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

Delayed binding and Type Checking in Database Programming Languages

M. Atkinson, P. Buneman & R. Morrison

Persistent Programming
Research Report 45
September 1987

Alan Dearle.

Delayed Binding and Type Checking in Database Programming Languages

Malcolm Atkinson [1], Peter Buneman[2] and Ronald Morrison[3]

Abstract

Attention is drawn to the issues regarding the timing and performance of binding and type checking in database programming languages. Examination of the relationships between long term data and programs leads to the recognition of four patterns of binding. A representative sample of such languages is reviewed to show that some give explicit control over binding without loss of strong typing, whereas others have no mechanisms to support the binding we deem necessary. We conclude that it is both possible and desirable to provide constructs to explicitly identify bindings that should be delayed, without loss of strong type checking. Such provision has an impact on the type equivalence rules available for database programming languages.

1
Prof. M.P. Atkinson,
University of Glasgow,
Dept. of Computing Science,
Glasgow
G12 8QQ
SCOTLAND.
TEL. NO. 041 330 4359.

2
Prof. O.P. Buneman,
University of Pennsylvania,
Dept. of Computer and
Information Science,
Moore School/D2,
Philadelphia Pa 19104
TEL. NO. (215) 898-7703

3
Prof. R. Morrison,
University of St. Andrews,
Dept. of Computational Science,
North Haugh,
St. Andrews KY16 9SS
Fife,
TEL. NO. 0334 76161.

1. Introduction

Many proponents of modern programming language design believe that languages should be entirely statically bound and statically type checked [Milner 78, Milner 84, Liskov *et al.* 81, Albano *et al.* 85a]. In a statically bound language all the information necessary to determine the types associated with all language constructs is available from inspection of the source text. There are three major advantages to this static binding:

- early detection of a large number of programming errors - which benefits program development and improves the safety of the final product;
- factoring out checks that would otherwise be made repeatedly at run-time to ensure that operations are applicable - which reduces execution time; and,
- removing the need to store type information (data description) with data - economising on storage volume and data-bus traffic.

In contrast, many database interfaces are entirely dynamically bound and checked. For example, with most Codasyl interfaces operations may be revalidated for each record to which they are applied, and in many relational systems details of a relation's type will be rechecked for each use of that relation. Similarly, there are many languages in which the binding (determining which location a name refers to) and the type validation is dynamic, depending on the execution sequence of the program. Examples are: LISP [MaCarthy *et al.* 62], POP-2 [Burstal *et al.* 71], APL [Iverson 79], Smalltalk [Goldstein & Bobrow 80] and Prolog [Clocksin & Mellish 81].

In this paper, we will demonstrate that some dynamic binding is necessary in database programming languages (DBPLs). We will review binding in existing DBPLs, showing that some have adopted an entirely static approach, but that others have adopted a combination of static and dynamic binding. The correct mixture of these two forms of binding for DBPLs we consider a research issue. We present our understanding of the options and criteria.

For a more general survey of DBPLs the reader is referred to [Atkinson & Buneman 87], and for a tutorial on binding and typechecking to [Tennent 81]. We take it as understood that a valid long term goal is the development of an integrated DBPL, which services all the needs of long term and short term data storage and manipulation. That case is argued elsewhere [Atkinson *et al.* 86, Atkinson & Morrison 86, Atkinson & Morrison 85a, Atkinson *et al.* 83].

2. Binding reconsidered

In traditional programming languages and database systems there are a number of binding mechanisms which are often not easy to comprehend or use. The action of binding associates a name with a value. This simple action is complicated by the fact that we may perform many bindings simultaneously. For example in binding a program to a database we may incrementally bind names in the program to values in the database as the program executes, or we may bind all the names to values at the same time. A mixture of these mechanisms is also possible.

The binding may also be performed statically or dynamically. Static binding occurs when the compiler arranges the binding. The bindings once established are immutable and thus any alteration to the database schema or the programs that run against the database, may require re-compilation to re-establish the bindings. Dynamic bindings do not suffer from these compilation overheads as they are only established as they are required.

In establishing the binding there are two major ways in which the name is interpreted or scoped. With static scoping the name is interpreted in its lexicographic context whereas with dynamic scoping the name is interpreted in the evaluation context. Static scoping is commonly found in the Algol family of languages. Dynamic scoping is found in Lisp, APL and in systems that bind different databases to the same name depending on the context the program is run in.

The issue of type checking is also present in binding. We would wish the system to check the valid use of data when it is bound. This can be done statically or dynamically depending on the time of the binding.

One final issue concerned with binding is that the name may be bound to a constant or variable. If variable, the referenced value may change without altering the binding [Tennent 81].

2.1 The delayed binding mechanism

The delayed binding mechanism for database programming languages is operated as follows. As databases are constructed they are made to conform to a type definition and naming scheme. The definitions of the objects in the database are stored with the database in the dictionary or schema.

Program construction proceeds in a similar manner with the program components conforming to a separate type definition and naming scheme. These definitions are stored as part of the program.

For programs to work against databases the type system and naming scheme in each case must be founded on a common model and stored in an agreed form. Thus, when any program is used with any database, at some time before any operation is performed, the two descriptions are compared to verify that they are equivalent or compatible definitions in the two contexts. Identifying good common models, canonical forms and the rules for determining compatibility and equivalence is the subject of current research. A central issue in this paper is the time at which the compatibility check is performed, and the extent to which the programmer may control its timing and its extent.

3. The case for delayed binding in DBPLs

In rather general terms, we may expect delayed binding to be useful, since, with delayed binding the interpretation of names depends on the history of the computation. In databases, we are recording the history of a number of computations, and then making that history available to subsequent computations. As we revise values and definitions in the database, it is not practical to recompile all programs to ensure they still bind correctly before they are used. This is simply an echo of the early statements about data independence.

To make the case more specific the following applications of DBPLs are considered:

i) re-use and distribution of programs;
ii) selection of databases;
iii) combination of information from several databases;
iv) incremental data and program definition.

Each of these will be elaborated in turn.

3.1 Re-use and distribution of programs

Often a program is economic to develop only if it can be used against many databases. For example, a CAD program would be used for many projects, and, normally a separate database would be used for each project.

