.L. & Atkinson, M.P.
"The Advantages of a Unified Treatment of Data", Software Tool 87: Improving Tools, Advance Computing Series, 8, 89-96, Online Publications, June 1987.

Atkinson, M.P, Morrison, R. & Dearle, A.
"A strongly typed persistent object store", 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, California (September 1986).

Atkinson, M.P., Morrison, R. & Pratten G.D.
"PISA : A persistent information space architecture", ICL Technical Journal 5, 3 (May 1987),477-491.

Atkinson, M.P. & Morrison, R.
"Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Cooper, R. & Atkinson, M.P.
"Requirements Modelling in a Persistent Object Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Wai, F.
"Distribution and Persistence". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Philbrow, P.
"Associative Storage and Retrieval: Some Language Design Issues". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Guy, M.R.
"Persistent Store - Successor to Virtual Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Dearle, A.
"Constructing Compilers in a Persistent Environment". Presented at the 2nd Internaional Workshop on Persistent Object Stores, Appin, August 1987.

Carrick, R. & Munro, D.
"Execution Strategies in Persistent Systems". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Brown, A.L.
"A Distributed Stable Store". Presented at the 2nd International Workshop on Persistent object Stores, Appin, August 1987.

Cooper, R.L., Atkinson, M.P., Dearle, A. & Abderrahmane, D.
"Constructing Database Systems in a Persistent Environment". Proceedings of the Thirteenth Internaional Conference on Very Large Databases, Brighton, September 1987.

Atkinson, M.P. & Morrison, M.
"Polymorphic Names and Iterations", presented at the Workshop on Database Programming Languages, Roscoff, September 1987.

## Internal Reports

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

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

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

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

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

## Theses

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

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

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

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

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

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

J. Livingstone
Graphical Manipulation in Programming Languages: Some Experiments, M.Sc., University of Glasgow, 1987.

## Persistent Programming Research Reports

This series was started in May 1983. The following list gives those which have been produced at 17th September 1987. Copies of documents in this list may be obtained by writing to the addresses already given.

| | | |
|---|---|---|
| PPRR-1-83 | The Persistent Object Management System - Atkinson,M.P., Bailey, P., Chisholm, K.J., Cockshott, W.P. and Morrison, R. | £1.00 |
| PPRR-2-83 | PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R. | £2.00 |
| PPRR-5-83 | Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G. | £1.00 |
| PPRR-6-83 | A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E. | £1.00 |
| PPRR-7-83 | EFDM - User Manual - K.G.Kulkarni | £1.00 |
| PPRR-8-84 | Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R. | £2.00 |
| PPRR-9-84 | Procedures as Persistent Data Objects - Atkinson, M.P. and Morrison, R. | £1.00 |
| PPRR-10-84 | A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A. | £1.00 |
| PPRR-11-85 | PS-algol Abstract Machine Manual | £1.00 |
| PPRR-12-87 | PS-algol Reference Manual - fourth edition | £2.00 |
| PPRR-13-85 | CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P. | £2.00 |
| PPRR-14-86 | An Integrated Graphics Programming Environment - 2nd edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P. | £1.00 |
| PPRR-15-85 | The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P. | £1.00 |
| PPRR-16-85 | Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R. | £15.00 |
| PPRR-17-85 | Database Programming Language Design - Atkinson, M.P. and Buneman, O.P. | £3.00 |

| | | |
|---|---|---|
| PPRR-18-85 | The Persistent Store Machine - Cockshott, W.P. | £2.00 |
| PPRR-19-85 | Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R. | £1.00 |
| PPRR-20-85 | Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P. | £1.00 |
| PPRR-21-85 | A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D. | £1.00 |
| PPRR-22-86 | Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P. | £1.00 |
| PPRR-23-86 | Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A. | £1.00 |
| PPRR-24-86 | Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M. | £1.00 |
| PPRR-25-87 | Applications Programming in PS-algol - Cooper, R.L. | £1.00 |
| PPRR-26-86 | Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P. | £1.00 |
| PPRR-27-87 | A Context Sensitive Addressing Model - Hurst, A.J. | £1.00 |
| PPRR-28-86b | A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A. | £1.00 |
| PPRR-29-86 | A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P | £1.00 |
| PPRR-30-86 | Data Types for Data Base Programming - Buneman, O.P. | £1.00 |
| PPRR-31-87 | An Introduction to PS-algol Programming (third edition) - Carrick, R., Cole, A.J. & Morrison, R. | £1.00 |
| PPRR-32-87 | Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A | £1.00 |
| PPRR-33-87 | Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L. | £1.00 |
| PPRR-34-87 | Constructing Database Systems in a Persistent Environment - Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D. | £1.00 |
| PPRR-35-87 | A Persistent Architecture Intermediate Language - Dearle, A. | £1.00 |
| PPRR-36-87 | Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D. | £1.00 |
| PPRR-37-87 | PS-algol Machine Monitoring - Loboz, Z. | £1.00 |
| PPRR-38-87 | Flexible Incremental Bindings in a Persistent Object Store - Morrison, R., Atkinson, M.P. and Dearle, A. | £1.00 |
| PPRR-39-87 | Polymorphic Persistent Processes - Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J. | £1.00 |
| PPRR-40-87 | Andrew, Unix and Educational Computing - Hansen, W. J. | £1.00 |
| PPRR-41-87 | Factors that Affect Reading and Writing with Personal Computers and Workstations - Hansen, W. J. and Haas, C. | £1.00 |
| PPRR-42-87 | A Practical Algebra for Substring Expressions - Hansen, W. J. | £1.00 |
| PPRR-43-87 | The NESS Reference Manual - Hansen, W. J. | £1.00 |
| PPRR-44-87 | Persistent Object Systems: their design, implementation and use. (proceedings of the Appin workshop August 1987) - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R. | £20.00 |
| PPRR-45-87 | Delayed Binding and Type Checking in Database Programming Languages - Atkinson, M.P., Buneman, O.P. & Morrison, R. | £1.00 |
| PPRR-46-87 | Transactions and Concurrency - Krablin, G.L. | £1.00 |
| PPRR-47-87 | Persistent Information Space Architecture - PISA Club Rules - Atkinson, M.P., Lucking, J.R., Morrison, R. and Pratten, G.D. | £1.00 |
| PPRR-48-87 | An Event-Driven Software Architecture - Cutts, Q. and Kirby, G. | £1.00 |
| PPRR-49-87 | An Implementation of Multiple Inheritance in a Persistent Environment - Benson, P.J., D'Souza, E.B., Rennie, I.S., Waddell, S.J. | £1.00 |
| PPRR-50-87 | A Distributed Stable Store - Brown, A.L. | £1.00 |
| PPRR-51-87 | Constructing Compilers in a Persistent Environment - Dearle, A. | £1.00 |
| PPRR-52-87 | Lgen, Pgen and Sgen - Language Development Tools for a Persistent Programming Environment - Blott, S.M. and Campin, J. | £1.00 |
| PPRR-53-87 | Polymorphic Names and Iterations - Atkinson, M.P. and Morrison, R | £1.00 |
| PPRR-54-87 | A Requirements Modelling Tool Built in PS-algol - Cooper, R.L. and Atkinson, M.P. | £1.00 |

PPRR-55-87 A Persistent Software Database with Version Control - Cooper, R.L. and Atkinson, M.P. £1.00

PPRR-56-87 User Interface Tools in PS-algol - Cooper, R.L., McFarlane, D.K. and Ahmed, S. £1.00