3.1.1 Static Mechanisms for reuse

To reuse the program it must be bound to each database.
If this is to be achieved by static binding two methods exist:

i) the program can be delivered in source form, and compiled in the context of each database;
ii) the program can be compiled in the context of a type definition (as in Poly [Matthews 85]) which is used to declare each database.

The first method runs into two problems:

a) the cost of reprocessing the source, and performing the binding may be unacceptably high, and may involve resources not available at each database;
b) it involves distributing programs in source form (or something similar) which means they reveal techniques which could then be more easily copied by rivals.

In any case, conducting the compilation is just another mechanism for doing the delayed rebinding. The question remains - is this recompilation built in, or achieved manually using external facilities.

The second method means that each instance of a database is declared in the context of the one type declaration. With static binding, this means in the scope of the type definition. With present technology that scope is limited to one computer system. This is clearly unacceptable as the databases will typically belong to different organisations that will wish to support the databases on different machines with different operating, protection etc. policies. This requirement for independent ownership militates strongly against this form of static binding. It may be possible to utilise this approach using distributed database technology, since it may provide a single "central" definition, of many database instances - but this is not complete independence, adherence to the conventions of the particular network is necessary and the problem reappears between networks.

3.1.2 Program and data as an ADT

Another tactic could be used to maintain static binding, by making the program the unit of ownership. Here, the would be database owner would possess a program. Operation of the program would then generate the database in terms of the definitions within the program. Provided all the

required operations on the database are supported by this program, then all use of the data requires no further binding. This is the approach taken by systems with a simple data model, e.g. spreadsheet systems. However, apart from very simple applications, it is not possible to anticipate all operations. The normal arrangement is to acquire new programs to work against the data as their need is perceived.

Other restrictions resulting from this approach are: the difficulty of combining data originating from more than one database; and the problem of replacing the software with improved versions. Both of these are discussed below. But, it should also be noticed that the relationship between data and program proposed here, is precisely the same as the relationship between the encapsulated data of an instance of an abstract data type (ADT) and the program that implements the operations on it. There are differences of scale, but the same problems arise. If we view the data and programs as an ADT, and a good set of basic operations is provided by the program encapsulating the data, then all intended operations can be achieved by composing those basic operations. If the composition is done by program (procedural languages, operating system command language etc.) then the problem of binding reappears. If it is done by hand, only trivial problems can be solved before the task is too laborious and error prone. A relational system is an example, the encapsulated data being instances of relational data bases, and the operations being the usual algebra, **plus** schema update operations. The operations are composed by the query language. As soon as queries become so complex that they are stored for reuse, and the schema may have been amended over that time, these issues of binding cannot be trivially solved.

From this section we conclude that delayed binding is required both to allow programs (particularly proprietory software) to be distributed and used on many independent databases, and to allow programs to be collected incrementally for use on a database.

3.2 Selection of databases

If a program is statically bound to a database then there is no choice about which database the program will operate against. Where there are independent databases (e.g. the CAD projects) they may be named in quite complex and user determined ways. It is then desirable to start the program, and as a consequence o' a dialogue with the user, determine which database (or databases) are to be used.

The more generic the function performed the more likely it is that the user will dynamically choose the subject database(s). To provide this requires delayed binding.

3.3 Composition of databases

Frequently, it is desirable to combine data from more than one database: statistics from two sources; parts from one project to be used on another; the captured company's and principal company's databases after a merger; etc.

Since a program cannot be statically bound to all possible subsets of databases, this requires delayed binding.

For example, when reusing part definitions or previous designs, it is likely the data will have been prepared using the same software, and that therefore (a) relatively large chuncks will comply with the current data definitions in force, and (b) this is a typical case where the user will make delayed decisions. When in the midst of a current design, then the user will perceive the need for a remembered part and decide which database to search.

3.4 Incremental definition

We have already observed the need to collect programs to perform operations when a need is recognised. For example, new methods of evaluation, manufacture and management may become relevant while the data is in existence. New programs will be installed to support the new methods.

But program is not special. There is no reason to assume that data definition can be completed once and for all, any more than program can be. The body of data, and hence the definitions which describe it, must also evolve to meet changing needs. To do this, with "live" data and programs, requires delayed binding.

This evolution of definition is divided into three parts: *extension* , *replacement* and *deletion*. For extension it must be possible to add new definitions and the corresponding data, and refer to it via the pre-existing data. For example, if this new data gave an additional specification of an aircraft window, one would expect to use the existing data to select a version of an aircraft and reach a particular window, in order to get to the new data appertaining to that window. The alternative, of building a parallel

description of the context which supports the new data, is unacceptable because successive changes would lead to a combinatorial explosion of replicated data, with unacceptable update anomaly problems. It is therefore necessary that existing meta data and the corresponding objects can accommodate extension and that exisiting programs run correctly on the extended data. This requires forms of incremental delayed binding to both the data and the schema.

Replacement is necessary to repair errors. For example, a CAD program may contain errors, and a new one be supplied to replace it. It is essential that this new program be installed at each database without loss of the design effort expended so far. This requires that, in some way, it be bound with the pre-existing database. If the errors also include erroneous design of parts of the data structure then a more difficult replacement is required.

Deletion will occur when a data structure subgraph has fallen into disuse. If this hypothesis is valid (the normal case) then no program will actually try to use the data again. Either all programs have to be recompiled to verify this validity, or it must be verified incrementally by delayed binding. Which is more economic depends on the volume of program and its frequencies of reuse and recompilation.

A common method of structuring programs is to encapsulate data in modules, presenting only the legitimate operations as an abstract data type. Such modules themselves may contain errors which become apparent in long lived data. Recovery requires replacement of the module definition, possibly with redefinition of the interface, and, again, there will be occasions when the encapsulated data must be carried forward to be re-used with the definition. We do not know of any module mechanism yet developed which would support this form of rebinding.

The need to recover from errors and for data definition to evolve has always existed for programs and data. The development of large bodies of data combined with large suites of program make solutions based on rebuilding which have been used in the past uneconomic. Ad hoc solutions prevent the development of easily understood programming languages. This paper traces the search for consistent treatments and presents our suggestions.

4. Treatment of binding in DBPLs

In the preceding section the need for some form of dynamic binding and delayed type checking in database programming languages was established. A number of languages are now reviewed to consider their treatment of this issue.

4.1 Pascal

In Pascal [ISO 82] the type **file** may be used to access external (persistent) data; it may be parameterised by another type as in **file of** *char* ; **file of** *integer*; **file of record** *a*: *integer*; *b*: *real*; *c*; *bool* **end**. Persistence is provided by mapping these files to those of the supporting operating system. The binding is performed by one of two methods, shown in figures 1 and 2.

```
program X (f1 , f2 );
...
var f1 : file of char ; f2 : file of integer ;
...
end.
```

Figure 1: binding external files to internal variables in Pascal

```
program Y ;
    ...
    var f1 : file of char ; theFile : packed array [1 .. 32] of char ;
    ... {carry out dialogue with user & obtain a value for theFile }
    while notFinished do
        begin
        openFile (theFile , f1 );
        ... {do something with the file, then get another name}
         end {repeat for the next file}
    ...
    end
```

Figure 2: selecting and binding files in Pascal

For the program *X* in figure 1, there are two file parameters. When a user of the program wishes to run *X* two parameters must be supplied with the operating system command e.g.

```
run X myFile herFile
```

The operating system will identify two specific files *myFile* and *herFile* and the run time support system will bind these to variables *f1* and *f2* in the program. To avoid nonsense executions it should perform a delayed type check at that time to ensure that the first is a file of characters and the second is a file of integers. If this mechanism is implemented, then the dynamic compositions of program and data described in section 3.1 are met, at least for this limited class of data structures. The Berkeley Pascal implementation [Joy 83], searches the current directory for files of the same name as the given parameters, and hence the binding is based on dynamic scoping.

For the program *Y* in figure 2 dynamic selection of the files to be processed is illustrated. Each time around the **while** loop a different value for *f1*, identified by the current value of *theFile*, is used. The procedure *openFile* must perform the binding of the actual file to the variable and perform a check to ensure it is a character file. If this mechanism were implemented it would meet the selection binding required in section 3.2. This method is not part of the formal definition, but something like it may appear in implementations. There is difficulty in providing procedures such as *openFile* in Pascal; since it offers no polymorphism, the procedure is specific to the type of *f1*. Hence, it can only be provided by the compiler writer, so the programmer moving a program cannot simply hide the local form of this operation in a procedure

The fact that more than one file may be bound by either mechanism meets the needs of 3.3. In Pascal there are limitations, especially in implementation, on the types that may parameterise.

4.2 Ada

Ada takes a traditional view of persistence by providing a library for persistence over program and a file store for persistence over data, in contrast to the view that one mechanism suffices for both [Atkinson & Morrison 85a].

The library contains all the statically bound components produced by the compiler and a loader combines these components into programs. Component reuse for programs is achieved by the loader using a library unit in many programs. This however is a static mechanism and is subject to the criticisms already made in Section 3.1.

Persistence for data is provided by files. Access to files is first obtained by instanciating a generic package of the particular access method, **direct** or **sequential** and then opening a file of the type returned by the package. For example

package *Sorted_integers* **is new** *Sequential_io* (*INTEGER*);

yields a package of sub-programs and types. The type *Sorted_integers.FILE_TYPE* is now the type of the file and may be used in the file accessing procedures. Figure 3 shows a generic package that will copy any sequential file.

```
generic
    type ELEMENT is private ;
procedure Copy_any_sequential_file (Source, Destination: STRING) is
    package Any_sequential_io is new Sequential_io(ELEMENT) ;
    use Any_sequential_io ;
    Source_file, Destination_file: FILE_TYPE ;
    Item: ELEMENT ;
begin
    Open (Source_file, In_file, Source) ;
    Create (Destination_file, Out_file, Destination) ;
    while not End_of_file (Source_file) loop
        Read (Source_file, Item) ;
        Write (Destination_file, Item) ;
    end loop ;
    Close (Source_file) ; Close (Destination_file) ;
end Copy_any_sequential_file ;
```

Figure 3: copying any file in Ada

This generic procedure can be instanciated to operate on objects of any type. For example

```
type PRIMARY_COLOUR is (Red, Yellow, Blue)
procedure Copy_pc is new Copy_any_sequential_file
                                        (PRIMARY_COLOUR) ;
```

The instanciation is performed at compile time. However, the Open statement dynamically binds the external file, given by name, to the internal file. The rules for determining to which file the external name belongs are implementation dependent and therefore the scoping may be static or dynamic. Selection of databases and simultaneous opening of several files can be supported by this mechanism since the file name is a string and not statically determined meeting requirements 3.2 and part of 3.3.

A number of points arise from Ada's view of file. First of all the mechanism only works well for certain types of file. The implementation of **access** types on I/O is implementation dependent and would presumably cause problems across heterogeneous implementations. Types with associated sub-programs, as we would normally find in packages have to be completely re-defined in each program that uses them. This method of type definition is more normally found with a structural equivalence rule and exposes a hole in the name equivalence of Ada This might impede the requirement to combine data that originated independently (3.3).

Each instantiation of the packages *Sequential_io* and *Direct_io* yields a unique file type. If this were truly the case then a file created in one program could not be used in another since the file types would differ. Although this is the defined rule it is clearly not the intention and indeed in implementations the file types are equal. This is really structural equivalence. As the implementors have provided only name equivalence for the rest of the implementation they may be tempted to limit the check to simple cases either restricting I/O or reducing security.

These rules for file type match prohibit the kind of incremental definition referred to in Section 3.4.

4.3 Pascal/R

Pascal/R is an extension of Pascal to include a relational type [Schmidt 77]. A related, derived, language is Modula/R [Koch *et.al* 83].

Figure 4 shows the outline of an example program in Pascal/R.

The binding mechanisms here are similar to those in Pascal. The type of a relation variable is declared using the **database ... end** construct, which is similar to a record type except that the fields must all be of various relation types. The declaration of the variable *db1* ensures that the required type is known in the program. Its appearance in the parameter list as *db1** (the * identifies this is a database, not file parameter) allows an actual database to be substituted when the program is run.

```
program X ( db1* );
type
relationType1 = relation <a, b > of record  a: int; b:string;... end;
...
var   db1  : database
  r1:  relationType1;
         ...
         end;
with db1 do
         begin
         ...  {the body using the relations in db1}
         end
end.
```

Figure 4: binding a database to a program in Pascal/R

The type check is performed as the program begins, comparing *db1's* type with the description of the stored database. After that, there is no further type checking, the rest having been performed at compile time.

Again this meets the requirements of 3.1, and with minor changes to allow multiple parameters, it would meet requirements 3.3. In this second case there would be no problem over clashing names (a common problem when combining databases) as the Pascal naming system localises the scope of field names, and provides ways of disambiguating their use (i.e. *db1.r2* *db2.r2*).

To dynamically select databases during execution (3.2) raises more problems. Some procedure *openDatabase* taking a database name and writing to a database variable parameter is needed. The programs will need different types for the database variable, possibly within the same program. Only by generating procedures associated with each database or type (c.f. the suggestion of Buneman [Buneman 82 *et al.*]) can this problem be overcome. But this still does not allow dynamic selection, except from a predefined subset, as those procedures cannot be dynamically created and included in the Pascal program.

Pascal/R begins to offer some adaptive binding (3.4) as, in principle at least, the type check between the program's database variable and the database type need not be an exact match. For example, the program's database variable could hold a subset of the fields that appear as relations in the actual database. This view mechanism permits extension of data and allows existing programs to run unchanged and without recompilation. More complex views, e.g. projections and joins run into two difficulties: in general they cannot be inferred and so need to be explicitly stated for each program to database binding; and, also in general, updates performed in the program then become illdefined [Rowe & Shoens 79]. If foreign keys are used to relate new and old structure this will also support local growth of data (c.f. the aircraft windows section 3.4 example). There is no information hiding in Pascal/R so that the ADT revision cannot arise.

4.4 PS-algol

PS-algol [PPRG 87] is an experimental language, with emphasis on the orthogonal provision of persistence. To provide this, it uses a database construct, an index construct and pointers (type **pntr**) whose referends may be any instance of any structure class type. Data remains in existence for as long as it is reachable either from an active program or from one or more databases. Example PS-algol programs are shown in figures 5, 6 and 7.

The program in figure 5 illustrates that requirements 3.1 and 3.3 are met by PS-algol. The declaration of *db1* initialises that value to the result of *openDatabase* performing a binding of that identifier to the database referred to by "MalcolmsDB", and gives *db1* the type **pntr**. *db1* now refers to the root object of that database, this is passed as a **pntr** to *lookup* in the initialising expression of the declaration for *a*. *lookup* expects data structures representing a table from strings to **pntr** values. By convention the root object of a database is such a data structure. Though this is checked by execution of *lookup,* the particular form of data structure implementing the table is not a concern of this program, and is only recorded where *lookup* is defined and with the data. The resulting value for *a* is also a **pntr**. *b* receives its value by a parallel course from a separate database.

```
when databaseOpenError do recoveryAction
let db1 = openDatabase ("MalcolmsDB")
let db2 = openDatabase ("PetersDB")
let a   = lookup ("aValue", db1 )
let b   = lookup ("bValue", db2 )
structure S1 (int x ; proc (int → int)y )
let ax  = a (x) ; let bx = b (x)
let ay  = a (y) ; let by = b (y)
print ax, by (ax) , bx, ay (bx)
```

Figure 5: An example of a PS-Algol program using two databases

The declaration of a structure class *S1* is a specification by the programmer of the referend type expected. The field names *x* and *y* are then used to obtain values from *a* and *b* by subscripting. At each of these subscriptions a delayed type check is performed on the **pntr** values. Associated with each **pntr** value is the structure class definition that was used to construct the value. Associated with each field name is the structure class definition that entered it into scope. If these definitions do not match an exception is raised, otherwise the field access proceeds. As the field name has a type, the subscripting expression and hence the declared identifier is given a specific type. Hence subsequent use, e.g. the **print** statement, including the function applications, may be statically checked.

This program has illustrated the dynamic binding of two databases to the program, a type match between the program and each of the databases, and delayed type checking. The delay in determining referend types permits the kind of extension referred to in 3.4. In this example, the *lookup* code has given associative access to data whose types were not defined when the *lookup* code was defined. Consequently items with new definitions may be put in the database without affecting exisiting programs. In particular, note that only the structures used from a database are actually declared in the program (and their scope may be limited) so that programs are sensitive only to the definitions of the data they use.

```
let db1 = openDatabase ("RonsDB")
let it  = lookup ("extender", db1 )
structure PFE (int a ; real b ; ... ; pntr extra )
structure Epack (proc (pntr) p )
let extend = it(p)
...
while moreToDo do
    begin
    ...  ! conduct a dialogue with the user to determine DB to process
    let theNextDB = readString ( )
    let theDB = openDatabase (theNextDB )
    ...  ! apply extend to each PFE in this DB
    commit ( )
end
...
```

Figure 6: A PS-algol program dynamically selecting databases and extending the structure in them

In figure 5, the dynamic selection of databases in PS-algol is illustrated showing that it satisfies requirement 3.2. As the universal union over structures (**pntr**) is used as the result of the binding operation there is no difficulty over typing the *openDatabase* procedure. A further step towards meeting requirement 3.4 is also illustrated. At the time when some structure, *PFE* , was first defined (i.e. instances of it were put in a database) the programmer had planned for extension by including the field *extra* . Since the referend does not require specification at that time, this field can be used for **any** extension. The program in Figure 5 takes a procedure which fills in this extension for all the *PFE* s in the databases selected by the user.

The code for a procedure which could be the value for *extend* in Figure 5 is given in Figure 6.

```
let anExtender = proc (pntr aPFE )
    begin
        structure PFE (int a; real b; ··· ; pntr extra )
        structure PFME (string x, y, z; ···; pntr more )
        let aPFME = PFME ("initial x", "initial y", "initial z", ···, nil)
        aPFE (extra ) := aPFME
    end
```

Figure 7: a procedure to extend a *PFE* structure

Other procedures could add other fields to instances of *PFE* and typically they would set up the initial values by consulting the user. Later similar procedures could extend the *PFME* instances by assigning to their *more* field.

Frequently, in actual database designs we find character fields in records for this purpose. Then in the programs one finds obscure code to cram data into these extension fields, with a complete loss of independence and type checking, in contrast to the provision here. Further, this kind of extension may itself be extended, whereas the overpacked character field eventually defeats the ingenuity to encode.

The programs that previously processed *PFE* s will operate unchanged. All these solutions depend on the delayed binding of the referend's class definition to **pntr** values. We believe it correct that the programmer plan for expansion rather than incur the cost of flexibility everywhere.

The ADT mechanism in PS-algol is implemented by returning procedures packaged in a structure class from a procedure [Atkinson & Morrison 85a]. The hidden data is accessed by statically bound references to locations within the activation record of the generating procedure, therefore, revision of ADT definitions, reusing existing instance data, is not supported. Practical experience with PS-algol's binding mechanisms and their exploitation to give all the facilities described here, is reported in a companion paper [Cooper *et al.* 87].

4.5 Amber

The language Amber [Cardelli 84a] introduces an explicit universal type, **dynamic**, to deal with program and database binding, and uses inheritance polymorphism to deal with the extension of data [Cardelli 84b]. This is illustrated in figure 8 (see page 17).

On some previous occasion the definition of the database type, *DBT*, and an exemplar value derived by *typeOf* (<a constructed example>), *DBTval*, are stored in a module "DBdef". That module is reopened in this program by **import**. A dynamic value is then obtained from "DBfile" by *import*, the program then checks if its type description data structure matches the exemplar type, to avoid an error during the **coerce** operation. Very few operations are available on a **dynamic** value; the **coerce** operation projects it to the specified type, and if that succeeds (the types match), then the operations of that type become available. For the rest of the inner block

```
import "DBdef"

     type DBT
     value DBTval
let newValue = import ("DBfile")
if typeOf (newValue) = DBTval then
            let db = coerce newValue to DBT
            ... {use that db - e.g. perform updates}
            do export ("DBfile", dynamic db)
else
            ...
```

Figure 7: Illustrating Amber's database binding mechanism

(shown by indentation) static binding and typechecking prevails. At the end of the block, the monadic operator **dynamic** reassociates the type description with the value, and returns a value with the extensible universal union type **dynamic**. This mechanism clearly meets requirement 3.1, and 3.3 without difficulty. In the case where all the selected databases have one of a predictable set of types, it also meets requirement 3.2. The advantage of Amber's approach is that it gives a clear syntactic identification of the moment of binding and typecheck, and therefore makes the programmer aware of when the option of delaying these is being exploited.

Amber may not actually meet the requirement for matching independently developed components. This depends on whether copies of values such as *DBTval* are equal to one another or whether identity is required. If identity is required, then a program developed at one site could not match data which was built at another site, and hence bound to a different instance of the description structure.

It is not clear whether the data definition (e.g. *DBT* and *DBTval*) may be extended in Amber, and, therefore, whether Amber addresses requirement 3.4. The type matching mechanism does however allow code to be written which will process extended data. A record type in Amber is declared by $<< f_1: \gamma_1, f_2: \gamma_2, ..., f_n: \gamma_n >>$ where f_i are field names and γ_i their types. Any record instance matches all the types described by all subsets of its field : type pairs. For example, a record fully described by << a: int, b: bool, c: real>>

would match <<a: int>>, <<b: bool>>, <<a:int, c: real>>, etc. Consequently a procedure written to process a given record type e.g. our example, will also process extended records such as <<a: int, b: bool, c: real, d: string>> with exactly the same code.

4.6 Napier

Amber and PS-algol were fairly similar in their semantics of binding to a universal type, though PS-algol limits **pntr** to a subset of all types and makes the projection and check implicit while Amber allows **dynamic** to range over all types and makes the projection explicit. Napier [Atkinson & Morrison 85b] also attempts to make the point of binding explicit, but sets out to overcome three difficulties that arise with the previous two languages:

i) to avoid a single space of field names;
ii) to explicitly identify the extension operations; and
iii) to avoid appeal to external name handlers.

Each of the difficulties will be reviewed first.

i) In PS-algol only one structure containing a particular field name may be in scope at once, otherwise it is not possible to infer the structure type from the field name. Since it is impossible to predict which combinations of structures will be used together, it proves to be good practice to keep all field names distinct. This is generally tedious and it makes merging collections of data very difficult. The same requirement arises in Amber for a slightly different reason. If names are reused, Amber may infer type hierarchy where none was intended. It is not clear whether the resulting program behaviour would be incorrect.

ii) In Amber the arrangements for extending existing data is not defined, and is certainly not explicit. In PS-algol it involves a particular usage of a **pntr** field and after extension the new data has to be accessed in a different way from the original fields. After a few amendments this access path will involve several indirections, leading to obscure code. There is no mechanism for withdrawing aspects of the data.

iii) In both Amber and PS-algol, names, encoded as strings, occur which have an interpretation external to the program, e.g. "DBdef", "DBfile", "RonsDB", etc. They are usually passed to the surrounding operating system for interpretation. This is unsatisfactory, as the rules of interpretation cannot be specified within the language, and hence suites of software are not portable.

(In general, language specification and design should be developed to eliminate these appeals to adhoc, potentially illdefined, external mechanisms.)

Persistence in Napier is similar to that in PS-algol except for the identification of its root. Data is kept for as long as it is reachable, determined by the transitive closure of references to objects beginning from the persistent root, called *PS*. Data not included in this transitive closure, and not accessible from any active program may be automatically deleted by the support system.

The distinguished point in the object graph, *PS*, has type **env** in Napier, short for environment. Objects of type **env**, environments, are collections of bindings, that is, sets of name-value-type-constancy quadruples. If β is the set of all possible bindings then the type **env** is the powerset β (i.e. a value of type **env** is a particular subset of β). Two operations allow the value of an environment to be changed:

insertion adds a new name-value-type-constancy quadruple to the environment.
and
deletion removes such a quadruple from the environment.

Notice that the type does not change as a result of these operations, whereas a value of record or structure type holds a constant subset of β, and the declaration of the type, by record or structure limits the permitted names and corresponding values, so that these operations are not possible. Consequently, a programmer making the choice between **env** and **record** to store some value is choosing whether the extension/revision operations will be available (corresponding to requirements in section 3.4), and complying again with our principal that the programmer should choose whether to incur the costs of flexibility.

Since any environment, whatever bindings it contains, is the same type, **env**, the other requirements 3.1 to 3.3 are easily met, as is illustrated below. Figure 9 shows how two environments may be created and made persistent, that contain the same data as that used by the PS-algol program in figure 5.

```
let a =  emptyEnv ()            ! create an empty environment
let y := (proc(x :int → int ); x * x) in a
let x := 1 in a                 ! a notation for insertion
                                ! now have two bindings in a
                                ! x bound to an integer variable
                                ! currently holding 1 and y
                                ! bound to a proc(int → int ) variable
                                ! currently holding a procedure which
                                ! squares its arguments
let b = emptyEnv ()             ! similarly for b
let y := (proc(i : int → int ); i * i *i ) in b
let x := 2 in b
                                ! now arrange that a and b persist
                                ! and have names used before
let db1 = a in PS ; let db2 = b in PS
checkpoint ()
```

Figure 9: creating two environments and making them persist

The identifiers *a* and *b* are in the local environment, and are both of type **env**; these bindings are lost at the end of the program. The identifiers *db1* and *db2* (bound to the same values and also of type **env**) are placed in *PS* and consequently continue to exist, and therefore the values they contain will also persist. Note that each of the **let** ... **in** (*environment*) constructs illustrates an extension operation.

Figure 10 shows how some subsequent program could use this data in the same fashion as the PS-algol program in Figure 5.

```
use PS as db1, db2 : env in
    use db1 as x : int ; y: proc (int → int ) in
        begin
           let  ax = x ; let ay = y ! rename
           use db2 as x : int ; y : proc (int → int ) in
               print ax , y (ax ), x , ay (x )
```

Figure 10:. reading data from two environments produced in Figure 9

Figure 11 shows a Napier program to select and process environments (as an analogue of a database) in a similar way to the PS-algol program in Figure 6.

```
use PS as RonsDB,UserDBs : env in
    use RonsDB as extender : proc(env) in
        begin
        ...
        while moveToDo do
           begin
           print "'n Provide name of next DB to process"
           let theNextDB = read [name[env]]
           let theDB = UserDBs (theNextDB )
           use theDB as set : index string to env in
               for each → e in set do extender (e )
           ...
           end
        end
```

Figure 11: A Napier program to extend the data in a user selected database

This program shows environments performing three rôles:

i) As the database - supplanting the table structure used in PS-algol i.e. *RonsDB* , *UserDBs* and *theDB* ;
ii) As the substitute for operating system name resolvers, such as file directories, i.e. *PS* , *UserDBs* ;
iii) As an extensible data structure, substituting for the ad hoc use of **pntr** in PS-algol, i.e. *e*.

To permit the name resolution, the type constructor **name** has been introduced, it is parameterised by a type which indicates the class of objects all such names may name, e.g. **name [env]** is the type of all names that may name an environment. **read** is now a polymorphic nuladic operator which requires a type parameter to indicate the type of result it is required to produce. An environment may be subscripted by a name analagous to subscripting a structure e.g. *UserDBs* (*theNextDB*).

The use of environments and names is further illustrated by figures 12 and 13. Figure 12 shows a definition for an *extender* procedure analagous to that shown in Figure 6; the *extender* defined in Figure 13 exploits more of the flexibility of environments and names illustrating Napier's considerable ability to provide the incremental definition established as a requirement in section 3.4. A fuller definition and discussion of names and environments may be found in [Atkinson & Morrison 87].

```
let PS as RonsDB : env in
    let extender = proc (e : env)
        begin
        let x := "initial x" in e
        let y := "initial y" in e
        let z := "initial z" in e
        ...
        end
    in RonsDB   ! make extender persist in RonsDB
```

Figure 12: A Napier procedure to extend an environment with predefined new name, value, constancy, type quadruples

The code shown in Figure 12 is significantly simpler than that in Figure 7, and the resulting revised data structure no longer requires a different method for accessing new fields.

The code in Figure 12 indicates that in Napier the names themselves may be chosen by the user rather than the programmer. This is significant because of persistence. The user may dynamically introduce names in this way, and later write programs which use those names in the conventional way

```
use PS as RonsDB in
    let extender = proc (e : env)
        begin
        ...        ! code to show the user the value of e
        while moreToDo do
          begin
          print "'n What name?"
          let newName = read [name [string ]]
          print "'n What initial value?"
          let initialValue = read [string ]
          insert newName := initialValue into e
          print "'n Insert more names?"
          moreToDo := affirmativeReply ( )
          end
        end
```

Figure 13: A Napier procedure to allow new quadruplets to be inserted into an environment with user determined names and values

4.7 Statically bound languages

The preceding examples present a progression of attempts to meet the needs of flexible binding. Static binding remains more convenient for the programmer writing an individual program, as it is not then necessary to specify the expectations on the long term data. Here we only draw the reader's attention to the existence of such languages in the DBPL camp. Examples are Poly [Matthews 85] and Galileo [Albano *et al.* 85b, Albano *et al.* 86]. These languages, like many others, such as Lisp, Prolog, APL etc, achieve persistence by saving the workspace and restoring it to resume the session. If they intend to be used to build database applications systems they will need to provide an alternative mechanism for dynamic binding. This could be based on a remote procedure call mechanism [Birrell & Nelson 83] perhaps. But that again introduces dynamic binding and delayed type checking.

5. Cost benefit analysis of delayed binding

There are three major costs:

i) delayed information regarding erroneous operations;
ii) additional descriptive effort required by the programmer;
and iii) implementation costs.

The delay in reporting a mismatch between the bound value and the required type of that value is inevitable if binding is delayed. If this were the case for every binding in the program, then the program could not be considered safe, and the programmer would be denied readily available aids to achieving correctness. By limiting this delay to **only** the points where it is logically required and focussing the checking at that point, the programming language is giving the best support for this type of programming. That strategy was adopted by PS-algol, Amber and Napier. In that case the rest of the program is statically checked, and as the binding is made, all the checks that need to be applied to ensure the expectations in the following code will be met, are applied at the moment of binding. We call this *eager checking* (doing as much as possible as soon as possible). It has the advantage that if an exception is raised indicating the binding cannot be made because the value doesn't match, then the programmer has a clear idea of the state of the computation.

To permit the program to be statically checked everywhere else, and to retain strong type checking over delayed binding, it is necessary to record the expected properties of the value. Since they must have been available when the value was created this is extra work. It is the avoidance of this extra work which gives the immediate appeal when programming on those systems which save their workspaces. References to *include* the source (e.g. from a file), and cut-and-paste editors only partially solve this problem. Firstly, they tend to result in including in the interface specification details not used in this particular binding, thus over-constraining the binding against data evolution. Secondly, if the data and programs are really independent, e.g. from different sites, there isn't a common context from which to cut-and-paste or *include*. Thus, duplicate definition is an inevitable requirement of real independence.

Implementation costs divide into additional space to store the duplicate data descriptions and additional processing to perform checks every time the binding is made. As programming and data management staff costs rise relative to memory and cpu costs, this becomes less significant. In any case, much of the description needs to be stored with data for other reasons, such

as space management and diagnostics. The additional checking should be limited to that which is essential, as is possible in some of the languages surveyed, Amber and Napier, the rest should be factored out at compilation time. Using data flow analysis, many of the remaining checks, left as delayed by the programmer, may be elided by the optimiser. Recompilation each time a definition changes may actually cost more.

We regard it as appropriate for the programmer (or applications systems builder) to have control over where the delayed binding will occur, and where static binding will be used. Only such persons have the opportunity to weigh the relative merits of economy in coding and execution of static binding, versus the flexibility of delayed binding. This balance depends on aspects of the application. We do not believe it is possible for the language designer to make an a priori choice which is appropriate (or even acceptable) for all applications.

6. Type checking issues

Delayed binding should not, and need not, weaken the strength of type checking, only modify its timing. But, the requirement of independence does have implications for type checking.

Two forms of type equivalence are used in languages: *name equivalence* and *structural equivalence* . For name equivalence, two values are of the same type if their type is defined by the same evaluation of the same type expression. In practice this means that when a type expression is evaluated some data structure is constructed, and the type is identified by a reference to this data structure. Two types are then equivalent if they are represented by the same reference value. If data and program are to be combined that originate from independent sources (e.g. different sites) then they cannot contain references that are equal. Hence, name equivalence cannot be used in DBPLs supporting the flexibility required.

Structural equivalence is more expensive to compute. Associated with a type is an encoding of the expression that generated the type, from the base types by recursive application of the type construction operators. To test structural equivalence, one investigates the two encoded expressions (usually by converting them to a canonical form) to determine whether they generate the same set of possible values. Though more computationally expensive, it meets the need for independence.

These concerns are by no means parochial to DBPLs. Pascal (section 4.1) uses name equivalence; but a binding mechanism for its files, if implemented properly, would need to use delayed binding and structural equivalence. Ada [Ichbiah *et al.* 79] faces similar problems (section 4.2). It takes more care than Pascal to specify persistence, defining IO packages that will form delayed bindings to files, but the name equivalence it uses can **not** be applied in this context and it fails to specify any alternative. If Pascal/R had been modified, as suggested, to allow access to multiple databases it would have had to abandon Pascal's method of type checking to meet requirement 3.3.

Further investigation of the interaction between type checking and persistence is required. For example, in languages like PS-algol, Amber and Napier, names are embedded in type structures as substructure labels, and are required to be identical for structural equivalence. What are the most convenient natural rules for identity of labels? A single universe of names is unmanageable.

In PS-algol and Napier type checks may be delayed until a part of the structure they describe is reached. This permits equivalence of types to be computed incrementally, which may have significant savings for a binding between a database and a typical program which only touches a small part of it. The utility of this incremental approach requires evaluation. Optimisations, e.g. not rechecking type matches that cannot have been invalidated, also need development [Owoso 84].

7. Conclusions

We have shown that normal database applications require a flexibility in binding time. This can be met by constructs in a programming language which allow specification of delayed binding so that the rest of the program can have the benefit of static binding. Examples of languages show there are a variety of constructs to be explored. That this is an issue is illustrated by the absence of these considerations in production languages (e.g. Ada) and in leading database research languages (e.g. Galileo).

The present constructs in Napier meet all the flexibility requirements except the need to replace the definitions of ADTs without loss of their data. New research is required here, while continued research is needed to refine the definition of the other constructs.

8. Acknowledgements

We acknowledge the observation by David Watt which drew our attention to the existence of these problems in the present Ada and Pascal standards. We also acknowledge the support of Alvey, the British Science and Engineering Research Council, the ICL University Research Council and the NSF in our work.

References

[Albano *et al.* 85a]
Albano, A., Cardelli, L. and Orsini, R.,
Galileo: A Strongly Typed Interactive Conceptual Language. ACM Transactions on Database Systems, Vol. 10, No. 2 (June 1985) 230-260.

[Albano *et al.* 85b]
Albano, A., Occhiuto, M.E. and Orsini, R.
Galileo reference manual VAX™/UNIX™ Version 1.0 , Universita Degli Studi di Pisa, Dipartimento di Informatica 1985.

[Albano *et al.* 86]
Albano, A., Ghelli, G., Occhiuto, M.E. and Orsini, R.
A Strongly Typed, Interactive Object Oriented Database Language, in *proceedings of 1986 International Workshop on Object-Oriented Database Systems* , Computer Society Press (September 1986) 94-103.

[Atkinson *et al.* 83]
Atkinson, MP., Bailey, PJ., Chisholm, KJ., Cockshott, W.P and Morrison, R.
An approach to persistent programming, The Computer Journal, Vol. 26, No. 4 (1983) 360-365.

[Atkinson & Morrison 85a]
Atkinson, M.P. and Morrison, R.
Procedures as persistent data objects, *ACM transactions on Programming Languages and Systems* , Vol. 7, No. 4 (October 1985) 539-559.

[Atkinson & Morrison 85b]
Atkinson, M.P. and Morrison, R.
Types, Bindings and Parameters in a Persistent Environment in *Proceedings of Persistence and Data Types Workshop* , Appin, Scotland (August 1985) 1-24.

[Atkinson & Morrison 86]
Atkinson, M.P. and Morrison, R.
Integrated persistent programming systems. In *proceedings of the 19th annual Hawaii international conference on System Sciences* , Vol. IIA, (January 1986) 842-854.

[Atkinson & Morrison 87]
Atkinson, M.P. and Morrison R.
Names and Name sets in Napier, in preparation.

[Atkinson *et.al.* 86]
Atkinson, M.P., Morrison, R. and Pratten, G.D.
Designing a persistent information space architecture, In *proceedings of Information Processing 1986,* North Holland Press (Sept. 1986) 115-119.

[Atkinson & Buneman 87]
Atkinson M.P. and Buneman, O.P.
Types and Persistence in Database Programming Languages, submitted to *ACM Computing Surveys.*

[Birrell & Nelson 83]
Birrell, A.D. and Nelson, B.J.
Implementing Remote Procedure Calls CSL-83-7 Xerox PARC (December 1983).

[Buneman *et al.* 82]
Buneman, O.P., Hiirschberg, J and Root, D.
A CODASYL interface to Pascal and Ada In *Proceedings of the second British National Conference on Databases* : Bristol, England, July 1982.

[Burstal *et al.* 71]
Burstal, R.M., Collins, J.S. and Popplestone, R.J.
Programming in POP-2 Edinburgh University Press, Edinburgh 1971.

[Cardelli 84a]
Cardelli, L.
Amber , A.T. & T. Bell Laboratories (1984).

[Cardelli 84b]
Cadelli, L.
A semantics of Multiple Inheritance *Semantics of Data Types: International Symposium, Sophia-Antipolis* . Springer-Verlag, Berlin, 1984, pages 51-67.

[Cardelli & Wegner 85]
Cardelli, L. and Wegner, P.
On understanding Types, Data Abstraction and Polymorphism, *ACM Computing Surveys* , Vol. 17, No. 4 (December 1985) 471 -523.

[Cooper *et al.* 87]
Cooper, R.L., Atkinson, M.P., Abderrahmane, D. and Dearle, A.
Constructing database systems in a persistent environment, to appear in the proceedings of the thirteenth international conference on Very Large Databases, Brighton, England (September 1987).

[Clocksin & Mellish 81]
Clocksin, W.F. & Mellish, C.S.
Programming in Prolog Springer-Verlag, Berlin 1981.

[Goldstein & Bobrow 80]
Goldstein, I.P. & Bobrow, D.G.
Extending object oriented programming in Smalltalk. In *proceedings of the 1980 Lisp Conference* (August 1980) 75-81.

[Ichbiah *et al.* 79]
Ichbiah *et al.*
Rationale of the Design of the Programming Language Ada. ACM SIGPLAN Notices, Vol. 14, No. 6 (1979).

[ISO 82]
International Standards Organisation,
Standard specification for the programming language Pascal , ISO 7185, 1982.

[Iverson 79]
Iverson, KE.
Operators. *ACM transactions on Programming Languages and Systems.* Vol. 1, No. 2 (October 1979) 161-176.

[Joy *et al.* 83]
Joy, W.W.*et al.*
Berkeley Pascal User's Manual Version 3.0 - in *Vol. 2.C Unix Programmers' Manual Berkeley 4.2* , July 1983.

[Koch *et.al.* 83]
Koch, J., Mall, M., Putfarken, P., Reimer, M., Schmidt, J.W. & Zehnder, C.A.
Modula/R report. Lilith version
Technical Report, Eidgenössische Technische Hochschule Zürich, Institue Für Informatick, 1983.

[Liskov *et.al.* 81]
Liskov, B, *et al.*
Lecture notes in computer science. Volume 114: *CLU reference manual* Springer-Verlag, Berlin, 1981.

[Liskov *et.al.* 83]
Liskov, B., Herliby, M., Johnson, P., Leavens, G., Scheiffler, R. & Weihl, W.
Preliminary Argus reference manual Technical Report, Memo 39, Programming Methodology Group, MIT, Cambridge, Massachusetts 02139, USA, October 1983.

[Matthews 85]
Matthews, D.C.J. An overview of the Poly Programming Language, in *Proceedings of Persistence and Data Types Workshop* , Appin, Scotland (August 1985) 255-264.

[McCarthy *et.al.* 62]
McCarthy, J., *et.al.*
Lisp 1-5 Programmer's Manual. MIT Press, Cambridge, Massachusetts, 1962.

[Milner 78]
Milner, R.
A theory of type polymorphismn in programming, *Journal of Computer and System Science*, Vol. 17 (1978) 348-375.

[Milner 84]
Milner, R.
A proposal for standard ML, in *proceedings of the 1984 ACM symposium on Lisp and Functional programming,* ACM, 1984.

[Owoso 84]
Owoso, G.O.
Data description and manipulation in persistent programming languages , Ph.D. thesis, University of Edinburgh, 1984.

[PPRG87]
Persistent Programming Research Group, *PS-algol reference manual - Third edition* , PPRR12 (January 1987), Universities of Glasgow and St. Andrews.

[Rowe & Shoens 79]
Rowe, L. & Shoens, K.
Data Abstraction, Views and Updates in Rigel. In *proceedings of ACM SIGMOD international conference on management of data* (1979) 71-81.

[Schmidt 77]
Schmidt, J.W.
Some High level Language Constructs for data of type Relation. *ACM transactions on database systems.* Vol. 2, No. 3 (September 1977) 247-281.

[Tennent, 81]
Tennent, R.D. *Principles of Programming Languages* Prentice/Hall, 1981.

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)

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, Septermber 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 9th July 1986. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., 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-31-86	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00
PPRR-34-87	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